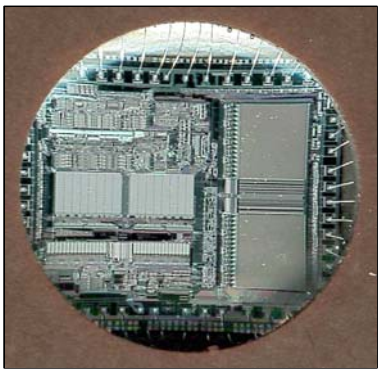
UNIVERSITÄT KARLSRUHE



Institut für Industrielle Informationstechnik

Hertzstr. 16, Geb. 06.35

Prof. Dr.-Ing. habil. K. Dostert

Skriptum zur Vorlesung

# Mikrorechnertechnik

Sommersemester 2007

# Mikrorechnertechnik (MRT)

Sommersemester 2007

# 1 Vorwort

Der Mikrorechner hat wie kaum ein anderer Baustein die Entwicklung der Technik im letzten Drittel des 20. Jahrhunderts beeinflußt. Im Bereich der Elektrotechnik ist er nicht nur bei der Datenverarbeitung, der Nachrichten- und Hochfrequenztechnik unentbehrlich, sondern auch in der Starkstromtechnik, der Leistungselektronik und der Kraftwerkstechnik. Auch aus den meisten Produkten des Maschinenbaus wie z.B. Kraftfahrzeuge, Flugzeuge oder Industrieanlagen aller Art sind Mikrorechner nicht mehr wegzudenken. Sie bilden die „Hirne“ der Maschinen und ermöglichen effiziente Energienutzung, Präzision und exakte Kontrolle auch bei hohen Geschwindigkeiten und garantieren in hohem Maße die Systemzuverlässigkeit.

Die folgende, keineswegs vollständige Liste vermittelt einen Eindruck der Vielfalt der Anwendungsgebiete von Mikrorechnern:

- Automatisierung: Steuerung bewegter Systeme (Antriebe, Fahrzeuge, Roboter)
- Kraftwerkstechnik, Energieerzeugung, erneuerbare Energien (Wind, Photovoltaik)
- Industrielle Produktion, Chemie/Verfahrenstechnik
- Fernmessen und Fernwirken Meßdatenerfassung/Verarbeitung/Auswertung
- Umwelttechnologie, Verkehrsleitsysteme, Weltraumtechnik
- Funkortung und Navigation (Radar, GPS)
- Medizintechnik
- Kommunikation: Signalverarbeitung und Systemsteuerung in Funk- und Leitungsnetzen

Insbesondere der Bereich Kommunikation kann wiederum in eine Vielzahl von Teilaufgaben, wie z.B. Filtern, Korrelieren, Transformieren, Kodieren/Dekodieren, Modulieren/Demodulieren Komprimieren/Dekomprimieren, Fehlerkorrektur oder Signalsynthese aufgespaltet werden. Bei diesen Aufgaben mit „harten“ Realzeitanforderungen zeigen sich wohl am schnellsten die Grenzen der Leistungsfähigkeit heutiger Mikrorechner. Deshalb ist sicher, daß die Entwicklung noch keineswegs als abgeschlossen betrachtet werden kann. Künftig wird sich das, was wir heute als typischen Mikrorechner klassifizieren, mehr und mehr von einem in großer Stückzahl gefertigten Standardbauteil weg zu vielfältigen anwendungsspezifischen Lösungen hin bewegen. Auf den Ingenieur kommen damit neue Aufgaben zu, die über das Verstehen der Arbeitsweise, die Programmierung und die richtige Einschätzung der Leistungsfähigkeit heutiger als Universalprodukte angebotener Mikrorechner weit hinausgehen.

Zur Vorbereitung auf solche Aufgaben bietet diese Vorlesung zunächst eine grundlegende Behandlung der Hardwarearchitekturen und Datenverarbeitungsprinzipien sowie eine Einführung in Entwicklungsumgebungen gängiger Mikrorechner. Dabei wird der Bogen vom Mikroprozessor über den Mikrocontroller hin zum digitalen Signalprozessor (DSP) geschlagen, und es werden die Gemeinsamkeiten und die signifikanten Unterschiede herausgearbeitet.

Ausgehend von den gängigen Mikrorechnerarchitekturen erfolgt der wichtige Schritt hin zum Entwurf anwendungsspezifischer Digitalschaltungen, mit dem Ziel, die nötigen Fähigkeiten zum Aufbau eigener, den jeweiligen technischen Problemstellungen angepaßten Lösungen in Form von "Application Specific Integrated Circuits“ (ASICs) zu vermitteln. Dies ist heute kein langwieriger, kostenintensiver Prozeß mehr, d.h. selbst ein hochkomplexes ASIC, z.B. in Form eines FPGA ≡ Field Programmable Gate Array, kann mit preisgünstiger PC-Ausstattung am Arbeitsplatz des Ingenieurs vollständig entwickelt werden.

## 1.1 Historische Entwicklung der Technik digitaler Rechner und Automaten

Altertum: ABAKUS (Rechenhilfe)

Mechanische Additionsmaschinen (vor etwa 300 Jahren):

- Schickard'sche Rechenuhr 4 Grundrechenarten
- (Prof. für biblische Sprachen in Tübingen, 1623)
- PASCAL (1623-1662) Radgetriebe

Programmsteuerung (seit etwa 100 Jahren):

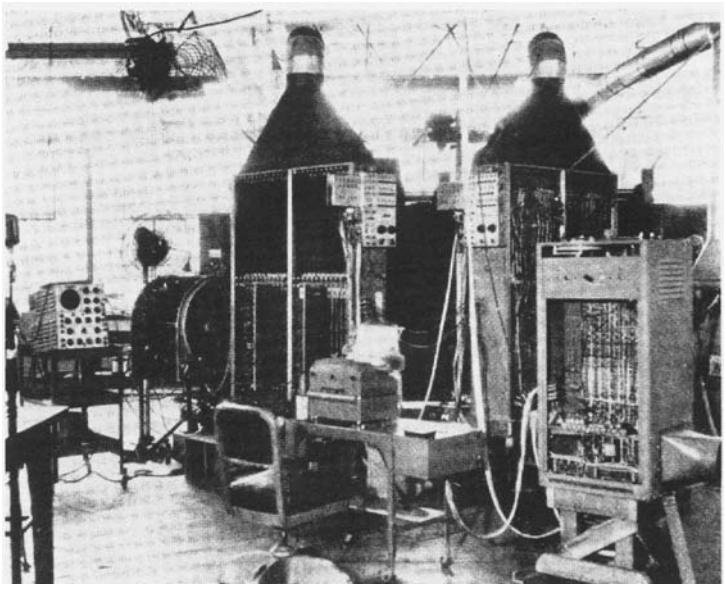
- England: Babbage (1792-1871): Pläne für Rechenwerk, Steuerwerk, Speicher
- Hollerith (1900): Lochkarten; 1928 erste funktionsfähige Maschinen

**Elektrische Rechner:**

Erster Relais-Rechner mit Festprogrammierung: Konrad Zuse (Bauingenieur) ca. 1935
Prof. AIKEN ca.1945

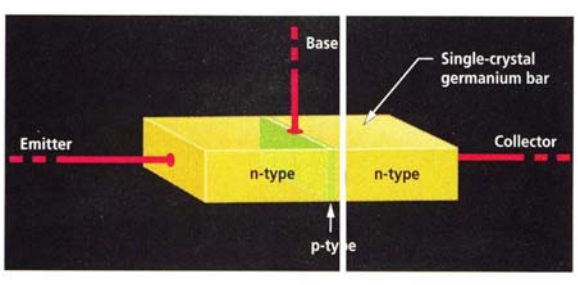
**Elektronische Rechner:**

Erster Röhrenrechner: ENIAC (Univ. of Pennsylvania, 1946) fürs Militär



- Flip-Flops mit Röhren aufgebaut
- Kernspeicher (Ferritringe als Arbeitsspeicher)

Bild 1.1: Foto des ENIAC



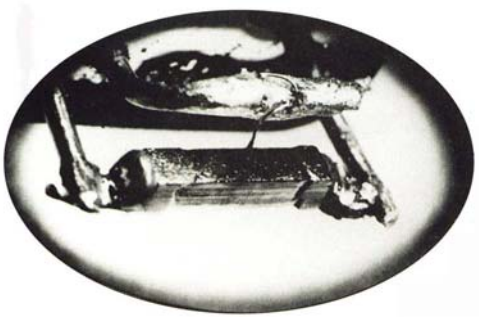



Bild 1.2: Erster Transistor: **Shockley, Bardeen, Brattain (1948)**

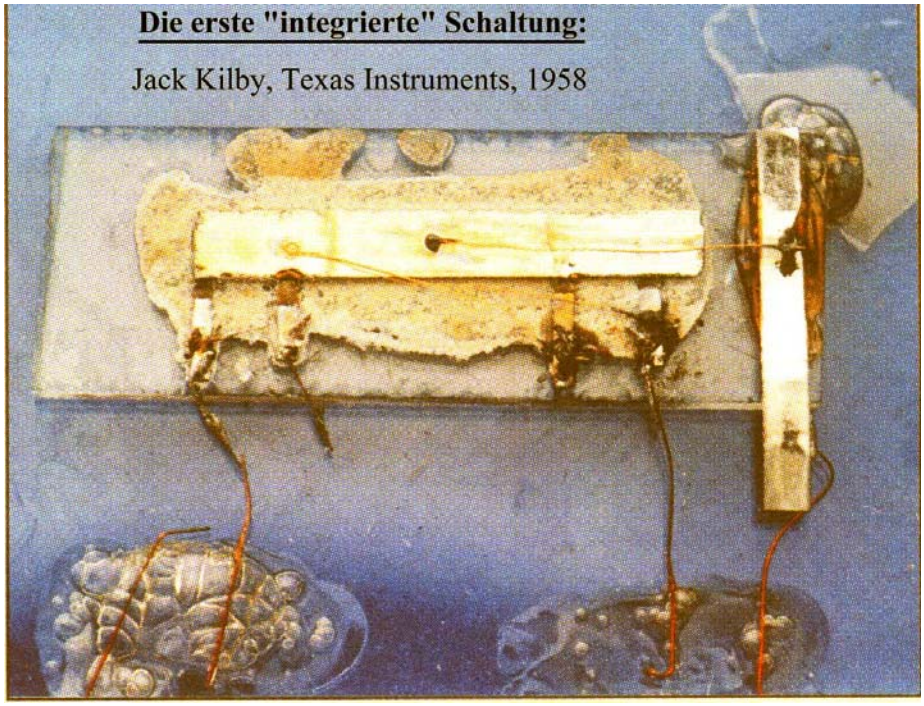
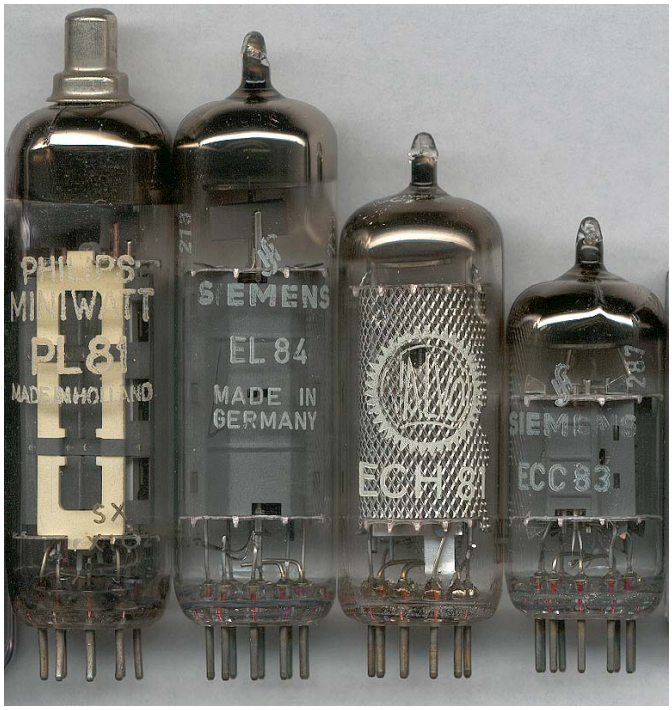


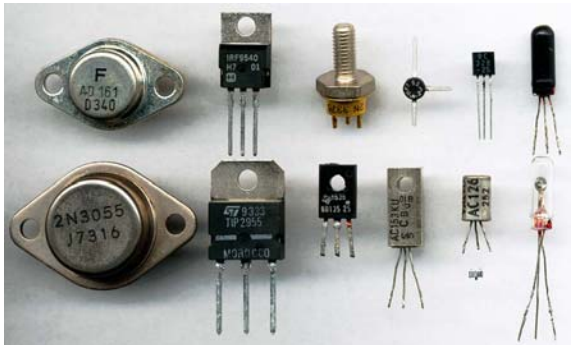

Bild 1.3: Der Beginn der Integrationstechnik

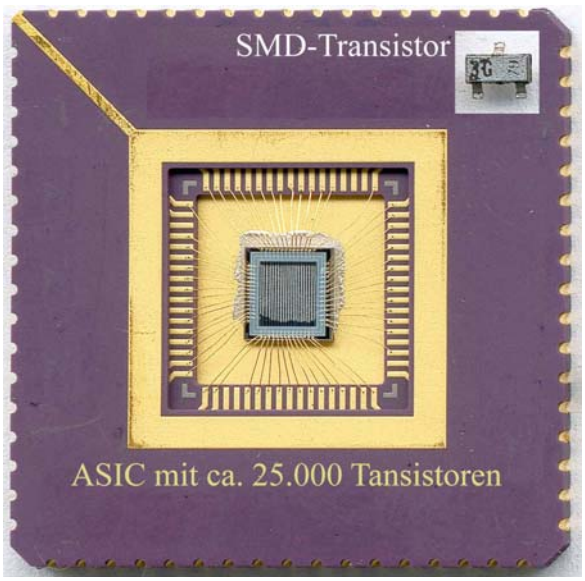
Meilensteine der Integrationstechnik:

- 1958 Jack Kilby (Texas Instruments): Multivibrator mit 4 Transistoren
- 1993 über 3 Mio. Transistoren auf einem Chip (Intel Pentium-Prozessor)
- 2004 AMD Athlon: 40 Mio. Transistoren, Intel Pentium 4: ca. 50 Mio. Transistoren

Bild 1.4: Bauteile der Elektronik



### 1.1.1 Vergleich von Analog- und Digitalrechner

Analogrechner:

- Es werden Schaltungen aufgebaut, bei denen Spannungen, Ströme über Differentialgleichungssysteme verknüpft sind
- Man hat gleiche Gesetzmäßigkeiten wie in vielen anderen physikalischen Systemen, z.B. mechanischen.
- Die Simulationsmodelle sind sehr anschaulich und der Denkweise des Ingenieurs angepaßt.
- Die Programmierung eines zu lösenden Problems erfolgt über einfache lineare Transformationen, d.h. es müssen im wesentlichen Maßstabsfaktoren bestimmt und eingestellt werden
- Die Ausgabe von Ergebnissen geschieht am Oszillographen, über Plotter oder X-Y-Schreiber.

Digitalrechner:

- dimensionslose Zahlen mit vielen Stellen als Rechengrößen
- Wertebereiche der physikalischen Größen müssen passend transformiert werden
- die Genauigkeit ist durch Rechenzeit praktisch beliebig steigerbar
- Informationen können gespeichert, sortiert und logisch verknüpft werden
- kein Genauigkeitsverlust bei Datenübertragung
- Programmierung ist aufwendig und es werden gute Algorithmen benötigt
- in der Regel ein einziges Rechenwerk, das seriell arbeitet, verfügbar
- Simulation und Regelung in Echtzeit sind problematisch

Tabelle 1.1: Vergleich von „Computergenerationen“

| | ENIAC (1946) | 8080-System (1973) | Pentium 4 im PC (2004) |
|---|---|---|---|
| Volumen: | 100 $m^3$ | 0,3 l | 40l |
| Masse: | 30.000kg | 500g | 20kg |
| Leistungsaufnahme: | 174kW | 2,5W | 300W |
| ROM: | 16kbit | 16kbit | |
| RAM: | 1kbit | 8kbit | >4 Gbit |
| Akt. Elemente: | 18.000 Röhren | 20.000 Transistoren | > 50 Mio. Transistoren |
| Addieren von 2 12-stell. Ziffern | 200µs | 100µs | < 250ps |
| MTBF | 1h | > 1 Jahr | > 1 Jahr |
| Kosten: | 500.000 € (Material) | ca. 50 € | ca. 1000 € |

## 1.1.2 Digitale Rechenmaschinen für die Signalverarbeitung: Historische Übersicht

Mathematische Modellierung kontinuierlicher Signale:

**Laplace** ⎫
⎬ Transformation: Veröffentlicht im 19. Jahrhundert
**Fourier** ⎭

Jean Baptiste Joseph, Baron de **Fourier** war Gouverneur in Unterägypten unter Napoleon

- 1801 Rückkehr nach Frankreich
- 1822 umfangreiches Werk über Wärmefluß an Kanonenrohren
  → *Theorie der Fourierreihe*
  → später: Diskrete FT (DFT)

Pierre Simon, Marquis de **Laplace** war theoretischer Astronom

- um 1800: Veröffentlichung der *Laplacetransformation*
  später: Diskrete LT (z-Transformation)

- 1965 Cooley & Tuckey FFT-Algorithmus *Fast Fourier Transform* ≡ schnelle FT
  (Ursprünge finden sich bei den Mathematikern Runge und Gauß ≈ 1900)

- 1980 - 1982 die ersten „Digitalen Signalprozessoren“
  - ⇒ AMI S2811
  - ⇒ Intel 2920
  - ⇒ NEC µPD7720
  - ⇒ TI TMS32010

Heute:

Motorola

- DSP 56300 24bit-Festkomma-DSP (Motorola) 100MHz ⇔ 100MIPS[1]
- MSC8101 **StarCore SC140** 1200 **True**[2] DSP MIPS oder 3000 RISC MIPS bei 300 MHz Taktfrequenz

Texas Instruments

- TMS 320C6201 „16bit“-Festkomma-DSP 200MHz ⇔ 1600MIPS
- TMS320DM642 – max. 600 MHz Taktfrequenz ⇔ 4800 MIPS

[1] Mega Instructions per Second (Millionen Befehlsabarbeitungen pro Sekunde)
[2] “True DSP MIPS” sind Multiply & Accumulate (MAC) Operationen, inklusive den zugehörigen Datentransporten und dem Aufdatieren der Zeiger (Pointer)

## Analog Devices

- ADSP-BF53x (Blackfin); Taktfrequenz: bis zu 600 MHz ⇔ 1200 MMACS
  (MMACS – Mega MAC-Operationen pro Sekunde ≡ $10^6$ MAC/s)

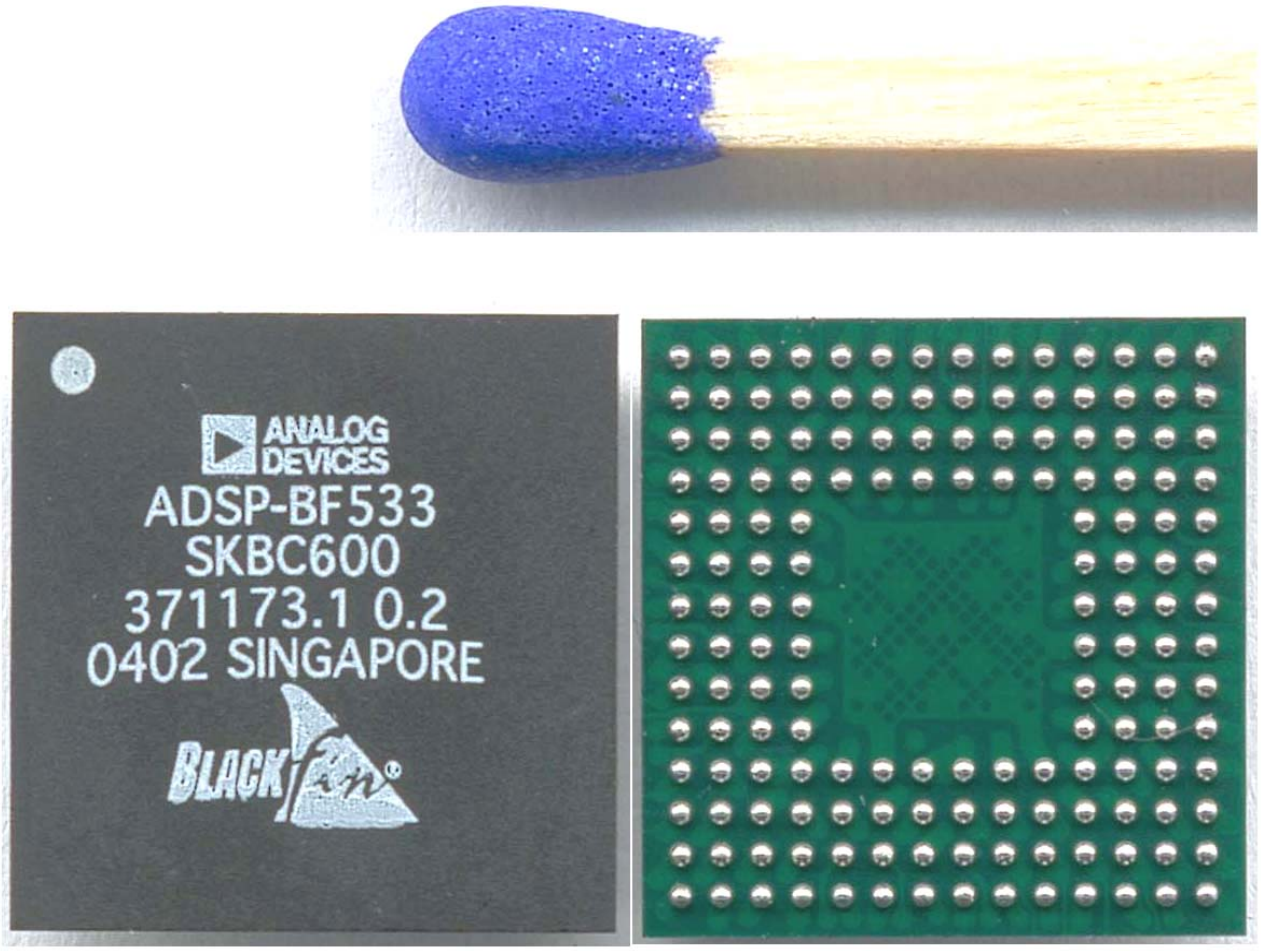



Bild 1.5: Blackfin-Prozessor im „Ball Grid Array“ (BGA) Gehäuse

# 2 Grundelemente und Technologie der Mikrorechner

## 2.1 Übersicht

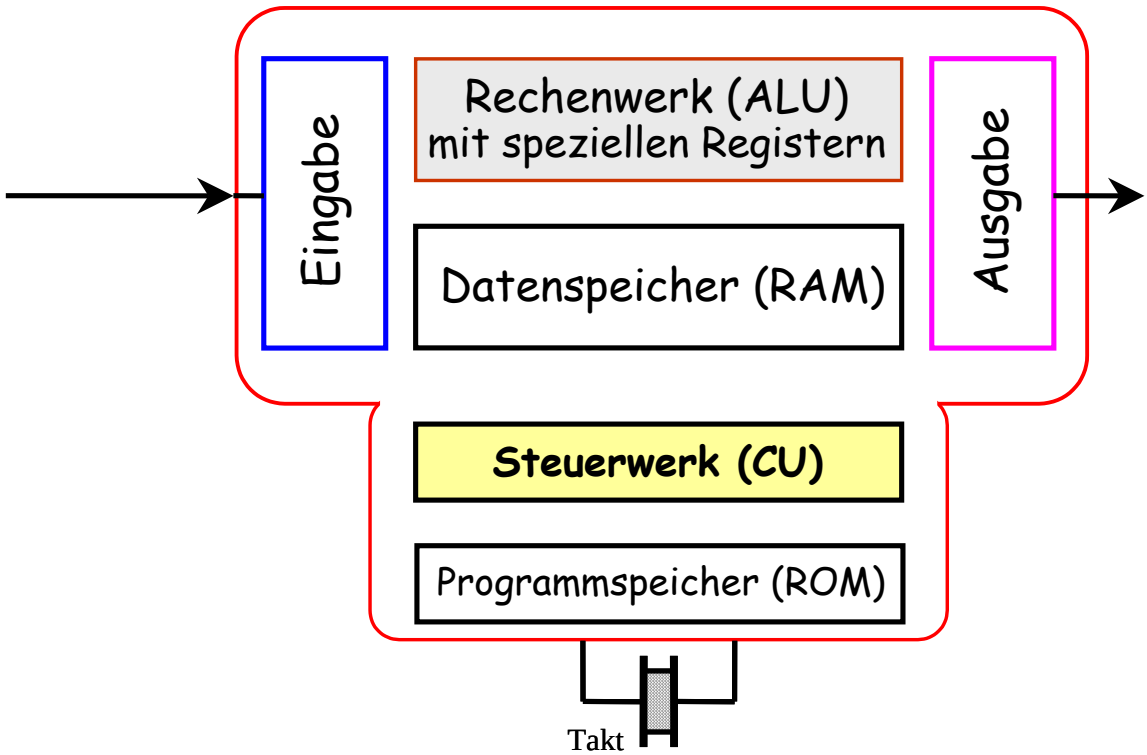



Bild 2.1: Grundstruktur eines Mikrorechners

Die ersten Pläne programmierbarer Rechner stammen von dem Engländer Babbage (1792-1871). Sie umfaßten bereits die Komponenten Rechenwerk, Steuerwerk und Speicher, die bis heute Kernbestandteile aller digitalen Rechenmaschinen sind. Eine technische Umsetzung der Pläne war zu Babbage's Lebzeiten noch in weiter Ferne. Erst die sich rasant entwickelnde Halbleitertechnik schuf mit Begin der 60er Jahre des vorigen Jahrhunderts schrittweise die Grundlagen, Rechner mit einer Struktur nach Bild 2.1 zu realisieren. Es war schon frühzeitig klar, daß nur Elektrotechnik und Elektronik zur Realisierung von Digitalrechnerfunktionen hinreichend Aussicht boten. Dennoch hatten eine Zeitlang auch strömungsmechanische Umsetzungen in Form sogenannter pneumatischer Rechner eine gewisse Bedeutung, z.B. zu Automatisierungszwecken in der chemischen Verfahrentechnik. In explosionsgefährdeten Bereichen konnten die frühen elektronischen Rechner wegen ihres hohen Energiebedarfs nicht eingesetzt werden. Die benötigten Leistungen hätten schon bei geringfügigen elektrischen Fehlern explosionsfähige Gemische zünden können.

Auch heute kann noch keineswegs jeder beliebige Mikrorechner in solch kritischen Umgebungen eingesetzt werden. Man hat jedoch eine reiche Auswahl geeigneter Bausteine z.B. in Form von Mikrocontrollern, deren Energiebedarf deutlich unterhalb der kritischen Grenzen liegt. Trotz aller Fortschritte hat sich aber an dem Grundsatz nichts geändert, daß die Leistungsfähigkeit eines Rechners eng mit der Aufnahme elektrischer Leistung korreliert ist, die im wesentlichen in Form von Wärme abgeführt werden muß – vgl. auch Tabelle 1.1.

Wie aus Bild 2.1 hervorgeht, sind für die Funktion eines elektronischen Rechners neben den schon genannten Bestandteilen noch eine Taktquelle (in der Regel ein Schwingquarz oder ein keramischer Resonator) sowie Ein- und Ausgabeeinheiten erforderlich. Der Anwender von Mikrorechnern hat so normalerweise Bausteine (auch Chips genannt) mit einer Vielzahl von Anschlußpins in verschiedenen Gehäuseformen vor sich. Es ist mittlerweile ausgesprochen schwierig und zeitraubend, eine Übersicht zu gewinnen und aufrecht zu erhalten.

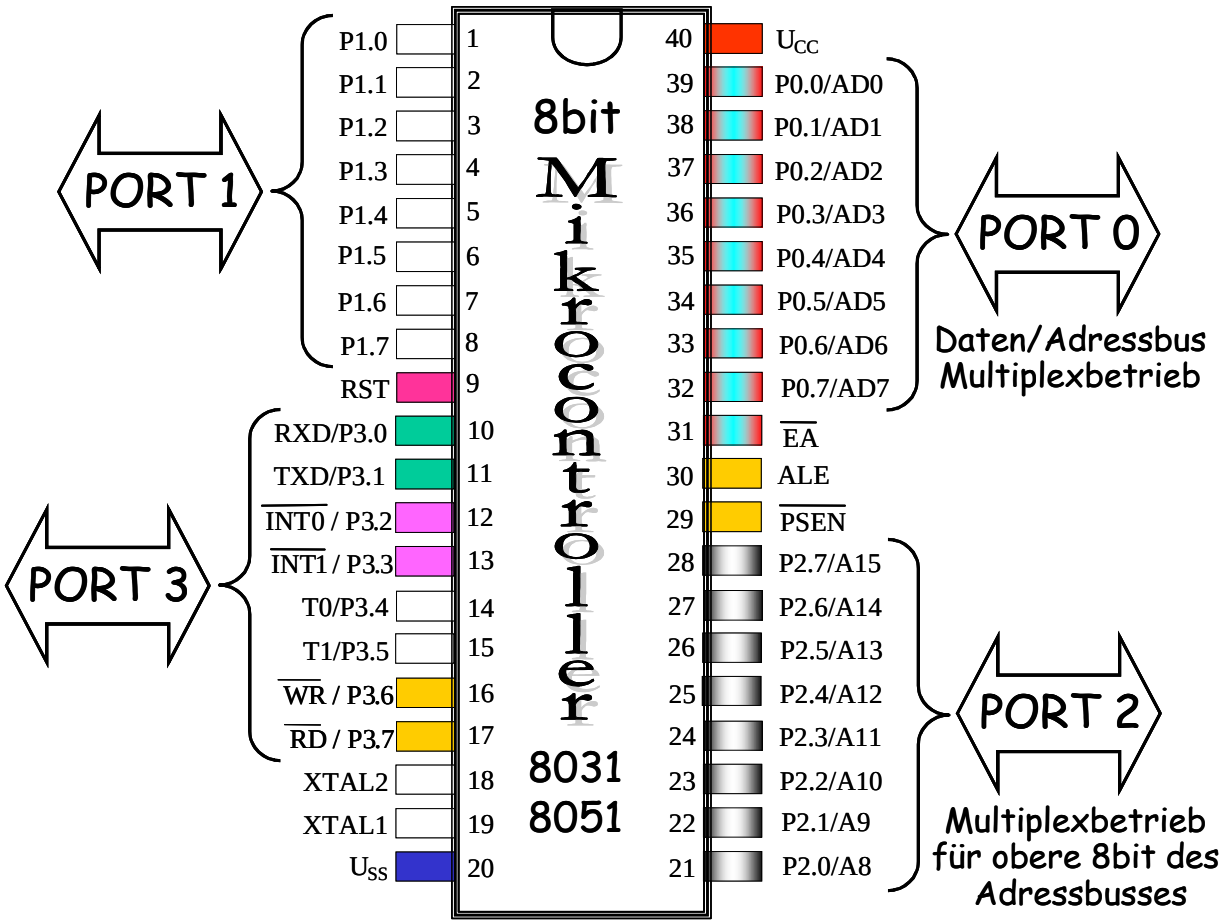



Bild 2.2: Pinbelegung eines einfachen 8bit-Mikrocontrollers in einem DIL-20-Gehäuse

In den 80-er Jahren des vorigen Jahrhunderts setze auf breiter Front ein der Siegeszug der sogenannten Mikrocontroller ein. Eine typische Bauform war das Dual-In-Line (DIL) Gehäuse – s. Bild 2.2. Ein Mikrocontroller nach Bild 2.2 stellt einen vollständigen, ohne externe Bauteile (mit Ausnahme des Taktgebers), funktionsfähigen Rechner dar. Die hohe Anzahl von Portleitungen, die wahlweise in beiden Richtungen, d.h. als Ausgänge oder Eingänge benutzbar sind, ermöglichte vielfältige Anwendungen in der Automatisierung, angefangen von der industriellen Fertigungstechnik, über Kraftfahrzeuge bis hin zu Hausgeräten und Unterhaltungselektronik. Auch heute werden 8051-Controller noch in beträchtlichen Stückzahlen in Computertastaturen verbaut und bilden – wie wir später noch sehen – die Kerne komplexer Systeme mit umfangeichen Peripheriekomponenten in Form von „Mikroconvertern“.

Mikrorechner in der Bauform nach Bild 2.2 wurden häufig in Sockeln plaziert und nicht unmittelbar in die Anwenderplatine eingelötet. Denn für die Programmierung wurden besondere Geräte benötigt, die freien Zugang zu Adreß- und Datenleitungen erforderten. Zum Einschreiben eines Programms war daher das Herausnehmen des Bausteins aus der Anwenderschaltung nötig. Wie Programme entwickelt werden und in die Mikrorechnerhardware gelangen oder auch geändert werden können, wird in späteren Kapiteln detailliert behandelt. Im Vergleich zur Programmierung eines Personalcomputers (PC) oder Notebooks werden hier andere Methoden und Hilfsmittel benötigt.

Während der Programmierer eines PC sich weder Gedanken über eine Trennung von Programm- und Datenspeicher macht, noch deren genaue Bedienung auf elektrischer Ebene kennen muß, ist die Kenntnis und Beherrschung dieser Zusammenhänge für die Mikrorechneranwendungen, wie wir sie hier behandeln wollen, unabdingbar.

Im Gegensatz zum PC verfügt der Mikrorechner – auch wenn es ein Mikrocontroller mit allen zum Betrieb erforderlichen Hardwarekomponenten ist – nicht über ein Betriebssystem, das z.B. unmittelbar den Transport von Information, d.h. das Laden von Programmen und Daten von Speichermedien wie Disketten, Festplatten oder USB-Sticks ermöglicht. Bei einem auf einer Anwenderplatine verbauter Mikrocontroller wird nach dem Anlegen der Betriebsspannung allenfalls der Taktoszillator laufen und es werden ggf. bestimmte Portleitungen – vom Steuerwerk ausgelöst - eine gewisse Aktivität zeigen, die in der Regel in der Ausgabe regulärer Impulsmuster besteht.

Die gewünscht Funktion kann der Mikrorechner erst dann ausführen, wenn in seinen, für den Anwender meist nicht sichtbaren, internen Programmspeicher ein geeignetes Programm eingeschrieben wurde. Der Programmspeicher ist daher normalerweise vom Typ ROM (Read Only Memory), d.h. nach "einmaligem" Beschreiben nur noch lesbar. Wichtig ist, daß das Programm nach Abschalten der Stromversorgung erhalten bleibt und nach dem Einschalten sofort wieder zur Verfügung steht. Ein "Booten" von einem "Massenspeichermedium" – wie es ein PC oder Notebook ausführt – kennt der Mikrorechner nicht, die dazu nötigen Ressourcen wären auch nicht vorhanden.

Während der PC über einen einzigen flüchtigen[3] Arbeitsspeicher beträchtlicher Größe vom Typ RAM (Random Access Memory ≡ Speicher mit wahlfreiem Zugriff - lesend und schreibend) verfügt, in dem Programme und Daten abgelegt werden, hat ein Mikrocontroller oder ein DSP meist erheblich kleinere Einheiten mit einer klaren Trennung von Programm- und Datenspeicher. Damit entfällt eine nicht unerheblicher "Verwaltungsaufwand" mit dem beim PC dafür gesorgt werden muß, daß beim Ablauf eines Programms anfallende Daten keinesfalls die im selben Speicher befindlichen Programmteile überschreiben. So etwas führt in der Regel nicht mehr auffangbaren "Abstürzen", die bei älteren Windows-Versionen mit der Fehlermeldung „allgemeine Schutzverletzung" endeten und einen kompletten Neustart des Systems erforderlich machten.

Die Verwendung separater Speicher für Programm und Daten ist bei Mikrocontrollern und DSPs aufgrund ihrer typischen Anwendungsgebiete praktisch vorgegeben. Speziell bei DSPs sind – wie wir noch im Detail sehen werden – parallele Transfers von Information unabdingbar, d.h. neben dem Programmcode (auch OP-Code genannt) müssen Operanden (Daten) gleichzeitig dem Rechnerkern (auch CPU ≡ Central Processing Unit genannt) zugeführt werden. Das bedeutet, daß nicht nur separate Speicher vorhanden sein müssen, sondern auch die Ressourcen für den Zugriff darauf. Man spricht dabei von "Bussen" und unterscheidet drei Kategorien

- Adreßbus – ermöglicht die Selektion eines Speicherplatzes (unidirektional)
- Datenbus – ermöglicht den Transfer von Information (bidirektional)
- Steuerbus – regelt den Zugriff auf den Datenbus, mit dem Ziel, Kollisionen auszuschließen

Bild 2.3 zeigt eine typische Bussystemkonfiguration, mit der nicht nur der Transfer von Information zwischen einem Prozessor und seinen Speichern, sondern auch zwischen weiteren „Peripheriekomponenten" wie z.B. Analog-Digitalwandlern, Digital-Analogwandlern oder Ein/Ausgabeports bewerkstelligt werden kann. Bei einem Mikrocontroller befindet sich alles was in Bild 2.3 dargestellt ist auf einem Chip, d.h. monolithisch integriert auf Silizium. Wir wollen im weiteren unter dem Oberbegriff Mikrorechner folgende Varianten unterscheiden:

- Mikroprozessor – auch CPU genannt – hierbei fehlen Speicher und Peripheriekomponenten
- Mikrocontroller – enthält alle Komponenten eines selbständigen Rechnersystems
- DSP ≡ digitaler Signalprozessor – bietet besonders hohe Rechenleistung, gekoppelt mit schnellen, parallelen Datentransfers für anspruchsvolle „Echtzeitaufgaben"

[3] das bedeutet, daß bei Ausschalten der Stromversorgung der Inhalt verloren geht

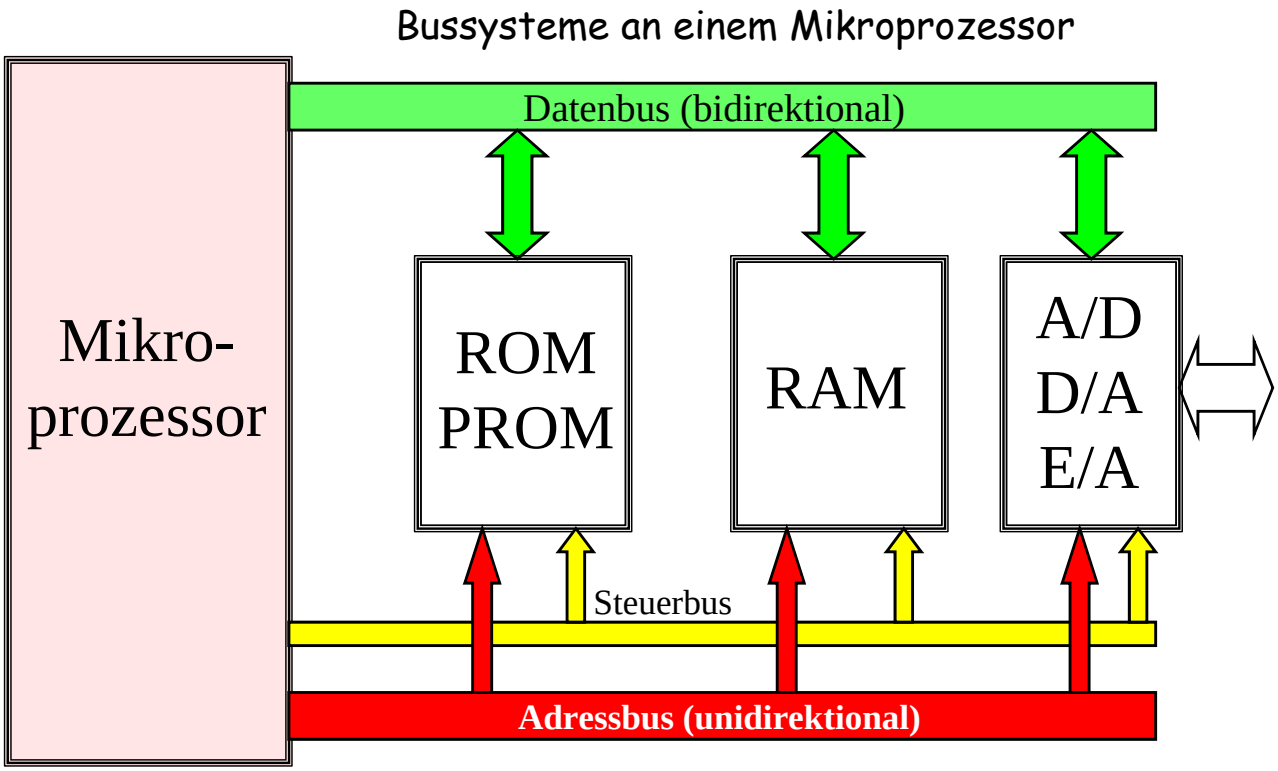



Bild 2.3: Aufgaben der verschiedenen Bussysteme beim Betrieb eines Mikroprozessorsystems

Aus den bislang dargestellten Zusammenhängen ist bereits erkennbar, daß ein Rechner in Form eines PC oder Notebooks nicht unmittelbar mit den Mikrorechnern, die im Rahmen dieser Vorlesung behandelt werden vergleichbar ist. Trotz der markanten Unterschiede die schon angesprochen wurden, gibt es eine wichtige Gemeinsamkeit auf der Ebene der CPU. Was eine CPU genau ist und über welche Bestandteile sie in der Regel verfügt, werden wir noch schrittweise herausarbeiten. An dieser Stelle genügt noch eine recht pauschale Funktionsbeschreibung folgender Art:

Eine CPU verfügt mindestens über ein Steuerwerk und eine arithmetisch-logische Einheit (ALU), die im einfachsten Fall nur Additionen und einfache boolesche Verknüpfungen wie UND, ODER, EXOR ausführen kann. Das Steuerwerk wird mit dem Einschalten der Stromversorgung in einen definierten Anfangszustand versetzt, d.h. eine Reset-Prozedur läuft ab. Dann beginnt ein Programm-Adreßzähler von 0 aus Adressen über den Adreßbus auszugeben. Gleichzeitig wird vom Steuerbus ein Aktivierungssignal für das Lesen aus einem Programmspeicher bereitgestellt. Die genauen zeitlichen Abläufe sind im Augenblick nicht von Bedeutung – sie werden später detailliert behandelt. Wenn ein Programmspeicher vorhanden, passend gefüllt und korrekt angeschlossen ist – vgl. Bild 2.3 – dann wird unter jeder Adresse, die der Adreßbus ausgibt, ein gültiger Operationscode gefunden und über den Datenbus in die CPU transferiert. Dort wird er mit Hilfe des Steuerwerks dekodiert, so daß schließlich die beabsichtigte Tätigkeit (Operation) ausführt werden kann. Das könnte ein Datentransfer, eine Addition oder eine boolesche Verknüpfung zweier Operanden sein. Auf welche Weise neben dem OP-Code die jeweils benötigten Operanden passend zur Verfügung gestellt werden, wollen wir im Moment noch außer Acht lassen. Die Anzahl der Leitungen, die ein Bus hat wird auch als Bitbreite des Busses bezeichnet; sie kann für Adreß- Daten- und Steuerbus unterschiedlich sein. Im Allgemeinen charakterisiert die Breite des Datenbusses einen Mikrorechner, d.h. man spricht in diesem Zusammenhang von einer 4-, 8-, 16-, 32- oder 64 bit-Maschine. Die Breite des Datenbusses wird auch Wortlänge genannt. In der Regel bestimmt sie weitgehend den inneren Aufbau, d.h. die Bitbreite des Rechenwerks, von Registern, Akkumulatoren und weiteren internen Speicherstellen sowie auch von integrierten Peripheriekomponenten.

Die hier noch sehr knapp und oberflächlich beschriebene Struktur und die Abläufe beim Holen und Abarbeiten von Operationscode sind allen Mikrorechnern gemeinsam. Auch in einem PC oder einem Notebook spielen sich auf der elektrischen Ebene des darin arbeitenden Mikroprozessors diese Vorgänge ab.

Das Erstellen von Programmen und das Einbringen in die Hardware, wo sie ablaufen sollen, sieht hingegen recht verschieden aus.

Ganz grob unterscheidet man zwischen Hochsprachen- und Assemblerprogrammierung. Als Hochsprachen seinen beispielhaft PASCAL, C und BASIC genannt. Hinter den Instruktionen, die in diesen Sprachen formuliert werden können, stecken häufig ziemlich komplexe Abläufe, die zur Abarbeitung auf einem Mikrorechner in zahlreiche „Unterschritte“ zerlegt werden müssen. Diese Unterschritte sind in der Regel „Befehle auf Assemblerebene“, die mit sogenannten Mnemonics formuliert werden. Das sind Abkürzungen aus der englischen Sprache, mit wenigen Buchstaben (meist 2…4), die so gewählt sind, daß sie die zugehörige Funktion andeuten, wie z.B.:

ADD für Addieren

CLR für Löschen, d.h. Nullen einschreiben

DEC für Dekrementieren (um eins vermindern)

INC für Inkrementieren (um eins erhöhen)

MOV oder MOVE für Datentransfer

MUL für Multiplizieren

SUB für Subtrahieren

Auch ein mittels solcher Mnemonics, beispielsweise mit eine Texteditor geschriebenes Programm, kann kein Mikrorechner abarbeiten. Es ist ein weiterer Umwandlungsschritt nötig. Denn das, was der Mikrorechner letztlich zur Ausführung der vom Programm beabsichtigten Funktion benötigt, ist stets ein binärer Operationscode in der zur jeweiligen Maschine passenden Bitbreite, der im einfachsten Fall in einem Programmspeicher unter fortlaufenden Adressen steht. In der Regel beginnt ein Mikrorechner nach Reset mit dem Einlesen des Inhaltes der Speicherstelle 0 und fährt nach Ausführung der zugehörigen Operation mit wachsenden Adressen fort. Falls keine Verzweigungen, Unterprogrammaufrufe oder Unterbrechungsanforderungen „dazwischenkommen“ wächst also der Inhalt des Programmadreßzählers linear, im einfachsten Fall von OP-Code zu OP-Code um eins. Das kann natürlich nicht beliebig lange fortgesetzt werden, da alle Speicher eine endliche Größe haben. Jedes Programm – egal auf welcher Ebene es erstellt wurde – muß demnach stets mindestens eine Schleife haben, in der es entweder endet oder auf ein Ereignis wartet. Manche Mikrorechner verfügen dazu explizit über einen Wartebefehl (WAIT), andere lösen die Aufgabe mit Hilfe einer sogenannten Endlosschleife. Beispiele dazu folgen in den Übungen.

Hochsprachen- und Assemblerprogrammierung unterscheiden sich somit wesentlich in den Schritten, die von der Programmerstellung bis hin zum Einschreiben des OP-Codes in den Programmspeicher des zu verwendenden Mikrorechners durchlaufen werden müssen. Die Assemblerebene ist also auch für eine Hochsprachenprogrammierung immer ein Zwischenschritt, der meistens jedoch für den Programmierer unsichtbar abläuft. Das was schließlich in den Programmspeicher kommt, sieht prinzipiell gleich aus, sowohl beim PC als auch beim “kleinen“ Mikrorechnersystem, das beispielsweise eine Waschmaschine oder eine Heizung steuert. Ein *.EXE-File für den PC enthält somit genauso in binärer Form den OP-Code, den der Zielprozessor Schritt für Schritt aus dem Programmspeicher lesen und ausführen muß, wie *.HEX- oder *-BIN-Files, die wir im Rahmen dieser Vorlesung für die behandelten Mikrorechner kennenlernen werden. Der wesentliche strukturelle Unterschied ist letztlich nur die Wortlänge, z.B. 32bit für den PC, 8bit für einen einfachen Mikrocontroller oder 24bit für einen DSP.

Abschließend sollen einige einfache Beispiele auf der Assemblerebene des 8bit-Mikrocontrollers vom Typ 8051 das oben Gesagte illustrieren. An dieser Stelle ist es noch nicht erforderlich zu wissen, was genau ein Akkumulator oder ein Register bzw. der Unterschied zwischen ihnen ist.

| Mnemonic | Operand | Kommentar |
|---|---|---|
| **CLR** | **A** | lösche den Akkumulator, d.h. alle Bits des Akku werden auf Null gesetzt |
| Binärer OP-Code | | |
| 1 1 1 0 0 1 0 0 | | |

| Mnemonic | Operand | Kommentar |
|---|---|---|
| **DEC** | **A** | vermindere den Inhalt des Akkumulators um eins |
| Binärer OP-Code | | |
| 0 0 0 1 0 1 0 0 | | |

| Mnemonic | Operand | Kommentar |
|---|---|---|
| **INC** | **Rr** | erhöhe den Inhalt des Registers mit der Nr. ’r’ um eins |
| Binärer OP-Code | | |
| 0 0 0 0 1 *r r r* | | *r r r* kann dabei Werte von 000 bis 111 annehmen (R0…R7) |

| Mnemonic | Operanden | Kommentar |
|---|---|---|
| **MOV** | **A, Rr** | bringe den Inhalt des Registers mit der Nr. ’r’ in den Akkumulator |
| Binärer OP-Code | | |
| 1 1 1 0 1 *r r r* | | *r r r* kann dabei Werte von 000 bis 111 annehmen (R0…R7) |

| Mnemonic | Operand | Kommentar |
|---|---|---|
| **MUL** | **AB** | multipliziere den Inhalt des Akkumulators mit dem von Register ’B’ |
| Binärer OP-Code | | |
| 1 0 1 0 0 1 0 0 | | |

Eine vollständige Beschreibung mit Beispielen zu allen Befehlen der 8051-Mikrocontrollerfamilie ist in dem 79-seitigen Dokument BEFSS_8051.PDF zu finden.

Nach diesem allgemeinen Vorspann folgt nun ein Einblick in die Technologie heutiger Mikrorechner, der den Zusammenhang zwischen der elektrischen Arbeitsweise von Bauteilen und den zu realisierenden logischen Funktionen herstellen soll. Darauf aufbauend werden schrittweise in allgemeiner Form Steuerwerk, Rechenwerk und Speicher detailliert behandelt. Dann folgt die Analyse “realer“ Mikrorechner, die zum heutigen Industriestandard zählen, angefangen vom einfachen 8bit-Controller bis hin zum komplexen digitalen Signalprozessor.

## 2.2 CMOS-Technologie für die Mikrorechnerherstellung

An dieser Stelle wird eine kurze, einführende Darstellung in die CMOS-Technologie gegeben, die von der Halbleiterseite her die wichtigste Grundlage zur Herstellung praktisch aller heutigen Mikrorechner bildet. Der Einblick in die Technologie ist für das Verständnis des elektrischen Verhaltens von Mikrorechnerschaltungen unerläßlich. Bei der Darstellung wird auf komplexe Zusammenhänge der Festkörperphysik verzichtet. Statt dessen wird das von Außen beobachtbare „makroskopische“ elektrische Verhalten in Grundzügen untersucht, wobei auch der Bezug zur Geometrie auf dem Silizium hergestellt wird.

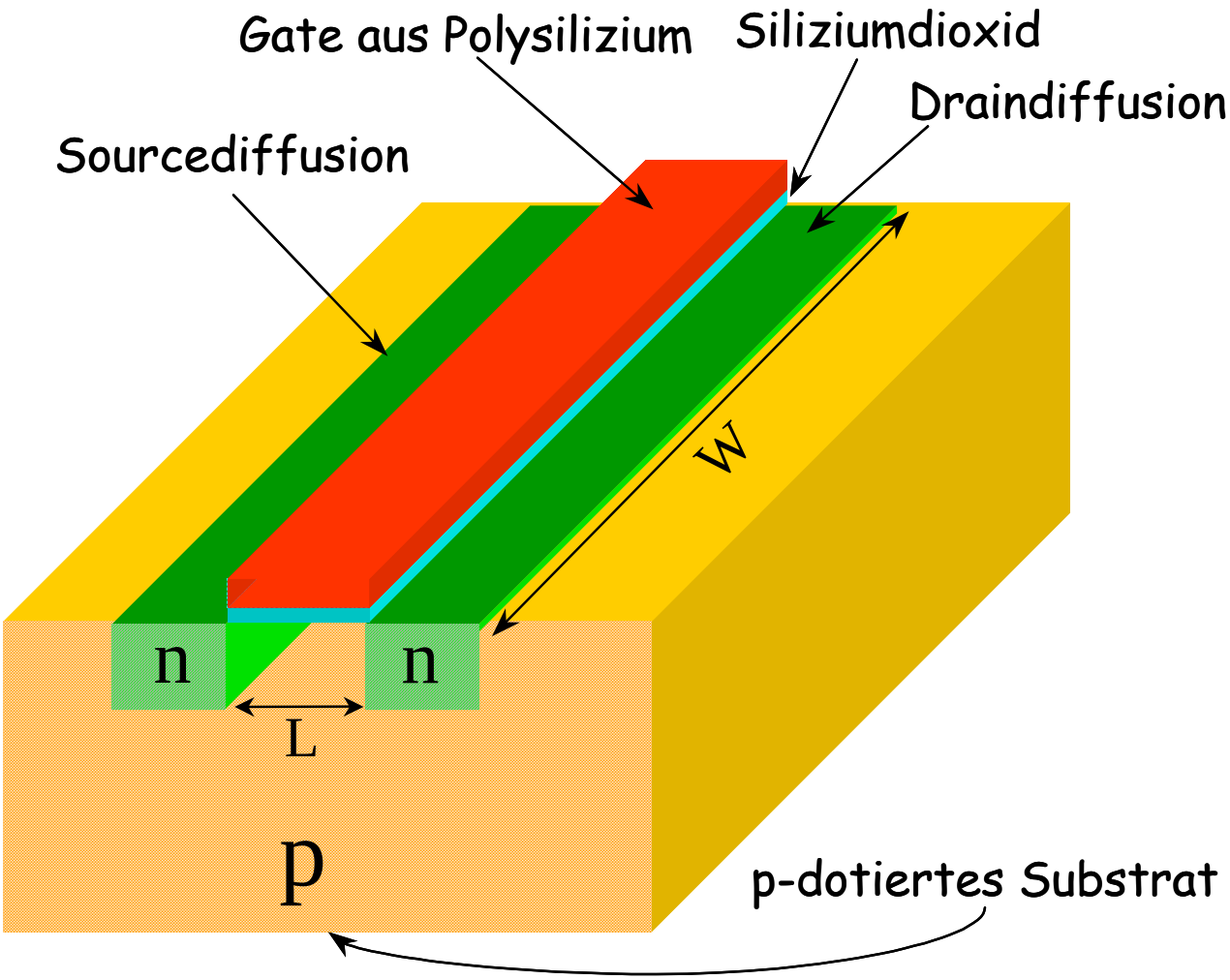



Bild 2.4: Aufbau eines n-Kanal MOS–Transistors auf p-Substrat

Die Abkürzung ’CMOS’ bedeutet ’komplementäre Metall-Oxid-Silizium’ Technologie. Komplementär heißt in diesem Zusammenhang, daß es zwei Arten von Transistoren gibt, die sich in ihrem elektrischen Verhalten sozusagen ergänzen – dies sind n- und p-Kanal MOSFETs. Zunächst sei anhand von Bild 2.4 der prinzipielle Aufbau und die Arbeitsweise eines n-Kanal MOSFET erläutert. Ausgangspunkt bei der Herstellung ist p-dotiertes Siliziumsubstrat, d.h. hier sind Fremdatome eines dreiwertigen Elements – z.B. Bor – eindiffundiert, so daß ein Elektronenmangel vorliegt. Man spricht auch von Löcherleitung[4]. Elektronenmangel bedeutet nicht, daß keine Elektronen vorhanden sind, sondern daß ihre Dichte (die auf ein Volumen bezogene Anzahl) sehr viel kleiner ist als die der Löcher. Man sagt deshalb auch, daß Elektronen in p-Material Minoritätsträger sind, Löcher

[4] in der Halbleitertechnik ist es üblich, Elektronenfehlstellen als Löcher zu bezeichnen

hingegen Majoritätsträger. Das Umgekehrte gilt für n-dotiertes Silizium, das durch Einbau von Fremdatomen eines fünfwertigen Elements – wie z.B. Arsen – hergestellt werden kann. Die Siliziumsubstrate zur Herstellung integrierter Schaltungen sind in der Regel relativ schwach p- oder n-dotiert. Deshalb ist es möglich, diese schwachen Dotierungen mit entgegengesetzten starken zu 'überschreiben'. Tut man dieses – wie in Bild 2.4 dargestellt – indem man zwei stark n-dotierte Bereiche einbringt, dann hat man bereits den Grundaufbau eines n-Kanal Transistors vor sich. Diese Bereiche werden Source und Drain genannt und über metallische Verbindungen – wie in Bild 2.5 (oben) dargestellt - nach Außen elektrisch zugänglich gemacht. Es fehlt jetzt noch der dritte Anschluß für einen vollständigen Transistor, nämlich die Steuerelektrode, die Gate genannt wird. Mit Hilfe dieser Elektrode kann der Stromfluß zwischen Source und Drain beeinflußt werden. Da im Rahmen dieser Vorlesung digitale Schaltungen im Vordergrund stehen, ist der exakte analoge Zusammenhang hier nicht von Interesse, sondern die Zustände 'kein Stromfluß', d.h. der Transistor ist ausgeschaltet und 'maximaler Stromfluß', wobei der Transistor eingeschaltet ist.

Die Gateelektrode besteht aus Material mit metallähnlichen Eigenschaften (Polysilizium) und sie ist durch eine sehr dünne isolierende Schicht aus Siliziumdioxid von der Halbleiteroberfläche getrennt. Wenn sich das Gate bei dem n-Kanaltransistor nach Bild 2.5 (oben) auf gleichem elektrischem Potential wie das Substrat befindet, wird bei Anlegen einer positiven Spannung zwischen Drain und Source kein Stromfluß zustande kommen. Hierzu würden Elektronen als Ladungsträger in dem als 'Kanal' bezeichneten Bereich zwischen den n-dotierten Bereichen benötigt. Ohne diese Ladungsträger sind np- bzw. pn-Sperrschichten vorhanden, die einen Stromfluß unterbinden. Wird jedoch das Gate hinreichend stark positiv gegenüber dem Substrat vorgespannt, dann werden dadurch negative Ladungsträger (Elektronen) angezogen, d.h. es findet eine Konzentration von Minoritätsträgern direkt unter dem Gate statt und die Elektronendichte im Kanal wächst erheblich, so daß ein Stromfluß zwischen Drain und Source einsetzen kann. Da das Gate isoliert ist, fließt kein Strom in diesen Anschluß des Transistors. Er verhält sich daher wie eine spannungsgesteuerte Stromquelle. Wir kommen anhand von Bild 2.6 noch mal auf diesen Sachverhalt zurück.

Das Verhalten des zum n-Typ komplementären p-Kanaltransistors läßt sich aus dem oben Gesagten unmittelbar durch Umkehren der Dotierungen und der Vorzeichen der anzulegenden Spannungen ableiten - s. Bild 2.5 unten. Eine negative Spannung zwischen Gate und n-dotiertem Substrat bewirkt hier eine Konzentration von Löchern im Kanal, die jetzt bei negativer Spannung zwischen Drain und Source einen entsprechenden Stromfluß ermöglichen.

Wie Bild 2.4 und Bild 2.5 nahelegen, sind MOS-Transistoren symmetrisch aufgebaut, d.h. Drain- und Sourceanschlüsse können vertauscht werden. Diese Möglichkeit kann beim Verdrahten komplexer integrierter Schaltungen vorteilhaft genutzt werden – s. auch Kapitel 5. Dort wird auch gezeigt, wie n- und p-Kanaltransistoren zusammen auf einem Substrat integriert werden können - nur dadurch entsteht schließlich eine CMOS-Schaltung. Hierzu sind zusätzliche Halbleiterdotierungsschritte nötig, da nach Bild 2.5 unterschiedlich vordotierte Substrate für die beiden Typen benötigt werden.

Bevor erste einfache Anwendungen der CMOS-Technologie zum Aufbau von Digitalschaltungen behandelt werden, soll das oben Gesagte noch anhand von drei leicht zu handhabenden algebraischen Gleichungen etwas vertieft werden. Eine detaillierte Beschreibung und die wichtigsten physikalischen Grundlagen des elektrischen Verhaltens von MOS-Transistoren sind im „weiterführenden Ergänzungsskriptum“ zu finden.

## 2.2.1 CMOS-Strukturen auf Silizium

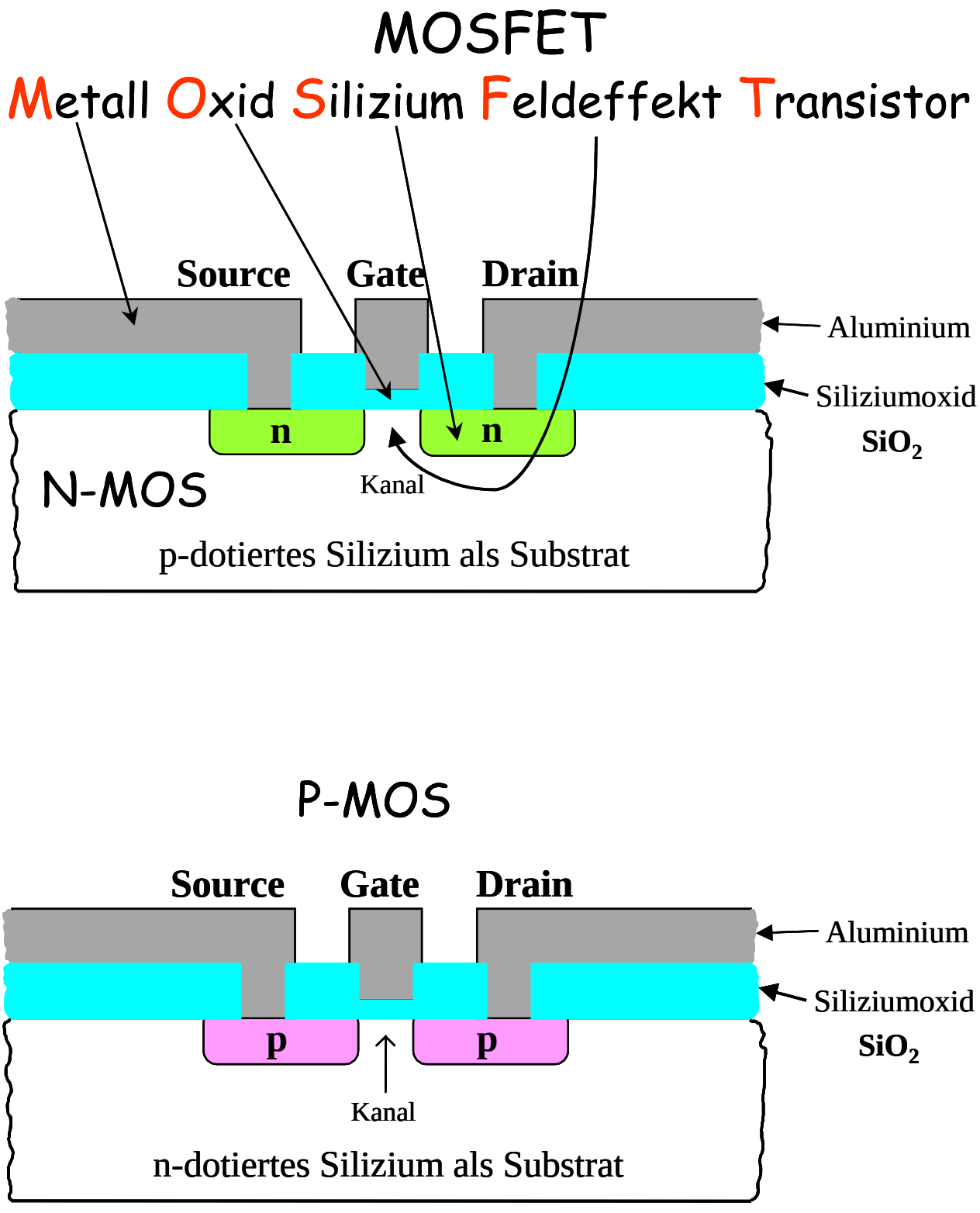



Bild 2.5: p– und n–MOS–Transistoraufbau auf Silizium

## Vereinfachte Beschreibungsgleichungen eines n-Kanal MOS-Transistors

Aus den obigen Betrachtungen ist bereits hervorgegangen, daß MOS-Transistoren in Digitalschaltungen grundsätzlich die beiden Zustände 'ausgeschaltet' oder 'eingeschaltet' einnehmen können. Wir können deshalb vereinfacht zunächst von einem Sperrbereich und einem „Leitbereich" sprechen. Die mathematische Definition des Sperrbereiches ist sehr einfach:

### Sperrbereich:

$$I_{DS} = 0 \quad \text{solange} \quad U_{GS} - U_{tn} \leq 0 \tag{2.1}$$

Der Drainstrom $I_{DS}$ (bei positiver Spannung $U_{DS}$ von Drain zu Source fließend) ist Null, solange die Gate-Source-Spannung $U_{GS}$ unterhalb einer Schwelle $U_{tn}$ bleibt.

Etwas komplizierter verhält sich der eingeschaltete Transistor. Wir benötigen die Kenntnis dieses Verhaltens, um z.B. das elektrische Verhalten von Portpins eines Mikrorechners oder auch von statischen Speicherzellen zu verstehen. Hierbei wird der Verlauf von Strom und Spannung beim Übergang der beteiligten Transistoren vom Sperr- in den Leitbereich benötigt.

Im leitenden Bereich unterscheiden wir den linearen und den Sättigungsbereich. Bei niedriger Drain-Source-Spannung steigt der Drainstrom $I_{DS}$ linear mit wachsender Gate-Source-Spannung $U_{GS}$ gemäß (2.2) an. Die Spannung $U_{GS}$ muß dabei stets größer als die Schwellspannung $U_{tn}$ sein und die Drain-Source-Spannung $U_{DS}$ muß unterhalb von $U_{GS}$-$U_{tn}$ bleiben.

### Linearer Bereich:

$$I_{DS} = \beta\left[\left(U_{GS} - U_{tn}\right)U_{DS} - \frac{U_{DS}^{\,2}}{2}\right] \quad \text{für} \quad 0 < U_{DS} \leq U_{GS} - U_{tn} \tag{2.2}$$

Der lineare Bereich wird somit bei jedem Einschaltvorgang – meist sehr schnell - durchlaufen. Der 'stationäre' Einschaltzustand befindet sich normalerweise im Sättigungsbereich. Dieser wird mit hoher Drain-Source-Spannung erreicht, wobei in der Regel auch die Gate-Source-Spannung hoch ist. In jedem Fall muß jedoch $U_{DS}$ größer als $U_{GS}$-$U_{tn}$ sein. Interessant ist, daß jetzt der Drainstrom gemäß (2.3) nicht mehr von der Drain-Source-Spannung abhängt, sondern quadratisch von $U_{GS}$. Man hat somit eine ideale Stromquelle vor sich, bei der Stromanstieg durch eine Konstante β mit der Dimension $A/V^2$ bestimmt wird, die unter anderem von der Transistorgeometrie abhängt.

### Sättigungsbereich:

$$I_{DS} = \frac{\beta}{2}\left(U_{GS} - U_{tn}\right)^2 \quad \text{für} \quad U_{DS} \geq U_{GS} - U_{tn} \tag{2.3}$$

Aus den obigen Betrachtungen wird deutlich, daß ein elektronischer Schalter in Form eines MOS-Transistors in seinem Verhalten erheblich von dem eines idealen Schalters abweicht. Während der Ausschaltzustand noch relativ gut vergleichbar ist, muß für den Einschaltzustand die Strombegrenzung nach (2.3) und oft auch das 'dynamische' Verhalten beim Übergang vom ausgeschalteten in den eingeschalteten Zustand berücksichtigt werden, um 'reale' Digitalschaltungen zu verstehen und korrekt auszulegen. Wir werden jedoch im weiteren immer dort, wo es nicht zu Fehlinterpretationen führt, die realen Strom-Spannungszusammenhänge soweit wie möglich vereinfachen, d.h. oft lineare Zusammenhänge annehmen.

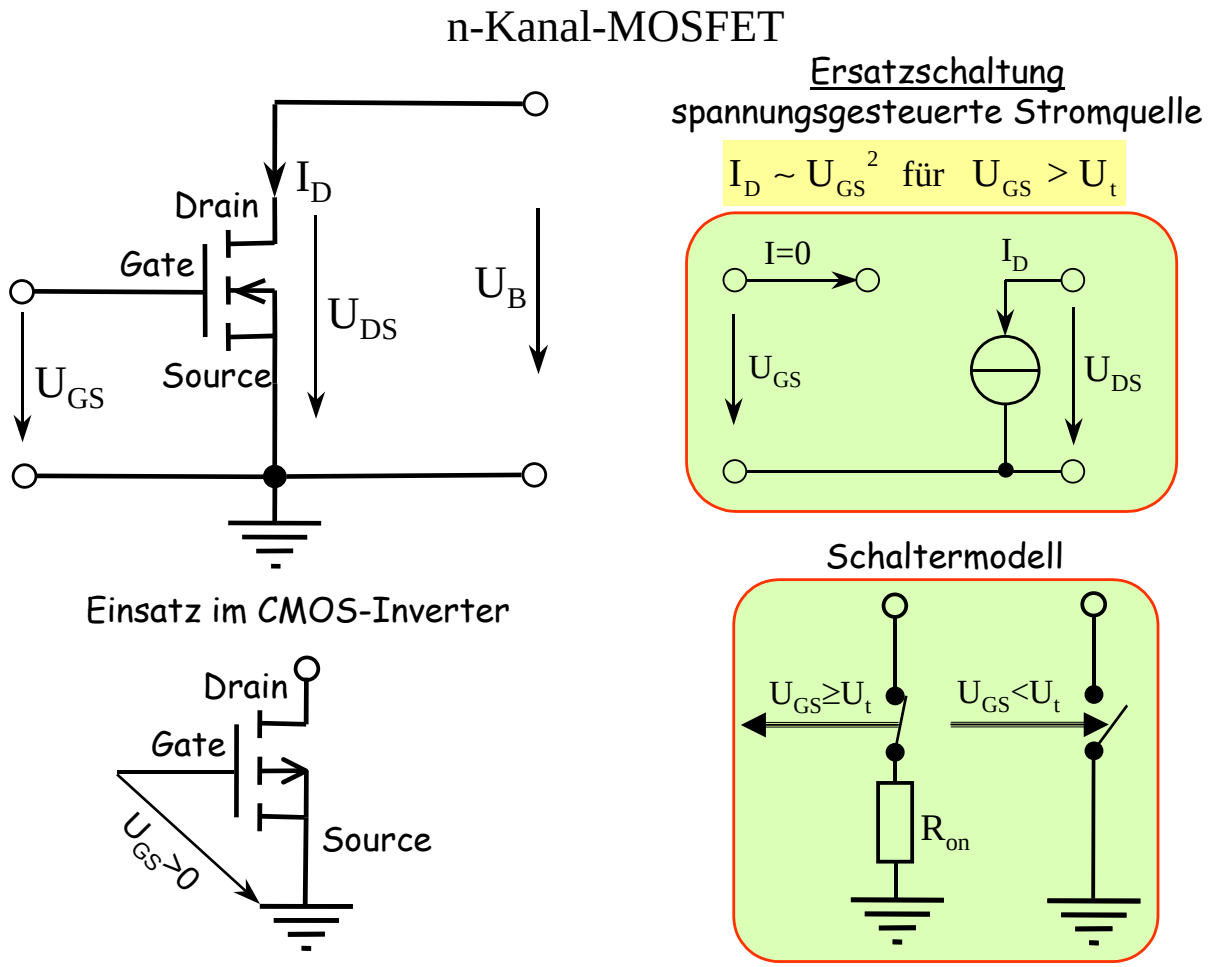



Bild 2.6: Grundschaltung eines n-Kanal MOS–Transistors für Schalterbetrieb

Anhand von Bild 2.6 werden die Zusammenhänge für den Schaltbetrieb eines n-Kanal MOS-Transistors zusammengefaßt. Solange $U_{GS}$ unterhalb einer positiven Schwellspannung $U_t$ liegt, hat man den Schaltzustand 'offen', d.h. es fließt kein Strom ($I_D$=0). In Schaltungen, die für eine Betriebsspannung von $U_B$=5V ausgelegt sind, ist $U_t \approx$1V ein typischer Wert für die Schwellspannung.

Sobald $U_{GS}$ die Schwelle $U_t$ überschreitet, beginnt der Drainstrom gemäß (2.3) zu steigen und erreicht schließlich bei $U_{GS}=U_B$ seinen Maximalwert. Hat die Konstante β nun z.B. den Wert 1mA/V$^2$, dann erhält man für den Maximalwert des Stroms im Einschaltzustand $I_{Dmax}$=0,5·16mA =8mA. Bei einem idealen Schalter würde dagegen der Strom schon bei beliebig kleiner Spannung gegen Unendlich gehen. Durch Einfügen eines Widerstands $R_{on}$ in Bild 2.6 kann die Strombegrenzung näherungsweise nachgebildet werden. Die ideale Stromquelleneigenschaft des Transistors wird dabei nicht berücksichtigt – was in vielen Fällen durchaus zulässig ist, ohne grobe Fehler zu machen.

Bild 2.7 zeigt die entsprechenden Verhältnisse für einen p-Kanal MOSFET, wobei die Umkehrung der Spannungs- und Stromrichtungen zu beachten ist.

Zum Aufbau eines CMOS-Inverters, der eine der wichtigsten digitalen Grundschaltungen darstellt, wird wie in Bild 2.6 und Bild 2.7 bereits angedeutet ist, je ein n- und ein p-Kanal Transistor benötigt, wobei der Sourceanschluß des n-Kanal FETs mit Masse und der des p-Kanal FETs mit $+U_B$ zu verbinden ist. Die beiden Drains bilden, miteinander verbunden, den Ausgang, während die Gates – ebenfalls miteinander verbunden - den Eingang darstellen.

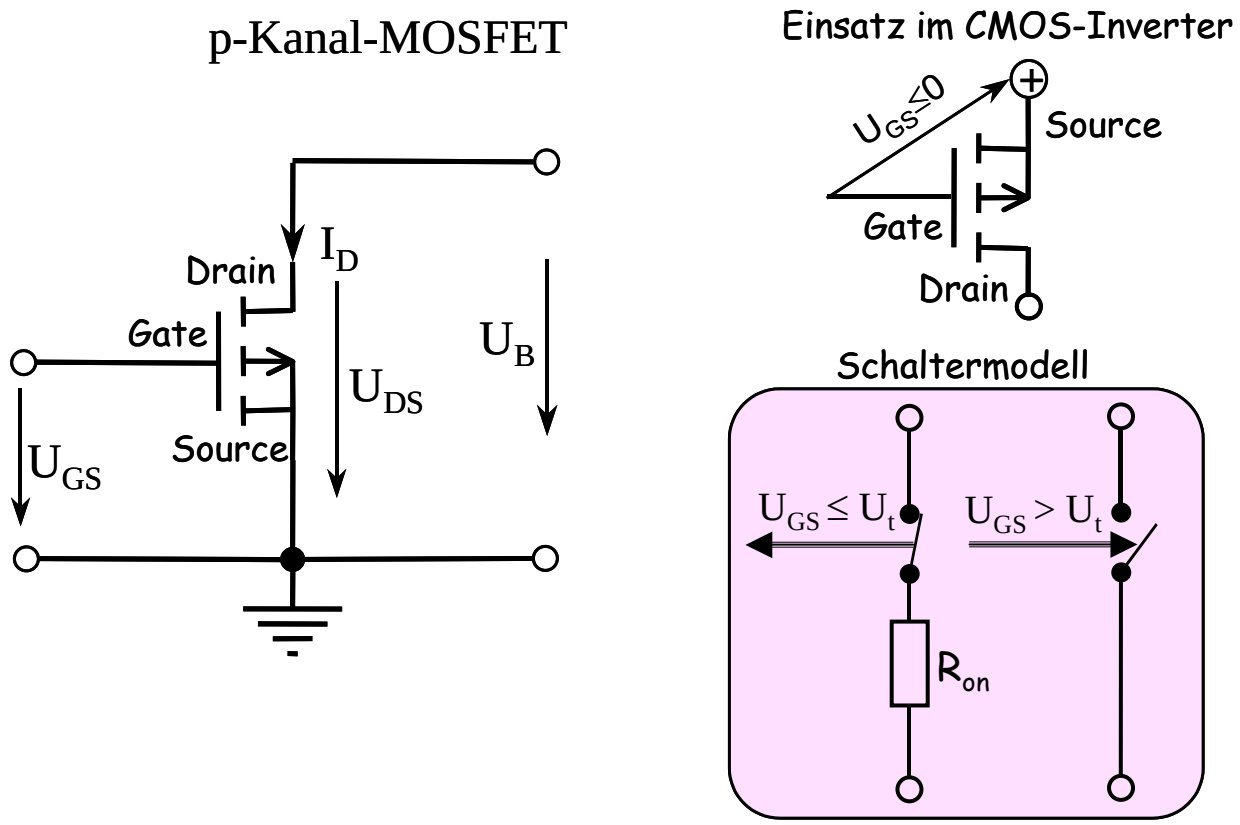



Bild 2.7: Grundschaltung eines p-Kanal MOS–Transistors für Schalterbetrieb

## 2.3 CMOS-Grundschaltungen

### 2.3.1 Inverter

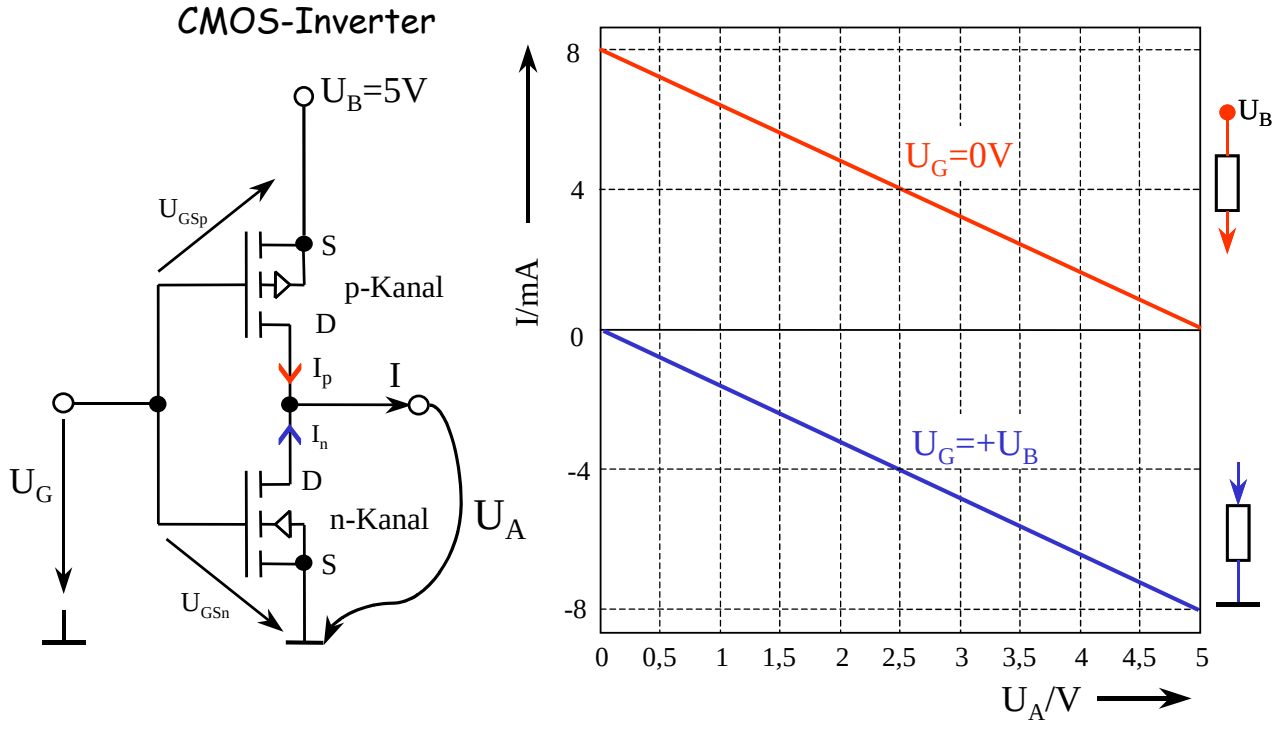



Bild 2.8: CMOS–Inverteraufbau und das zugehörige elektrische Verhalten (vereinfacht)

Der schon beschriebene Aufbau eines CMOS-Inverters ist in Bild 2.8 zusammen mit dem vereinfachten elektrischen Verhalten dargestellt. Die Vereinfachungen bestehen zum einen darin, daß der 'krummlinige' Zusammenhang von Strom und Spannung linear dargestellt ist und zum anderen darin ,daß das 'dynamische' Verhalten hier noch nicht betrachtet wird, d.h. der Übergang vom Sperr- in den Leitzustand erfolgt für beide Transistoren ohne Verzug, ebenso wie der Wechsel der Eingangsspannung $U_G$ von 0 auf $+U_B$ oder umgekehrt. Zudem ist angenommen, daß sich die beiden Transittoren exakt komplementär zueinander verhalten. In Wirklichkeit kann ein p-Kanaltransistor gleicher Geometrie nur rund ein Fünftel des Stromes liefern wie sein n-Kanalkomplement. Zu gegebener Zeit muß daher diese Vereinfachung zurückgenommen werden.

Zusammengefaßt führt die Vereinfachung nach Bild 2.8 zu einem Model, das die Transistoren als widerstandsbehaftete Schalter, jeweils mit $R_{on}$=625Ω, darstellt.

Für $U_G$=0 ist der p-Kanaltransistor eingeschaltet und der n-Kanaltransistor ist aus. Die Ausgangsspannung nimmt den Wert $U_A=+U_B$=5V an und es fließt nirgends Strom. Würde man durch Belastung, z.B. mit einem Widerstand nach Masse, die Ausgangsspannung $U_A$ absenken, dann ergäbe sich ein Ausgangsstrom $I$ gemäß der oberen (roten) Kennlinie ($U_G$=0V), die bei einem Kurzschluß ($U_A$=0) bei dem Maximalwert $I=I_P$=8mA enden würde.

Entsprechend ist für $U_G$=+5V ist der n-Kanaltransistor eingeschaltet und der p-Kanaltransistor ist aus. Die Ausgangsspannung nimmt den Wert $U_A$=0V an und es fließt wiederum nirgends Strom. Würde man nun mit einem Widerstand nach $+U_B$ versuchen, die Ausgangsspannung $U_A$ anzuheben, dann ergäbe sich ein Ausgangsstrom $I$ gemäß der unteren (blauen) Kennlinie ($U_G=+U_B$), die bei $U_A$=+5V bei dem Maximalwert $I=I_n$=-8mA enden würde.

Die Inverterfunktion ist also für einen 'unbelasteten' Ausgang ideal realisiert. Je nach Belastung können aber die idealen Pegel erheblich beeinflußt werden, so daß unter Umständen nachfolgende Schaltungen nicht mehr korrekt arbeiten. In der Praxis muß dabei auch das dynamische Verhalten berücksichtigt werden, d.h. Pegelübergänge mit beliebigem Zeitverhalten. Diese Grundbetrachtungen zeigen bereits, daß spätesten hier das Modell des idealen Schalters für den Entwurf digitaler Schaltungen nicht mehr brauchbar ist.

Neben Invertern spielen Transfergates in der CMOS-Technologie eine wichtige Rolle. Man kann damit digitale Schaltungen erheblich einfacher realisieren als in 'gewöhnlicher' Gattertechnik. Der Aufbau eines Transfergates ist sehr einfach: Es besteht aus der Parallelschaltung einen p- und eines n-Kanaltransistors, wie Bild 2.9 zeigt. Der Name Transfergate bedeutet in etwa Übertragungsgatter, d.h. ein solches Bauteil soll nach Maßgabe eines Steuersignals Signalpegel weiterleiten. Wie wir noch sehen werden, könnte einzelner Transistor nicht beide Pegel "H" und "L" aktiv weiterleiten – deshalb ist die Parallelschaltung nötig. Da die beiden Transistortypen zueinander komplementär sind, benötigt ein Transfergate ein ebenfalls komplementäres Einschaltsignal ($T$ und $\overline{T}$).

Zum Verständnis der Funktion gehen wir ähnlich vor wie oben beim Inverter. Auch hier ist wieder der vereinfachte lineare Zusammenhang zwischen Strom und Spannung an den Transistoren zugrunde gelegt. Im Sperrzustand ($T$=0V und $\overline{T}$=5V) kann keiner der Transistoren leiten, ganz gleich, welcher Pegel $U_{IN}$ anliegt. Im Einschaltzustand, d.h. bei $T$=5V und $\overline{T}$=0V wird ein "H"-Pegel ($U_{IN}$=5V) über den p-Kanaltransistor zu $U_{out}$ weitergegeben, und zwar gemäß der oberen (roten) Kennlinie in Bild 2.9. Wenn der Ausgang unbelastet ist, d.h. kein Strom fließt, stellt sich $U_{out}$=5V ein. Würde man durch Belastung, z.B. mit einem Widerstand nach Masse, die Ausgangsspannung $U_{out}$ absenken, dann ergäbe sich ein Ausgangsstrom gemäß der oberen (roten) Kennlinie, die bei einem Kurzschluß ($U_{out}$=0), ähnlich wie beim Inverter, bei einem Maximalwert von z.B. einigen mA enden würde.

### 2.3.2 Transfergates

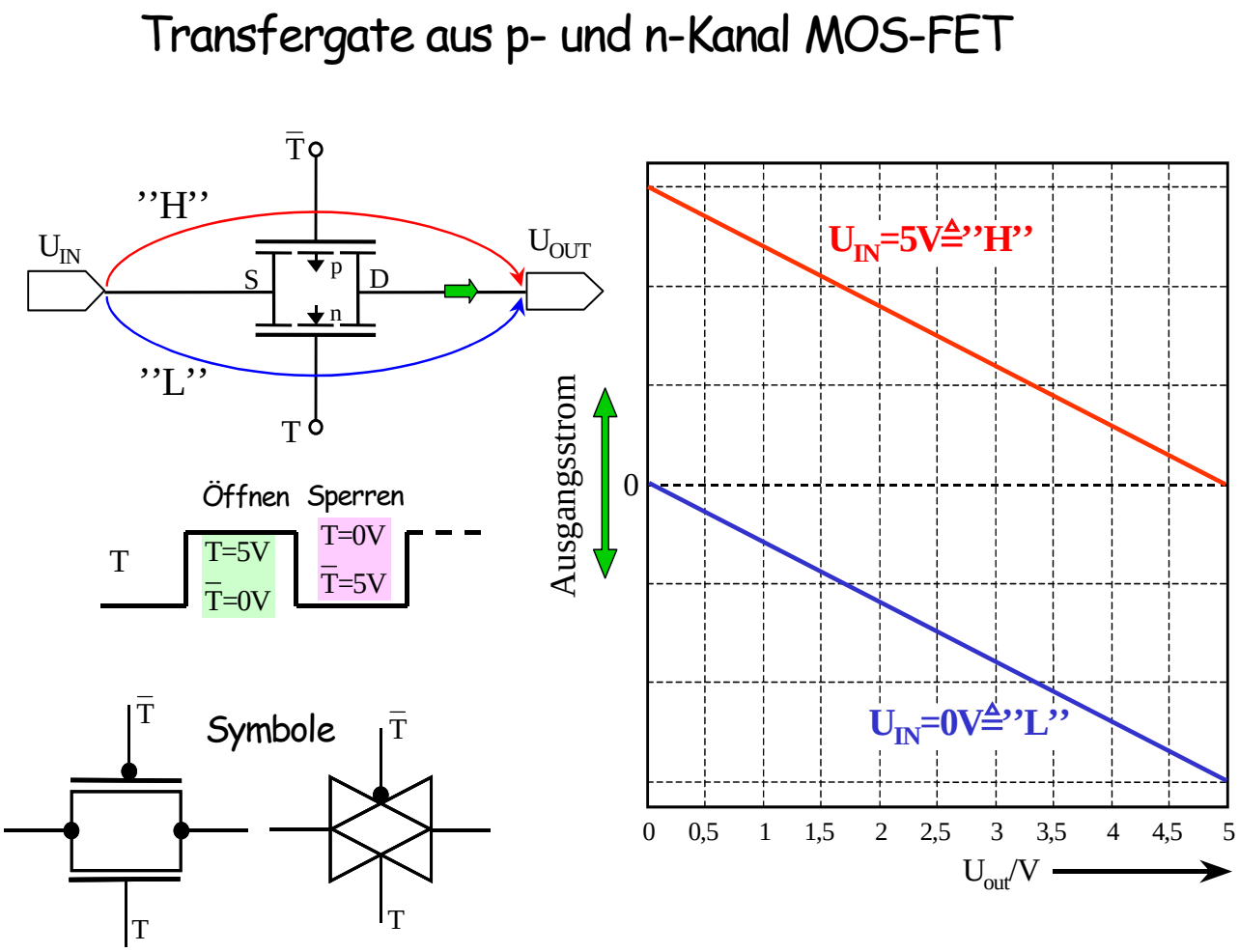



Bild 2.9: Grundlagen der CMOS–Transfergate–Technologie (vereinfacht)

Ein “L“-Pegel ($U_{IN}$=0V) wird dagegen im Einschaltzustand über den n-Kanaltransistor zu $U_{out}$ weitergegeben, und zwar gemäß der unteren (blauen) Kennlinie in Bild 2.9. Wenn der Ausgang unbelastet ist (d.h. kein Strom fließt), stellt sich jetzt $U_{out}$=0V ein. Würde man nun mit einem Widerstand nach $+U_B$ versuchen, die Ausgangsspannung $U_{out}$ anzuheben, dann ergäbe sich ein Ausgangsstrom gemäß der unteren (blauen) Kennlinie, der bei $U_{out}$=+5V wiederum - ähnlich wie beim Inverter - bei einem Maximalwert von einigen mA, aber mit negativem Vorzeichen enden würde.

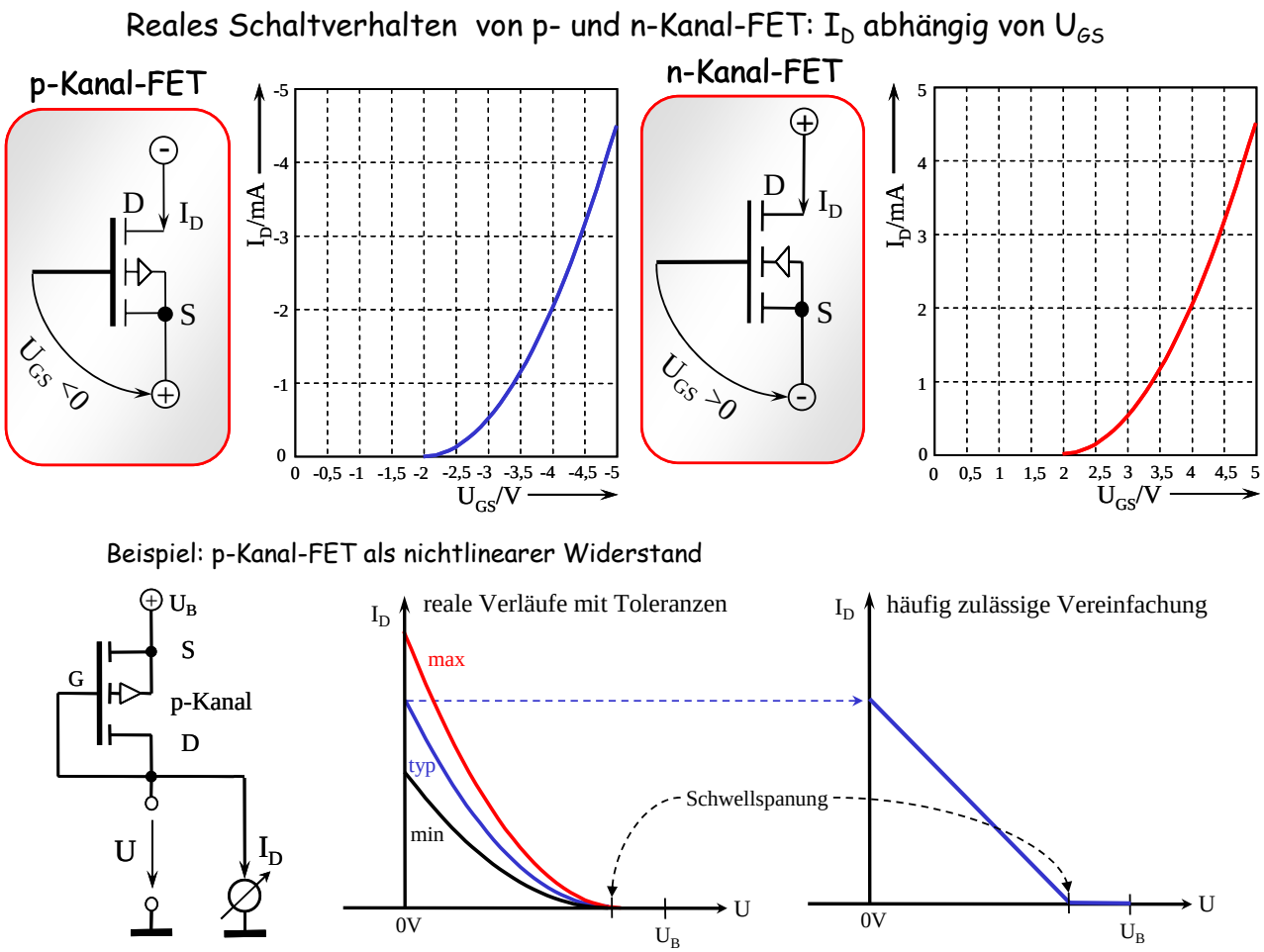



Bild 2.10: MOS–Transistorschaltungen an E/A-Pins von Mikrorechnern: p-Kanal-FET als „Pull-up“ und n-Kanal-FET als „Pull-down“

Bei den Ein- und Ausgabeports von Mikrorechnern ist die vereinfachende Annahme eines linearen Zusammenhangs von Strom und Spannung häufig, aber nicht immer zulässig. Deshalb sind in Bild 4.53 die realen Kennlinien für p- und n-Kanaltransistoren dargestellt. Man beachte, daß der Stromfluß erst ab einer bestimmten Schwellenspannung (hier -2V für p- und +2V für n-Kanal) einsetzt. Während der ’Anlaufbereich’ unmittelbar nach Überschreiten der Schwellspannung noch gekrümmt ist, ergibt sich für größere Ströme mit guter Näherung ein linearer Verlauf.

### 2.3.3 Leistungsaufnahme bei CMOS-Technologie

Beim Einsatz von CMOS-Technologie zur Realisierung digitaler Schaltungen wird im Ruhezustand keine elektrische Leistung aufgenommen. Das schon beim Inverteraufbau vorgestellte Schalterprinzip läßt sich auf Gatter aller Art und damit auch auf Speicherelemente übertragen. Es besagt, daß ein logischer „1"-Zustand immer nur durch Transistoren (in der Regel p-Kanal MOS-FETs), die eine Verbindung mit der positiven Betriebsspannung $U_B$ herstellen, erzeugt wird, wobei die mit „Masse" verbundenen Transistoren (in der Regel n-Kanal MOS-FETs) ausgeschaltet sind. Im logischen „0"-Zustand sind die beschriebenen Verhältnisse gerade umgekehrt. In keinem Fall fließt somit ein „statischer" Strom.

Sobald jedoch Pegeländerungen auftreten, beispielsweise bei Anlegen eines Taktsignals, nehmen CMOS-Schaltungen Wirkleistung auf. Es werden nicht nur Kapazitäten in Form von Busleitungen auf einer Platine oder Leiterbahnen auf dem Halbleiter, sowie Gate-Substrat-Kapazitäten umgeladen, sondern es fließt auch z.B. bei einem Inverter kurzzeitig bei jeder Taktflanke im Umschaltmoment durch beide Transistoren ein Strom. Eine detaillierte Analyse, auf die hier verzichtet wird, findet sich im *weiterführenden Ergänzungsskript*. Dort ist gezeigt, daß für die Gesamtleistung, die eine mit der Spannung $U$ und der Frequenz $f$ betriebene CMOS-Digitalschaltung aufnimmt

$$\boxed{P_{vG} \sim f \cdot U^2} \qquad (2.4)$$

gilt. Aus dieser einfachen Beziehung ergibt sich die wichtige Schlußfolgerung :

Eine Absenkung der Betriebsspannung von digitalen CMOS-Schaltungen bringt wegen der quadratischen Abhängigkeit der Leistungsaufnahme enorme Vorteile im Hinblick auf die Verluste, die in Form von Wärme abgeführt werden müssen:

| | | | |
|---|---|---|---|
| **Ausgangspunkt:** | 5V entspreche 100% Leistungsaufnahme, dann hat man bei | | |
| | 3,3V | 43,5% | der Verluste gegenüber 5V |
| | 2,5V | 25% | der Verluste gegenüber 5V |
| | 1,8V | 12,9% | der Verluste gegenüber 5V |

## 2.4 Digitale Grundschaltungen in CMOS–Technologie

### 2.4.1 CMOS–NAND, EXOR–Gatter

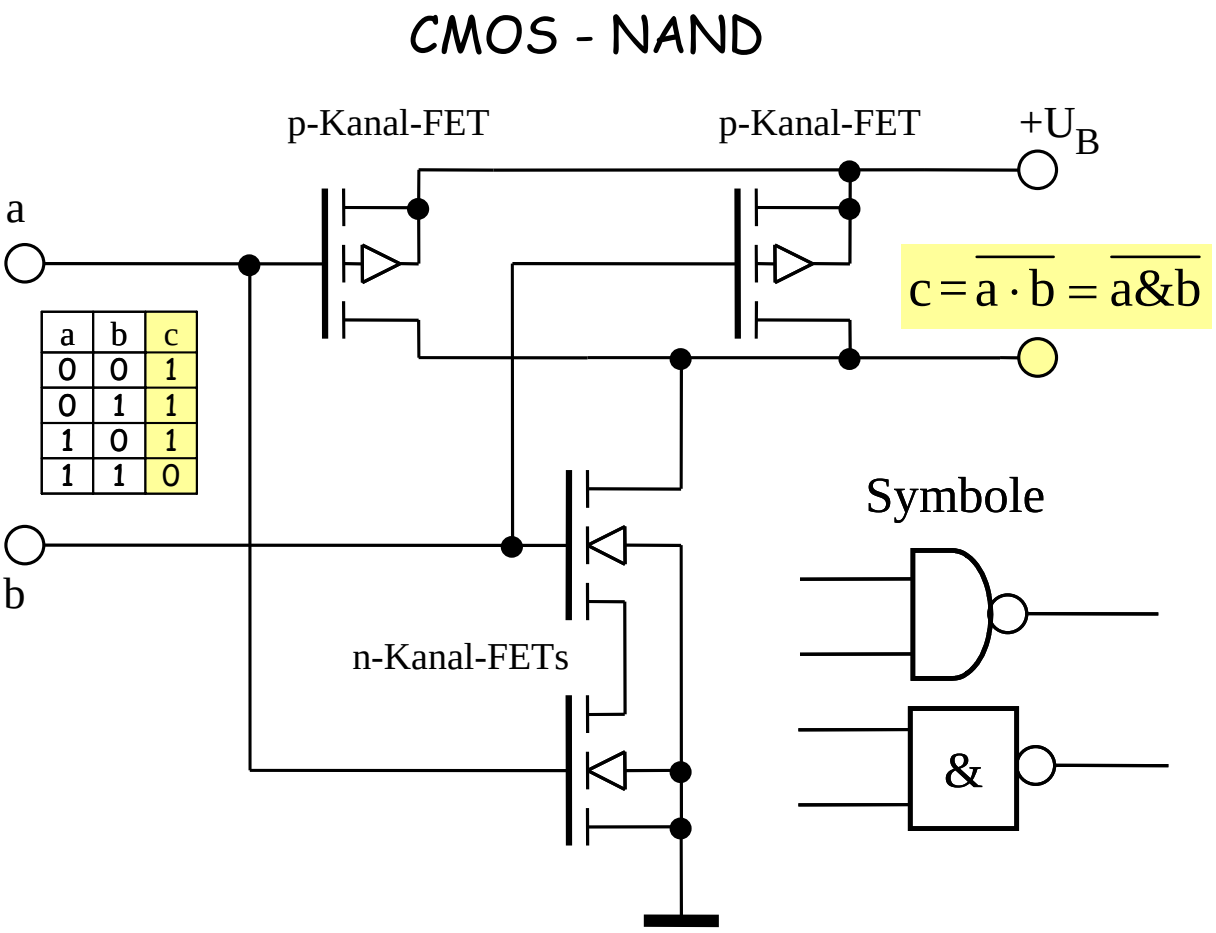



Bild 2.11: Aufbau eines NAND–Gatters in CMOS–Technologie

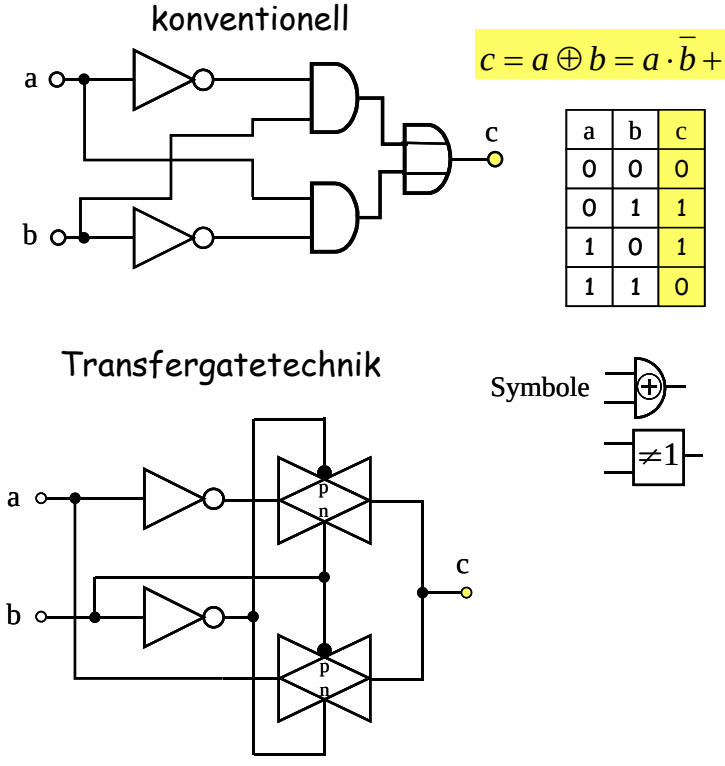



Bild 2.12: CMOS EXOR–Gatter

Die CMOS-Technologie ermöglicht die Realisierung logischer Funktionen ausschließlich mit Transistoren, d.h. es werden keine Widerstände, Dioden oder Kondensatoren benötigt. In vielen Fällen erlaubt die Transfergatetechnik besonders einfache Lösungen, die bis zur Hälfte der Transistoren im Vergleich zu konventionellen Aufbauten einsparen können.

Als Beispiele sind hier ein NAND- und ein EXOR-Gatter aufgeführt. Aus dem Schaltbild des NAND-Gatters ersieht man leicht die Funktion: Um eine logische ’0’ am Ausgang c zu erzeugen, müssen die beiden in Serie geschalteten n-Kanal FETs eingeschaltet sein, d.h. sie benötigen

beide einen ’1’-Pegel an ihrem Gate. In allen anderen Fällen ist mindesten ein ’0’-Pegel an einem der Eingänge a oder b vorhanden, so daß stets einer der p-Kanal-FETs eingeschaltet ist und einen ’1’–Pegel an den Ausgang c bringt. Das NAND-Gatter ist ein wichtiges Grundelement der digitalen Schaltungstechnik, mit dem prinzipiell durch einfache Modifikationen beliebig komplexe digitale Funktion aufgebaut werden können. Die Größe höchst komplexer hochintegrierter Digitalschaltungen wie z.B. „Field Programmable Gate Arrays (FPGA)“, die in letzten Kapitel betrachtet werden wird häufig mit der Zahl äquivalenter Gatter angegeben. Hiermit sind CMOS-NAND-Gatter gemeint, so daß sich die Zahl der Transistoren einer solchen Schaltung einfach durch Multiplikation mit 4 abschätzen läßt.
Ein EXOR–Gatter ist als zweites Beispiel aufgeführt, weil es beim Aufbau von Rechenschaltungen, d.h. sowohl bei Addier/Subtrahierwerken als auch bei schnellen Parallelmultiplizierern eine wichtige Rolle spielt. Wenn man etwas vereinfachend davon ausgeht, daß die beiden UND- sowie das ODER-Gatter der konventionellen Lösung in Bild 2.12 jeweils mit 4 Transistoren realisierbar sind, dann erhält man zusammen mit den beiden Invertern eine Gesamtzahl von 16 Transistoren. Im Vergleich dazu ermöglicht die Transfergatetechnik den Aufbau der EXOR-Funktion mit nur 8 Transistoren. Man erkennt leicht, daß sich zudem auch noch ein Geschwindigkeitsvorteil ergibt, weil nur zwei ’Stufen’ zu durchlaufen sind, während es bei der konventionelle Lösung drei sind.
Die elektrische Funktion des EXOR-Gatters in Transfergatetechnik ist einfach zu verstehen, wenn man sich wie in Bild 2.9 dargestellt, klarmacht, daß ein Transfergate nur dann eingeschaltet ist, wenn an seinem n-Kanal-Transistor ein ’1’-Pegel anliegt – der zugehörige p-Kanaltransistor hat dann immer ’0’-Pegel. Der Inverter an Eingang b sorgt in Bild 2.12 stets für ’komplementäre’ Pegelverhältnisse an beiden Transfergates (TG). Somit ist das obere TG bei ’1’-Pegel an b durchgeschaltet, während das untere bei ’0’-Pegel an b leitet. Bei ’1’-Pegel an b wird somit der invertierte Pegel von a an c erscheinen – damit ergeben sich die Zeilen 2 und 4 der Wertetabelle. Bei ’0’-Pegel an b ist nur das untere TG leitend, so daß a direkt auf c weitergeleitet wird – damit hat man die restlichen Zeilen 1 und 3 der Wertetabelle abgedeckt.

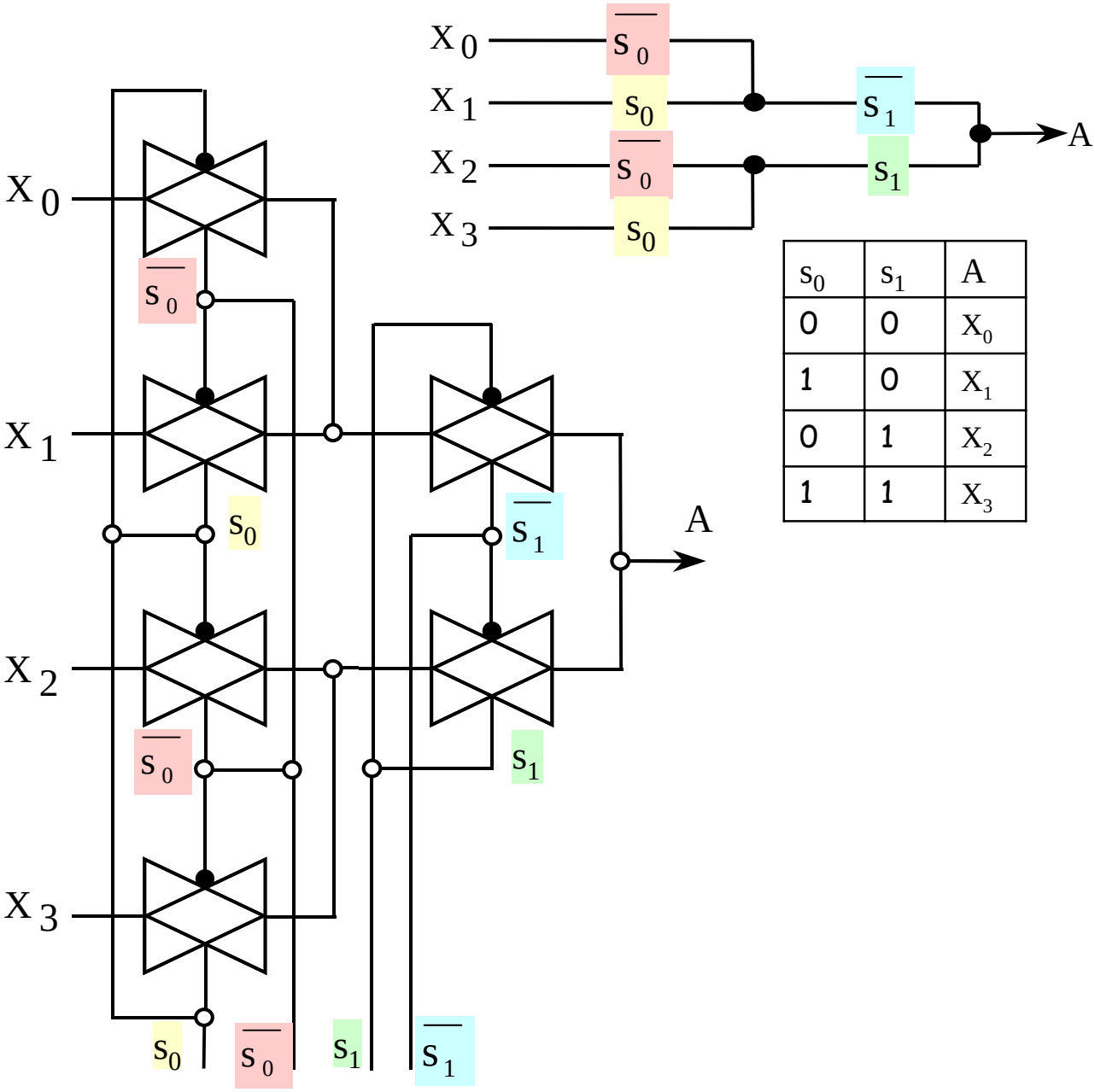
### 2.4.2 Multiplexer

Als abschließendes Beispiel wird ein 4-zu-1-Multiplexer betrachtet. Große Multiplexer, d.h. solche mit einer hohen Zahl von Eingängen, werden in konventioneller Gattertechnik sehr aufwendig. Die CMOS–Transfergatetechnik bietet auch hier elegante und leicht überschaubare Möglichkeiten wie im folgenden anhand Bild 2.13 gezeigt wird.

Ein Multiplexer hat in der Regel viele Eingangssignale $X_i$, von denen eines mit Hilfe von genau $\log_2(X_i)$ Steuersignalen $s_i$ ausgewählt, d.h. auf den Ausgang A durchgeleitet wird.

In der oberen Bildhälfte ist die abstrahierte Schalterdarstellung zu sehen, die praktisch unmittelbar auf die Transfergaterealisierung kopiert werden kann.

Auch hier muß man sich klarmachen, daß ein Transfergate nur dann eingeschaltet ist, wenn an seinem n-Kanal-Transistor ein ’1’-Pegel anliegt, während der zugehörige p-Kanaltransistor dann ’0’-Pegel hat. Wenn $s_0$ und $s_1$ beide ’0’-Pegel haben, führen ihre Komplemente ’1’-Pegel, so daß die beiden obersten TGs in Bild 2.13 leitend werden und $X_0$ auf $A$ durchschalten – dies deckt die erste Zeile der Funktionstabelle ab. Für die zweite Zeile gilt $s_0$=’1’ und $s_1$=’0’. Man sieht sofort, daß jetzt $X_1$ auf $A$ gelangt. Der Rest der Funktionstabelle ist offensichtlich und bedarf keiner weiteren Erläuterung.




| $s_0$ | $s_1$ | A |
|---|---|---|
| 0 | 0 | $X_0$ |
| 1 | 0 | $X_1$ |
| 0 | 1 | $X_2$ |
| 1 | 1 | $X_3$ |

Bild 2.13: 4-zu-1-Multiplexer, realisiert mit Transfergatetechnik

# 3 Sequentielle Schaltungen, Automaten und Speicher

## 3.1 Sequentielle Grundschaltungen

Sequentielle Schaltungen sind durch Auftreten von Zustandsänderungen gekennzeichnet, und zwar in der Weise, daß sie gemäß einer Funktion $f$ Folgezustände $Q^{n+1}$ einnehmen, die von Eingangssignalen $e_i$, aktuellen Zuständen $Q^n$ und einer diskreten, in der Regel von einem Taktgeber bestimmten Zeit $t$ abhängen. Somit ergibt sich die Beschreibungsgleichung $Q^{n+1} = f(e_i, Q^n, t)$.

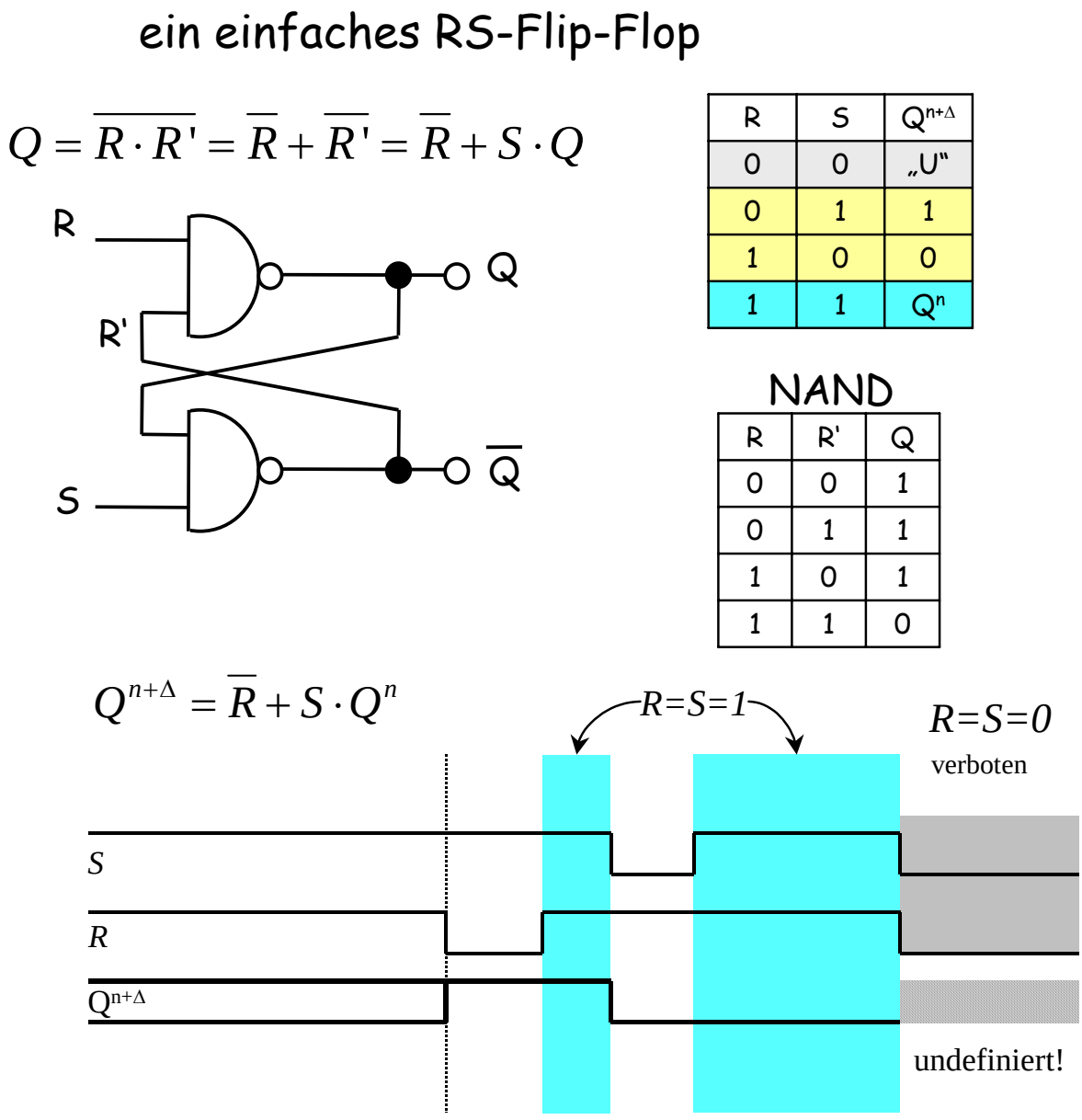



Bild 3.1: RESET-SET (RS) Flip-Flop mit Wertetabellen und Signalverlauf über der Zeit

Beim RS-Flip-Flop gibt es keinen Takt, so daß Zustandsänderungen unmittelbar nach gewissen Gatterdurchlaufzeiten auftreten. Dies wird durch Einführung eines „kleinen" Zeitintervalls $\Delta$ berücksichtigt. Wie aus der Schaltung in Bild 3.1 ersichtlich ist, gilt $R' = \overline{S \cdot Q}$, bzw. $\overline{R'} = S \cdot Q$; daraus folgt unmittelbar

$$Q^{n+\Delta} = \overline{R} + S \cdot Q^n \qquad (3.1)$$

(3.1) besagt somit, daß der Ausgang $Q$ des RS-Flip-Flops den Wert '1' annimmt, wenn $R$=0 wird, wobei $S$=1 sein muß, oder wenn $Q$=1 und $S$=$R$=1 sind. Der Fall $S$=$R$=0 muß ausgeschlossen wer-

den, denn er würde dazu führen, daß beide Ausgänge $Q$ und $\overline{Q}$ gleichzeitig den Wert '1' hätten. Zum einen würde das Verhalten der Schaltung jetzt nicht mehr einem Flip-Flop entsprechen, und zum anderen wäre der Zustand nach Änderung der Eingangsbelegung z.B. auf $S$=$R$=1 undefiniert.

Die technischen Einsatzmöglichkeiten eines RS-Flip-Flops sind somit sehr begrenzt. Wie wir noch sehen werden, taugt es nicht zum Aufbau von Registern, Zählern oder synchronen Schaltwerken aller Art, die innerhalb von Mikrorechnern unentbehrlich sind. In der Speichertechnik hingegen können RS-Flip-Flops gut eingesetzt werden und ermöglichen den Aufbau sehr einfacher Zellen, was eine entscheidende Voraussetzung für hochintegrierte Speicher mit großer Kapazität ist.

Ein wichtiger Schritt in Richtung universell verwendbarer Speicherelemente in Form sequentieller Schaltungen ist die Einführung eines Taktes, nach dessen Maßgabe, d.h. im Rhythmus der Taktdauer, nun die Zustandsänderungen gesteuert werden.

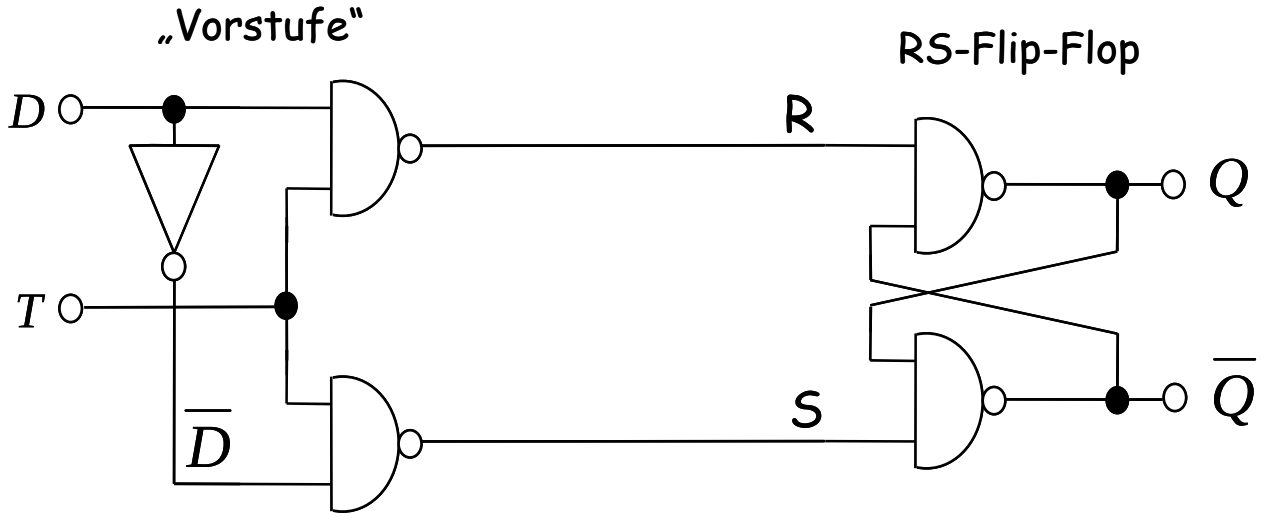



Bild 3.2: Entwicklungsschritt vom RS-FF zum taktgesteuerten D-Flip-Flop

Wie man sieht, ist in Bild 3.2 ein Vorstufe zum bekannten RS-Flip-Flop hinzugefügt worden, so daß nach Außen nun die Eingangssignale $D$ und $T$ auftreten und

$$R = \overline{D \cdot T} \quad \text{und} \quad S = \overline{\overline{D} \cdot T} = D + \overline{T} \tag{3.2}$$

gilt. Damit läßt sich die Grundgleichung für das RS-Flip-Flop wie folgt modifizieren:

$$Q^{n+1} = \overline{R} + S \cdot Q^n = D \cdot T + D \cdot Q^n + \overline{T} Q^n = D \cdot T + \left(D + \overline{T}\right) \cdot Q^n \tag{3.3}$$

Mit einigen Zwischenschritten kann (3.3) noch weiter vereinfacht werden.

$$\left(D + \overline{T}\right) \underbrace{\left(T + \overline{T}\right)}_{\text{Erweiterung}} \cdot = D \cdot T + D \cdot \overline{T} + \underbrace{\overline{T} \cdot T}_{0} + \overline{T} \cdot \overline{T} = D \cdot T + D \cdot \overline{T} + \overline{T} \tag{3.4}$$

Man erhält schließlich

$$Q^{n+1} = D \cdot T + \left(D \cdot T + D \cdot \overline{T} + \overline{T}\right) \cdot Q^n = D \cdot T \left(1 + Q^n\right) + \overline{T} \cdot (1 + D) \cdot Q^n \; = D \cdot T + \overline{T} \cdot Q^n \tag{3.5}$$

zur Beschreibung eines taktgesteuerten D-Flip-Flops nach Bild 3.2.

Das Impulsdiagramm in Bild 3.3 zeigt das Zeitverhalten des taktgesteuerten D-Flip-Flops und offenbart die wichtigsten Nachteile, die auch dieses Flip-Flop für eine Vielzahl von Mikrorechnerfunktionen ungeeignet erscheinen lassen. Wie wir noch sehen werden, ist es aber als „Peripheriebaustein“ bei Mikrocontrollern mit „gemultiplextem“ Daten/Adreßbus vorteilhaft einsetzbar.

$$Q^{n+1} = D \cdot T + \overline{T} \cdot Q^n$$

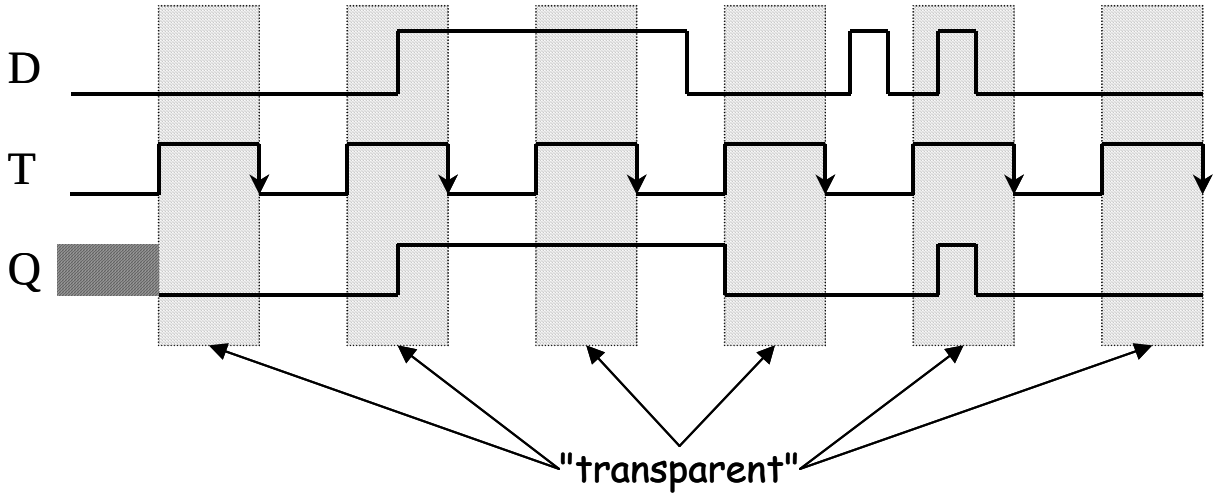



Bild 3.3: Zeitverhalten des taktgesteuerten D-Flip-Flops

Wie man in Bild 3.3 sieht, ist das Flip-Flop während der ’1’-Phase des Taktes immer „transparent“, d.h. der Zustand am *D*-Eingang erscheint – verzögert um ca. 2 NAND–Gatterlaufzeiten – am Ausgang *Q*. Daraus ergibt sich trotz der Taktsteuerung ein asynchrones Verhalten, was bedeutet, daß sich die Ausgangsignale des Flip-Flops während der ’1’–Phase zu beliebigen Zeiten ändern können, wenn der Eingang *D* dies vorgibt. Während der ’0’–Phase wird dagegen jede Zustandsänderung blockiert, so daß Änderungen von *D* in dieser Zeit keine Wirkung haben. Ein gewisses taktsynchrones Verhalten ist an den Taktflanken gegeben. Bei der Vorderflanke wird so der Zustand von *D* unmittelbar an *Q* weitergegeben, und bei der Rückflanke wird der Zustand von *D* „eingefroren“, bis die nächste ’1’–Phase beginnt.

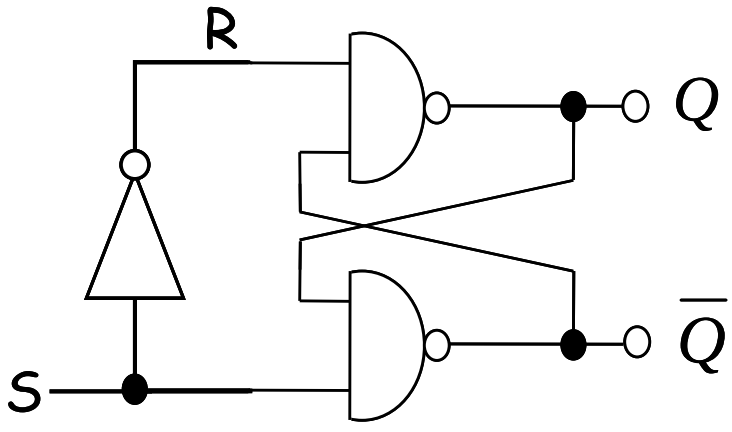



Bild 3.4: Unbrauchbare Idee, um S=R=0 auszuschließen

Einen naheliegenden Gedanken zum Ausschluß der verbotenen Belegung *S*=*R*=0, der aber sofort wieder verworfen werden muß, zeigt Bild 3.4. Würde man mit dem Inverter tatsächlich $R = \overline{S}$ erzwingen, hieße das:

$$Q^{n+\Delta} = \overline{R} + S \cdot Q^n = S + S \cdot Q^n \equiv S$$

Somit ginge die Speicherfunktion vollständig verloren und der Ausgang *Q* würde *S* unmittelbar folgen – lediglich verzögert um die entsprechenden Gatterlaufzeiten.

Das betrachtete taktgesteuerte D-Flip-Flop wird häufig auch als D–Latch[5] oder einfach nur als Latch bezeichnet. Aus Bild 3.3 sind bereits die wesentlichen Nachteile für praktische Anwendungen sichtbar geworden. Das folgende Bild 3.5 vertieft die Problematik noch etwas weiter, indem reale Signalverläufe betrachtet werden. In der Praxis – auch in der Digitaltechnik – gibt es selten saubere rechteckförmige Signale. Man hat es stets mit mehr oder weniger bizarren Zeitverläufen zu tun, die es korrekt zu interpretieren gilt.

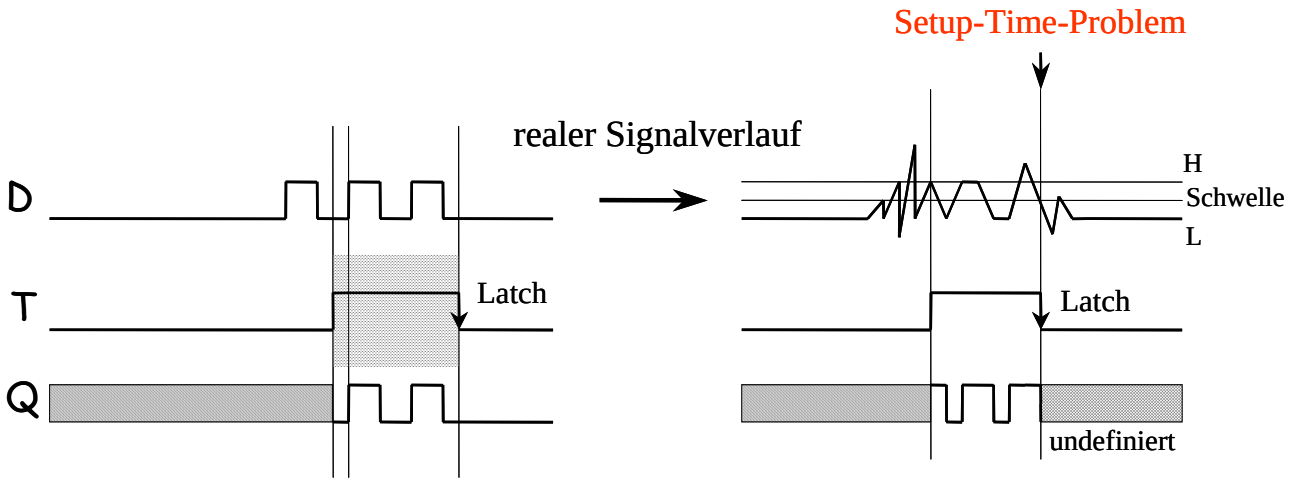



Bild 3.5: Latch–Probleme bei realen Eingangssignalen

Zur Lösung der hier kurz umrissenen Probleme ist ein weiterer Schritt im Aufbau von Speicherelementen in Form von Flip-Flops erforderlich, der zur Taktflankensteuerung hinführt. Dadurch werden Übernahmezeitpunkte von Signalen präzisiert, d.h. nur noch an den Taktflanken (entweder an Vorder- oder Rückflanke) wird das Eingangssignal „abgetastet“ und der detektierte Wert wird übernommen. In allen übrigen Zeitbereichen kann das Eingangssignal einen beliebigen Verlauf aufweisen, ohne daß der Flip-Flop-Zustand davon beeinflußt wird. Diese Fähigkeit ist z.B. für den Aufbau von Bussystemen unerläßlich.

In der Praxis muß zum Erzielen einer korrekten Arbeitsweise sichergestellt werden, daß keine Pegeländerung des Eingangssignals exakt mit der relevanten Taktflanke zusammenfällt. Darüber hinaus muß das Eingangssignal eine gewisse Zeit (Setup Time) stabil sein, bevor die relevante Taktflanke auftritt. Der Grund dafür sind technologiebedingte Verzögerungszeiten. In der heutigen CMOS—Technologie bewegen sich Setup-Zeiten im Bereich von ca. 100ps...1ns. Da in der CMOS-Technologie aufgrund von Kapazitäten Speichereffekte für Signale vorhanden sind, kann auf eine „Hold“–Time, d.h. ein Aufrechterhalten des Pegels für eine gewisse Zeit nach der relevanten Taktflanke verzichtet werden (Hold-Time=0).

Bevor die technische Realisierung der Taktflankentriggerung untersucht wird, betrachten wir zunächst den Aufbau eines taktgesteuerten D-Flip-Flops nach Bild 3.2 unter Verwendung von Transfergates. Hierbei ergeben sich sehr einfache und übersichtliche Lösungen, die der CMOS–Technologie im Vergleich mit bipolaren Technologien einen klaren Vorsprung gesichert haben. Es ist leicht zu erkennen, daß die Schaltung nach Bild 3.6 sich genauso verhält wie das D-Flip-Flop nach Bild 3.2. Für $T$=1, d.h. ($\overline{T}=0$) ist das eingangsseitige Transfergate leitend, so daß der Zustand von $D$ unmittelbar auf $Q$ durchgeschaltet wird Für $T$=0, d.h. ($\overline{T}=1$) hingegen wird der Eingang $D$ isoliert und das 2. Transfergate leitend, so daß eine Rückkoppelschleife über die beiden Inverter entsteht, womit der Zustand von $Q$ ’eingefroren’ wird.

[5] bedeutet ’festhalten’

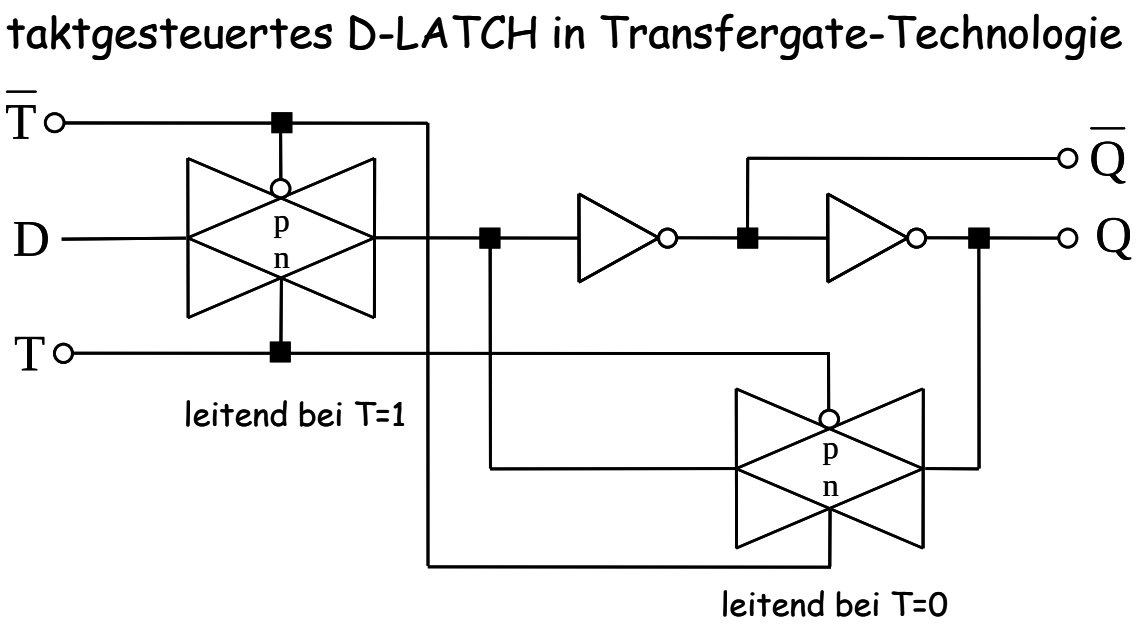



Bild 3.6: D–Latch mit Transfergates: T=1 ⇒ transparent; T=0 Zustand 'eingefroren'

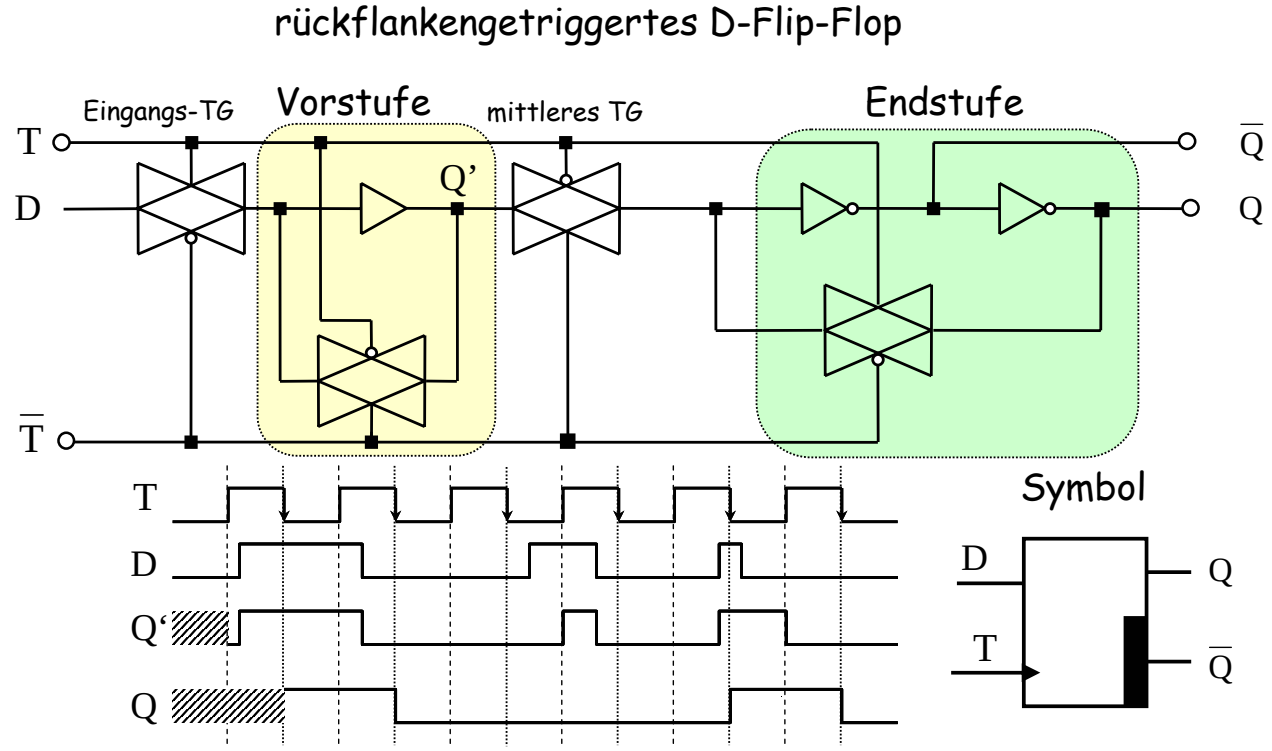



Bild 3.7: Einführung der präzisen Taktflankentriggerung

Wenn man die Schaltung nach Bild 3.8 jetzt noch mit einer „speicherfähigen" Vorstufe, die aus einem Buffer (nichtinvertierender Verstärker), der über ein Transfergate rückgekoppelt ist und mit einem weiteren eingangsseitigen Transfergate versieht, dann entsteht das taktflankengetriggerte D-Flip-Flop nach Bild 3.7. Für *T*=1 ist jetzt nur noch die Vorstufe transparent, d.h. *Q*'=*D*. Die Endstufe ist über ihre Rückkopplung im Haltezustand und durch das gesperrte mittlere Transfergate von der Vorstufe abgetrennt. Wechselt jetzt der Takt auf *T*=0, dann geht die Vorstufe in den Haltezustand, *D* wird isoliert und das mittlere Transfergate gibt den Wert von *Q*', der an der Taktrückflanke vorhanden war, an den Ausgang *Q* weiter. Man hat damit das gewünschte taktflankensynchrone Verhalten, d.h. der Ausgang *Q* kann sich nur noch an Taktrückflanken ändern und übernimmt dort *Q*'.

### 3.1.1.1 Synchrone und asynchrone Realisierung digitaler Schaltungen

Am Beispiel eines kurzen Binärzählers (3bit) soll hier der Unterschied zwischen synchronen und asynchronen Digitalschaltungen herausgearbeitet werden. Anhand dieser Betrachtungen wird deutlich, warum beim Aufbau heutiger Mikrorechnerhardware – insbesondere beim Steuerwerk – synchrone Konzepte bevorzugt werden.

In einer asynchronen Digitalschaltung sind Zustandsänderungen zu jedem beliebigen Zeitpunkt erlaubt. Das Zeitverhalten wird somit essentiell durch unterschiedliche Durchlaufzeiten (bzw. Verzögerungen) einzelner Komponenten bestimmt. Bekanntlich hängt die Durchlaufzeit von Signalen entscheidend von der Größe und von der Struktur einer Komponente ab. In einer komplexen Digitalschaltung ist die Analyse des Zeitverhaltens eine schwierige und, aufgrund der hohen Anzahl möglicher Zustände und Zustandsübergänge, eine oft nicht vollständig lösbare Aufgabe. Man hat sozusagen ein „Zeitkontinuum“ vor sich, das man meßtechnisch z.B. mit einem hinreichend schnellen Oszillographen oder einem hochauflösenden Logikanalysator auf korrekte Funktion hin untersuchen müßte. Für den als Beispiel gewählten Zähler ist die Situation hingegen relativ leicht überschaubar, wobei sich die Problematik dennoch uneingeschränkt darstellen läßt.

Der asynchrone 3-bit-Zähler nach Bild 3.8 besteht aus drei rückflankengetriggerten D-Flip-Flops, die zu Beginn der Analyse alle zurückgesetzt sein sollen. Mit der ersten Taktrückflanke wird $Q_1$='1' und mit der zweiten wieder '0'. Dieses ist jetzt die relevante Rückflanke für das 2. Flip-Flop, mit der $Q_2$='1' wird. Eine weitere Rückflanke des Eingangstaktes $T$ setzt $Q_1$ wieder auf '1'. Das Timingdiagramm (Impulsdiagramm) zeigt die weiteren Abläufe. Man sieht, daß der Zähler nach dem beschriebenen Muster wie erwartet bis 7 zählt und dann wieder bei 0 beginnt. Wenn die beteiligten Flip-Flops keine Verzögerung hätten, also ideale Komponenten mit der Durchlaufzeit Null wären, ergäben sich in der Praxis keine Probleme. Das ist jedoch technisch nicht möglich. Zur Analyse betrachten wir den Übergang vom Zählerstand 7 auf 0 im grau schattierten Rechteck rechts unten in Bild 3.8 und nehmen an, daß für jedes der 3 Flip-Flops eine Verzögerung von 1ns zwischen Rückflanke am Takteingang und Zustandsänderung am Ausgang liegt. Daraus ergibt sich, daß der Zustand 0 des Zählers nicht sofort mit der zugehörigen Rückflanke des Eingangstaktes eingenommen wird, sondern mit 1ns Verzögerung kippt zuerst $Q_1$, so daß sich der Zählerstand 6 ergibt. Dieser bleibt genau für 1ns bestehen, bis die Rückflanke von $Q_1$ sich auf $Q_2$ fortgepflanzt hat. Damit hat man den Zählerstand 4, der wiederum auch nur für 1 ns erhalten bleibt, bis schließlich die Rückflanke von $Q_2$ auf $Q_3$ propagiert ist. Erst jetzt steht der Zähler auf dem Wert 0. Es ist unmittelbar einzusehen, daß die Sequenz der unerwünschten Zustandsübergange unabhängig von der Eingangtaktfrequenz immer so wie beschrieben abläuft, d.h. nur die Durchlaufzeit der Flip-Flops bestimmt hier das Zeitverhalten. Man kann sich leicht vorstellen, daß ein Dekoder für Zählerzustände durch das beschriebene „Metazustandsphänomen“ irritiert wird. Solche Metazustände gibt es nicht nur bei 7=>0, sondern sie können auch an anderen Übergängen auftreten. So geht z.B. der Zählerstand 3 keineswegs unverzüglich in 4 über, sondern zunächst in 2 und dann erst in 4. Es ist unmittelbar einsichtig, daß es bei sehr langen Zählern zu einer erheblichen Metazustandsdauer kommt.

Die synchrone Realisierung nach Bild 3.9 hingegen vermeidet die geschilderte Problematik, indem jedem einzelnen Flip-Flop der Eingangtakt $T$ zugeführt wird. Unter der Annahme gleich aufgebauter Flip-Flops kann man jetzt davon ausgehen, daß alle Zustandsübergänge gleichzeitig stattfinden. Wie man jedoch sieht, hat sich der Schaltungsaufwand vergrößert. Man muß jetzt logische Verknüpfungen der Ausgänge $Q_i$ der Vorstufen zur Erzeugung der Eingangssignale $D_i$ der Flip-Flops heranziehen, wobei der Aufwand von Stufe zu Stufe wächst. Es ist daher zweckmäßig, eine Minimierung zum Einsparen von Schaltungsaufwand durchzuführen. Dies ist hier exemplarisch anhand von Tabelle 3.1 und des darunter stehenden Karnough-Veitch-Diagramms gezeigt.

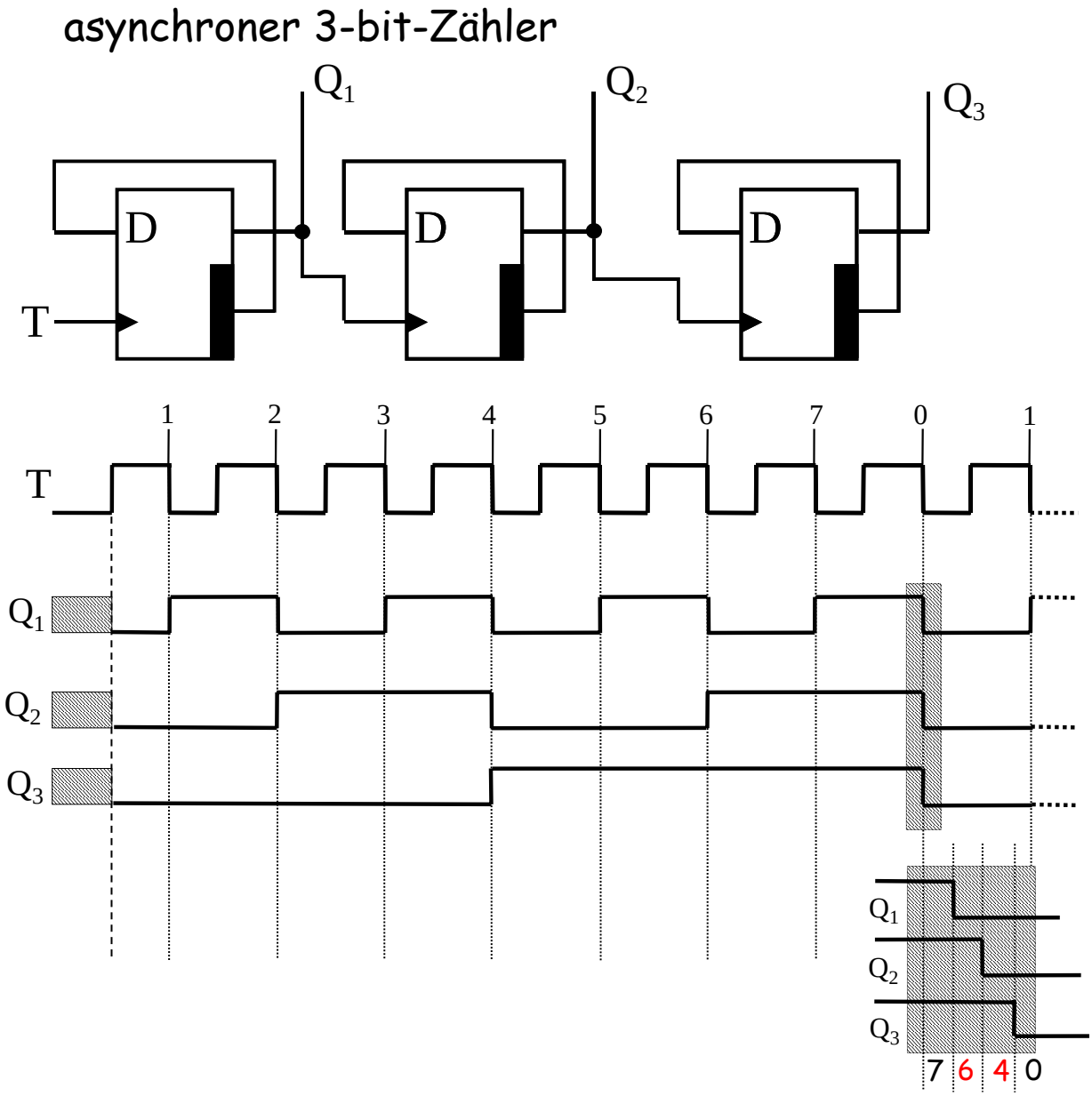



Bild 3.8: Asynchroner 3-bit-Binärzähler mit Timing-Diagramm

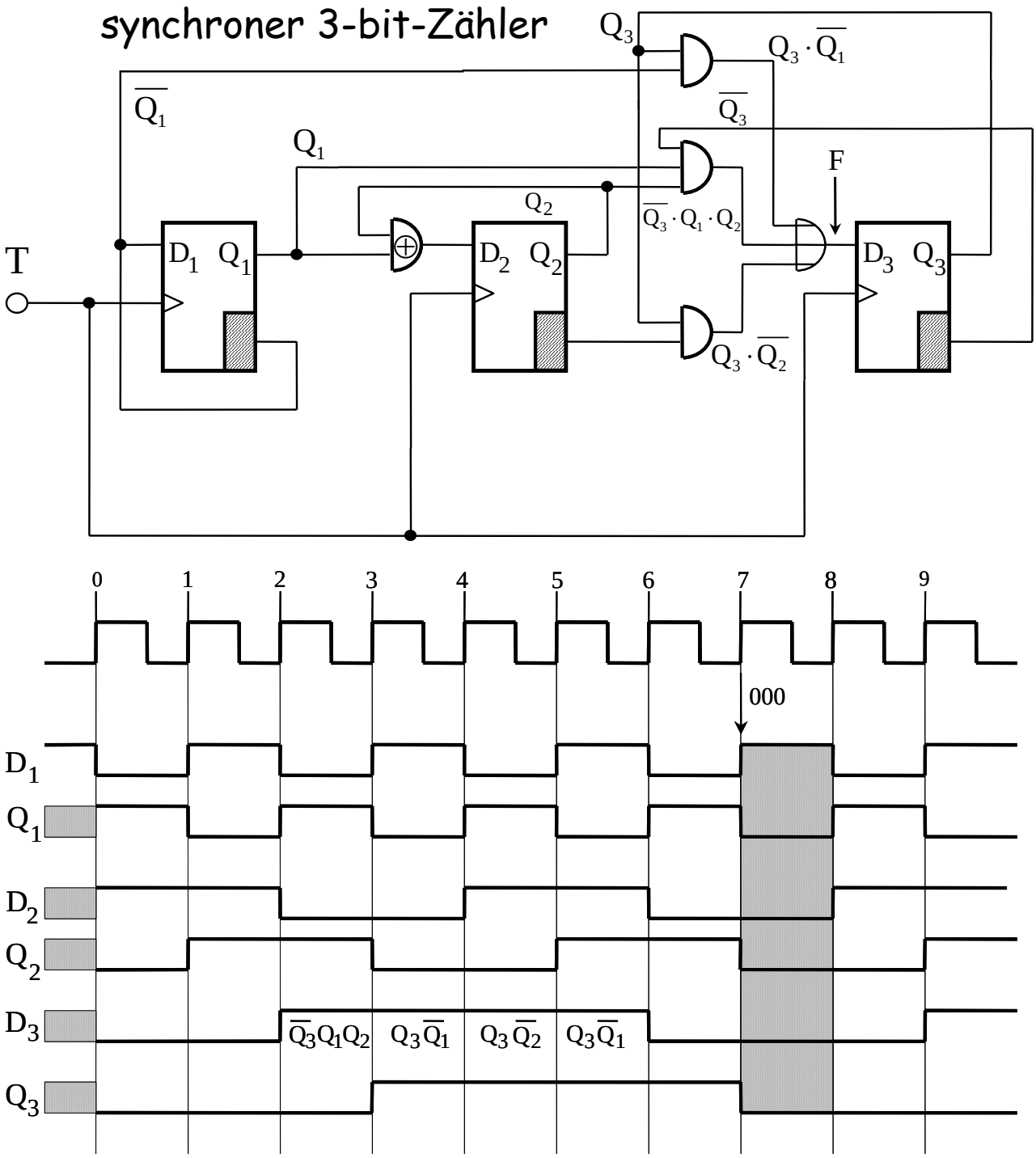



Bild 3.9: Aufbau des synchronen 3 bit–Zählers mit vorderflankengetriggerten D-Flip-Flops und das zugehörige Timing–Diagramm

Aus den obigen Betrachtungen wird deutlich, daß synchrone Digitalschaltungen trotz erhöhten Schaltungsaufwandes aus folgenden Gründen (fast) immer zu bevorzugen sind:

- Zustandsänderungen treten stets in festem Bezug zu einer Taktflanke auf – daraus ergeben sich einfache und übersichtliche Testmöglichkeiten
- auch bei großen, komplexen Schaltungen kommt es nicht zwangsläufig zu einer Summierung von Verzögerungszeiten, d.h. synchrone Schaltungen können mit schnelleren Takten betrieben werden als asynchrone

# 3-stufiger synchroner Binärzähler

Tabelle 3.1: Zur Aufstellung der booleschen Gleichungen für die D-Eingänge

| $Q_3$ | $D_3$ | $Q_2$ | $D_2$ | $Q_1$ | $D_1$ | Takt |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 1 | $\rightarrow 0$ |
| 0 | 0 | 0 | 1 | 1 | 0 | $\rightarrow 1$ |
| 0 | 0 | 1 | 1 | 0 | 1 | $\rightarrow 2$ |
| 0 | 1 | 1 | 0 | 1 | 0 | $\rightarrow 3$ |
| 1 | 1 | 0 | 0 | 0 | 1 | $\rightarrow 4$ |
| 1 | 1 | 0 | 1 | 1 | 0 | $\rightarrow 5$ |
| 1 | 1 | 1 | 1 | 0 | 1 | $\rightarrow 6$ |
| 1 | 0 | 1 | 0 | 1 | 0 | $\rightarrow 7$ |
| 0 | 0 | 0 | 0 | 0 | 1 | $\rightarrow 8$ |
| 0 | 0 | 0 | 1 | 1 | 0 | $\rightarrow 9$ |

$$D_1 = \overline{Q_1}$$

$$D_2 = Q_1 \oplus Q_2$$

$$D_3 = \overline{Q_3} \cdot Q_1 \cdot Q_2 \quad \leftarrow 011$$
$$+ Q_3 \cdot \overline{Q_1} \cdot \overline{Q_2} \quad \leftarrow 100$$
$$+ Q_3 \cdot \overline{Q_2} \cdot Q_1 \quad \leftarrow 101$$
$$+ Q_3 \cdot Q_2 \cdot \overline{Q_1} \quad \leftarrow 110$$

KV-Diagramm

| | $Q_1$ | | $\overline{Q_1}$ | |
|---|---|---|---|---|
| $Q_3$ | 0 | 1 | 1 | 1 |
| $\overline{Q_3}$ | 1 | 0 | 0 | 0 |
| | $Q_2$ | $\overline{Q_2}$ | | $Q_2$ |

Minimierung ergibt:

$$F = Q_3 \cdot \overline{Q_1} + Q_3 \cdot \overline{Q_2} + \overline{Q_3} \cdot Q_1 \cdot Q_2$$

Bei Zählern ist es oft erforderlich, daß man sie auf einen beliebigen Zustand setzen kann, bevor der Zählvorgang beginnt. In Mikrorechnern wird diese Fähigkeit insbesondere bei Timern benötigt. Wie wir noch im Detail sehen werden, sind ladbare Zähler das Kernstück eines jeden Timersystems. Bild 3.10 zeigt die ersten beiden Stufen des synchronen Zählers aus Bild 3.9, wobei in die Verbindungen zu den D-Eingängen jeweils ein Umschalter eingefügt wurde. Im Zählbetrieb befinden sich die Schalter in ihrer unteren Stellung, so daß D-Verbindungen wie in Bild 3.9 vorhanden sind. Das Steuersignal Load/$\overline{\text{Count}}$ hat den Pegel '0', wobei $\overline{\text{Count}}$ andeutet, daß die Zählfunktion bei diesem Pegel aktiviert ist. Bei '1' hingegen (Load-Funktion) werden die D-Eingänge mit externen Datenleitungen $D_{1L}$, $D_{2L}$… verbunden, über die die einzuschreibenden Werte taktsynchron in den Zähler übernommen werden. Beim Betrieb ist darauf zu achten, daß sich das Steuersignal Load/$\overline{\text{Count}}$ und die zu ladenden Daten nicht gleichzeitig mit der relevanten Taktflanke ändern.

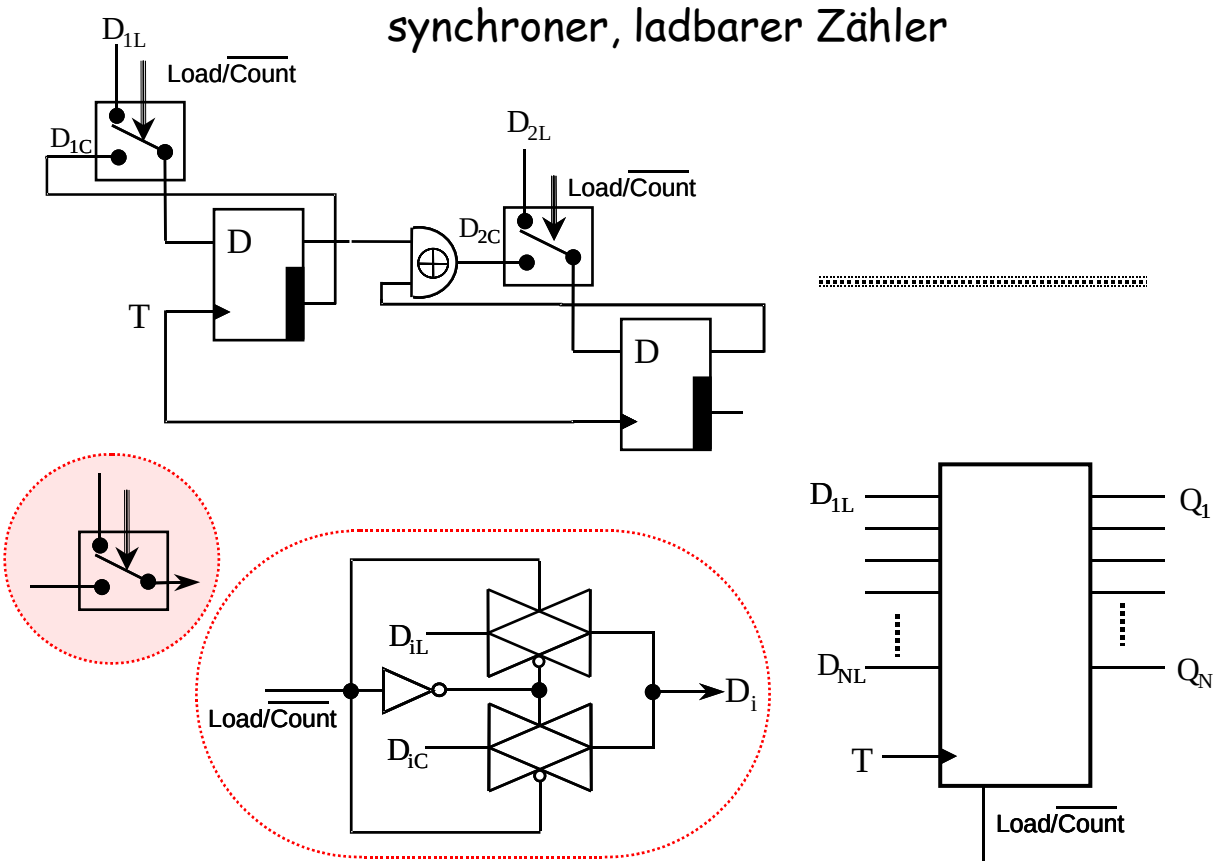



Bild 3.10: Vorrichtung zum Laden eines synchronen Zählers

Im unteren Teil von Bild 3.10 ist eine Möglichkeit zum Aufbau der Umschalter mit Hilfe von 2 Transfergates gezeigt. Der Inverter stellt das benötigte Komplement des Steuersignals Load/$\overline{\text{Count}}$ bereit, so daß für den Ladevorgang bei Load/$\overline{\text{Count}}$ ='1' das obere TG die externen Daten weitergibt, während bei '0' das untere die für das Zählen benötigte Verbindung herstellt.

### 3.1.2 Halb- und Volladdierer

## Halbaddierer

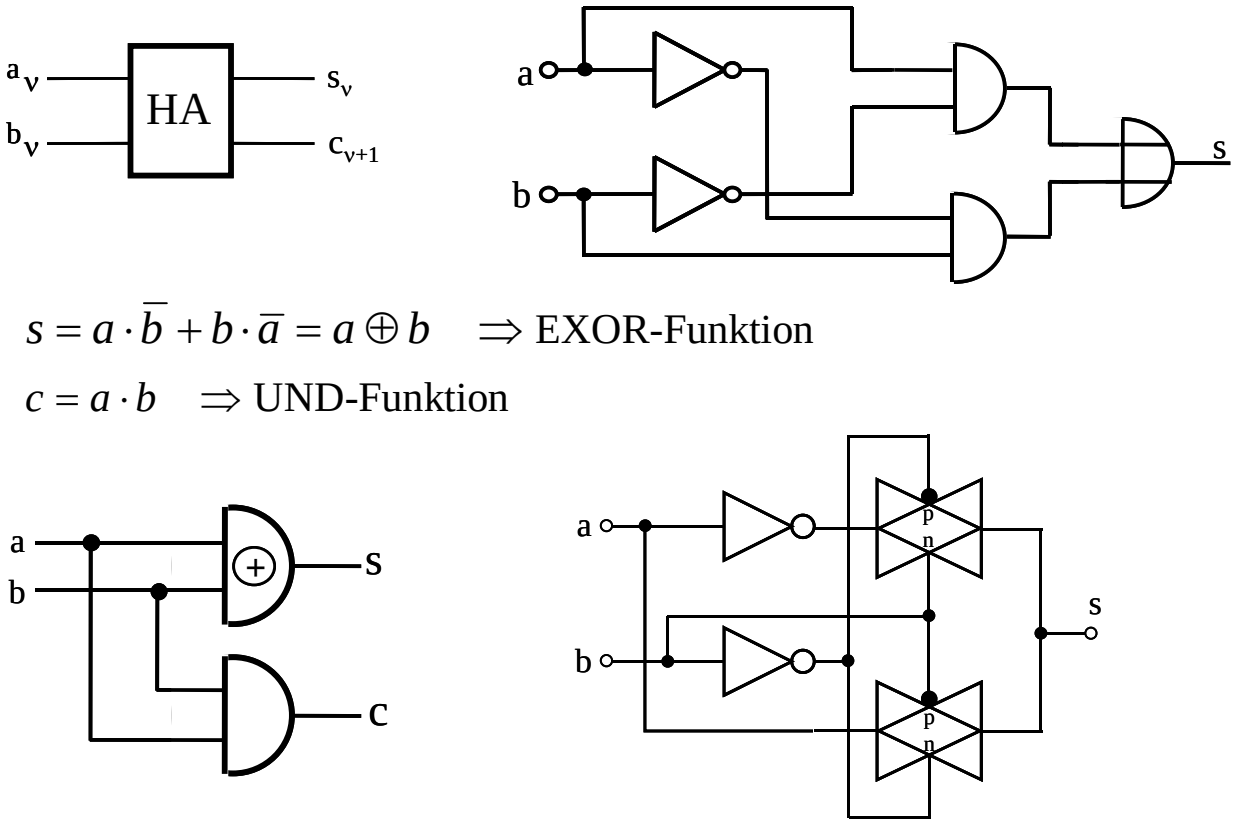



Bild 3.11: Halbaddierer als einfache Rechenschaltung zur „Bitaddition“

Eine Addierschaltung für 2 bit ist sehr einfach zu realisieren. Für die Bildung der Summe wird weiter nichts als das bekannte EXOR-Gatter benötigt. Wie wird schon gesehen haben, läßt es sich in CMOS-Technologie vorteilhaft mit Transfergates aufbauen. Der vollständige Addierer erfordert des weiteren noch eine Übertragsberechnung, die sich hier leicht mit einem UND-Gatter realisieren läßt. Wir sprechen von der in Bild 3.11 dargestellten Schaltung von einem Halbaddierer, weil sie sich in dieser Form noch nicht als Grundelement für den Aufbau komplexerer Addierer für beliebig große Wortlängen eignet. Was dazu noch fehlt, ist die Möglichkeit, auch eingangsseitig hereinkommende Überträge zu verarbeiten. Ein solcher Volladdierer bekommt also 3 Eingangsbits und erzeugt daraus ein Summenbit und ein Übertragsbit. Die zugehörigen Rechenregeln sind einfach:

Das Summenbit $s_\nu$ wird nur dann ’1’, wenn eine ungerade Anzahl von ’1’-Bits an den Eingängen anliegt, d.h. man muß eine modulo-2-Addition durchführen, die sich – wie in Bild 3.12 gezeigt ist – mit 2 EXOR-Gattern bewerkstelligen läßt.

Der Ausdruck für den Übertrag ist ein wenig komplizierter: $c_{\nu+1}$ nimmt den Wert ’1’an, wenn mindestens 2 der 3 Eingangsbits ’1’ sind, d.h. auch wenn alle 3 ’1’ sind, also:

$$c_{\nu+1} = \overline{a_\nu} b_\nu \cdot c_\nu + a_\nu \overline{b_\nu} \cdot c_\nu + a_\nu b_\nu \cdot \overline{c_\nu} + a_\nu b_\nu \cdot c_\nu .$$

Eine einfache Umformung ergibt $c_{\nu+1} = a_\nu b_\nu + c_\nu \cdot \left[ a_\nu \oplus b_\nu \right]$.

Wie man in der Schaltung in Bild 3.12 (unten) sieht, lassen sich die beiden Eingangssignale für das ODER-Gatter, das $c_{\nu+1}$ liefert, sehr einfach mit zwei UND-Gattern aus vorhandenen Signalen gewinnen.

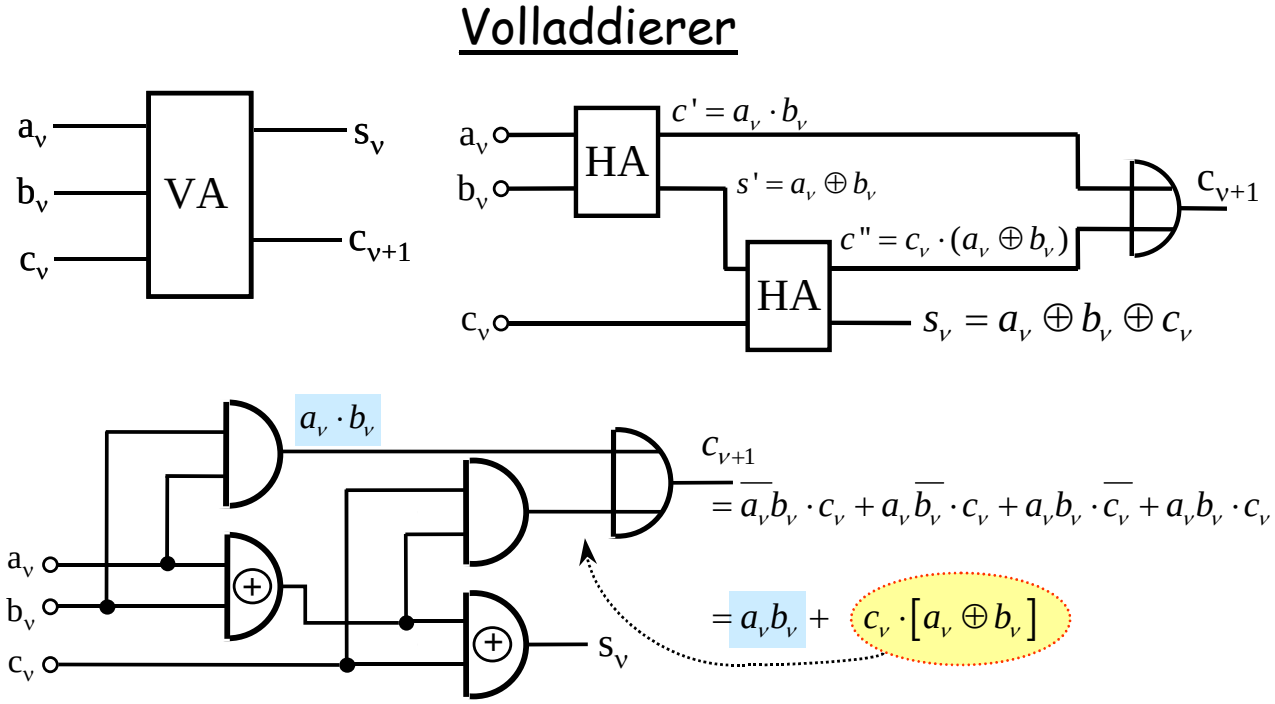
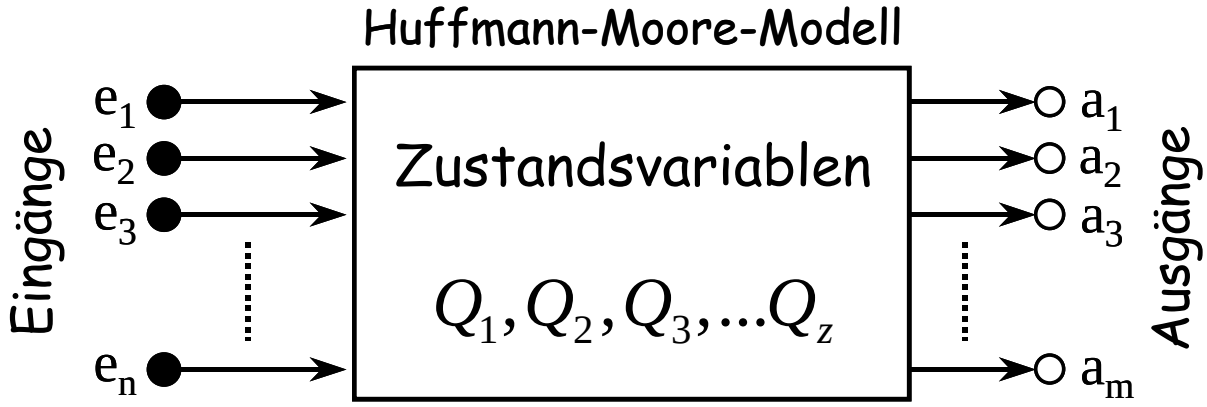



Bild 3.12: Der Volladdierer als Grundelement zum Aufbau komplexer Rechenschaltungen

## 3.2 Schaltnetze, Schaltwerke, endliche Automaten




Ausgänge:

$$a_i^t = f_i\left(e_1, e_2, ..., e_n; Q_1, Q_2, Q_3, ...Q_z\right)^t \quad \text{mit } i = 1, 2, ...m$$

Zustände:

$$Q_i^{t+1} = g_i\left(e_1, e_2, ..., e_n; Q_1, Q_2, Q_3, ...Q_z\right)^t \quad \text{mit } i = 1, 2...z$$

Bild 3.13: Grundlegendes, allgemeines Automatenmodell

Für den Aufbau von Mikrorechnern haben endliche Automaten eine besondere Bedeutung, weil mit ihnen die Steuerwerke dieser Rechner in übersichtlicher Weise realisiert werden können. Die Bezeichnung 'endlich' drückt aus, daß eine solche Maschine[6] nur eine bestimmte – eben endliche – Anzahl von Zuständen annehmen kann, die in klar vorgegebener Reihenfolge zur Erfüllung einer Aufgabe ablaufen. Bevor typische Aufgaben und die dazugehörigen Abläufe näher spezifiziert werden, untersuchen wir zunächst die Entstehungsweise und den Grundaufbau endlicher Automaten anhand einiger Grafiken. Das in Bild 3.13 dargestellt Huffmann-Moore-Modell gibt einen sehr allgemeinen Überblick. Hier werden Eingangssignale $e_i$, Ausgangssignale $a_i$ und Zustände $Q_i$ und die Relationen zwischen ihnen definiert. Eine wichtige Größe, die indirekt in diesem Modell enthalten ist, ist die Zeit, die in den Modellgleichungen hochgestellt als $t$ und $t$+1 erscheint. Daraus ersieht man schon, daß es sich um eine diskretisierte Darstellung handelt, die technisch einfach mit einem Taktgeber zu realisieren ist.

Betrachtet man nun die $m$ Ausgänge $a_i$ zu einem bestimmten Zeitpunkt $t$, dann sind sie über Funktionen $f_i$, die sich in irgendeiner Weise mit booleschen Gleichungen beschreiben lassen, von den $n$ Eingangssignalen $e_i$ und den $z$ aktuellen Zuständen $Q_i$ zum Zeitpunkt $t$ abhängig. Während die Zustände sich nur in einem bestimmen, durch den erwähnten Takt vorgegebenen Zeitraster ändern können, sind die Eingangssignale in der Regel als 'zeitkontinuierlich' zu betrachten, was bedeutet, daß in jedem beliebigen Moment Änderungen möglich sind. Solche Änderungen können sich im allgemeinen Modell unmittelbar – d.h. nach einer gewissen Durchlaufverzögerung – auf die Ausgangssignale auswirken. Zustandsänderungen sind hingegen immer an ein Zeitraster gebunden, d.h. ein Folgezustand $Q_i^{t+1}$ ergibt sich immer nach einem fest vorgegebenen Zeitschritt nach Maßgabe einer booleschen Funktion $g_i$, die die am Ende dieses Zeitschrittes (meist definiert durch Taktvorderflanke oder auch Rückflanke) vorhandenen Werte von Eingangssignalen $e_i$ und Zuständen $Q_i$ verknüpft.

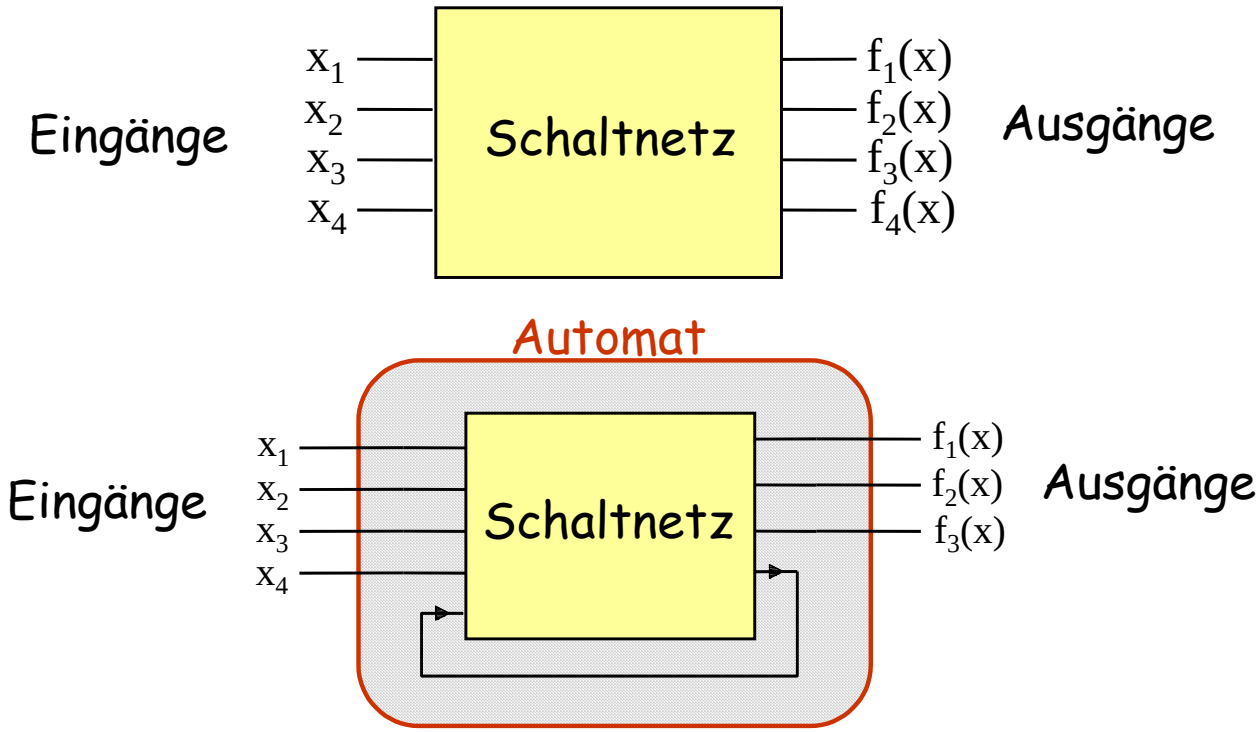



Bild 3.14: Vom Schaltnetz zum Schaltwerk (Automat)

[6] in der englischsprachigen Literatur ist die Bezeichnung 'Finite State Machine' (FSM üblich)

In Bild 3.14 ist ganz allgemein der Übergang von einem Schaltnetz zu einem Automaten skizziert. Unter einem Schaltnetz versteht man logische Schaltungen, die kein 'Gedächtnis' haben, d.h. keine Speicherfunktion beinhalten, sondern lediglich boolesche Verknüpfungen ausführen. Sobald man jedoch einen oder mehrere Ausgänge auf die Eingangsseite zurückführt ist dieses nicht mehr gegeben – es entsteht eine Speicherfunktion. Das ist sehr einfach und unmittelbar anhand des RS-Flip-Flops nach Bild 3.1 nachvollziehbar. In der Praxis vermeidet man aus Gründen der Übersicht und besseren Handhabbarkeit Strukturen nach Bild 3.14 (unten) und trennt - wie in Bild 3.15 gezeigt - Schaltnetz und Speicher sauber voneinander.

### 3.2.1 Moore- und Mealy-Automat

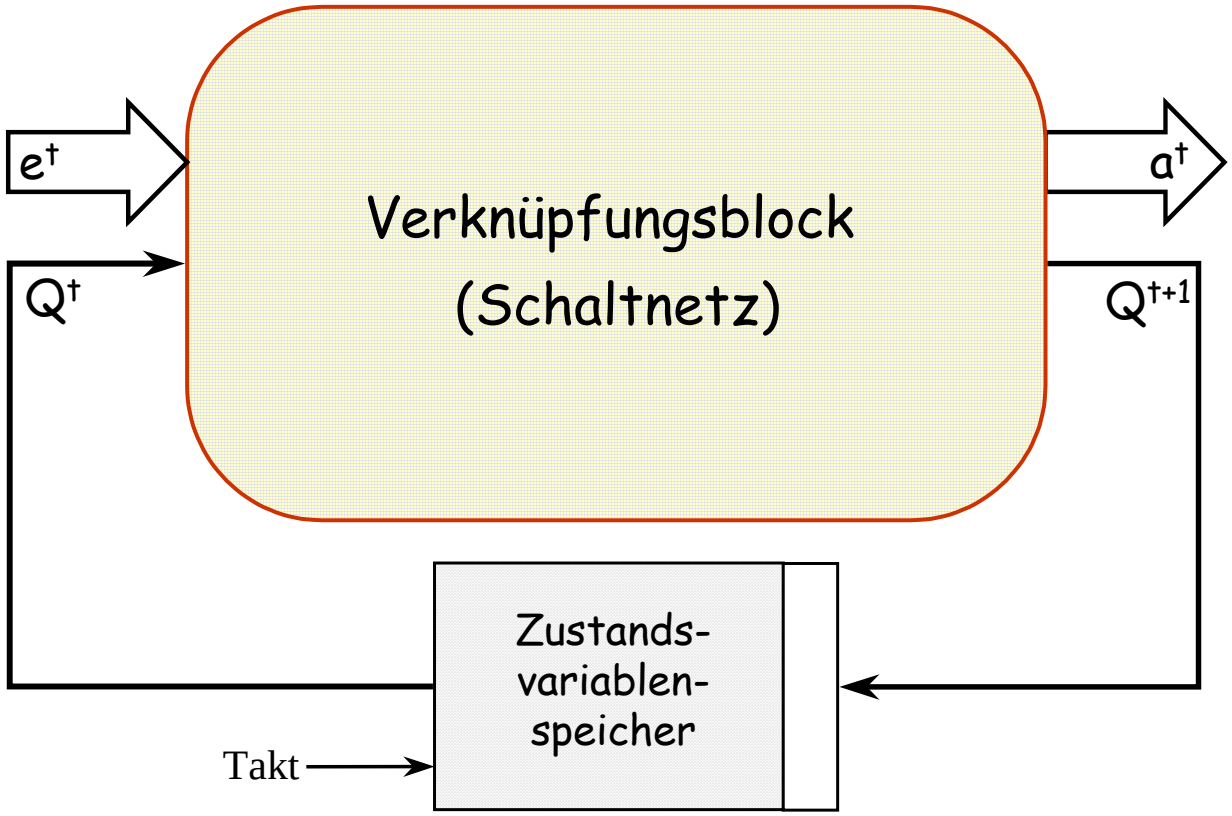



Bild 3.15: Klare Trennung von Schaltnetz und Speicher beim Aufbau eines Automaten

In dieser Darstellung spiegelt sich schon deutlich der Ansatz zu einer technischen Realisierung des Huffmann-Moore-Modells wieder. Es ist ein Takt vorhanden, nach dessen Maßgabe sich der Inhalt des Zustandsspeichers ändern kann, während innerhalb des Schaltnetzes die booleschen Funktionen $f_i$ und $g_i$ realisiert sind. Bild 3.15 gibt allerdings noch keinen Aufschluß darüber, in welchem Zeitraster sich die Ausgangssignale ändern können, d.h. wenn die die Eingangssignale als 'zeitkontinuierlich' zu betrachten sind, könnten auch in jedem beliebigen Moment Änderungen der Ausgangssignale möglich sein. Um in diesem Punkt Klarheit zu schaffen, werden die beiden technisch wichtigsten Varianten von endlichen Automaten untersucht, nämlich der Mealy und der Moore-Automat.

Dabei wird sich zeigen, daß bei einem Mealy-Automaten in der Tat in jedem beliebigen Moment Änderungen der Ausgangssignale möglich sind, wenn zeitkontinuierliche Eingangssignale vorliegen. Dagegen zeigt der Moore-Automat auch für diesen Fall an seinen Ausgängen immer zeitdiskretes Verhalten – d.h. Änderungen sind nur exakt im Taktraster möglich. Hieraus ergeben sich erhebliche technische Vorteile, die so gewichtig sind, daß in den Steuerwerken von Mikrorechnern nahezu ausschließlich Moore-Automaten zu finden sind. Der Mealy-Automat wird daher nur kurz der Vollständigkeit halber vorgestellt. Die grundsätzlichen Unterschiede im Aufbau von Mealy-

und Moore-Automaten sind – wie aus Bild 3.16 und Bild 3.17 hervorgeht - in der Tat ziemlich gering, sie haben aber dennoch gravierenden Einfluß auf das Verhalten.

Zunächst bestehen beide Automatentypen nach der allgemeinen Vorgabe aus Bild 3.15 aus einem Schaltnetz mit rein kombinatorischen Funktionen (Verknüpfungsblock) und einem mit einem Taktversorgten Speicher. Der Verknüpfungsblock enthält jeweils die mit dem Huffmann-Moore-Modell eingeführten booleschen Funktionen *f* und *g*.

In Bild 3.16, das den Moore-Automaten darstellt, erkennt man, daß die Ausgangssignale *a* die mittels der Funktion *g* gebildet werden, ausschließlich von gespeicherten Zustandswerten abhängen. Sie können sich somit nur synchron mit dem Takt des Speichers ändern. Ein neuer Inhalt des Zustandsspeichers ($Q^{t+1}$) wird dagegen mittels der Funktion *f* aus dem Eingangssignal *e* und den bisherigen Speicherwerten $Q^t$ gebildet und – natürlich ebenfalls taktsynchron - eingeschrieben.

Im Gegensatz dazu gibt es im Verknüpfungsblock eines Mealy-Automaten unmittelbare Verbindungen von Eingangssignalen *e* in das Ausgangsschaltnetz mit der Funktion *g*. Das bedeutet, daß hier die Ausgangssignale *a* sofort - d.h. lediglich verzögert um die entsprechende Durchlaufzeit des Schaltnetzes für *g* - am Ausgang wirksam werden. Somit besteht kein Synchronismus mehr mit dem Takt. Warum ist ein derartiges Verhalten in der Praxis nicht günstig?

- Das asynchrone Verhalten kann zu wesentlich schnelleren Änderungen der Ausgangssignale führen, als sie aufgrund der Taktfrequenz des Speichers zu erwarten sind. Nachfolgende Schaltungen können auf solche schnellen Änderungen fehlerhaft reagieren.
- Das Testen eines Automaten mit asynchronem Verhalten ist sehr viel aufwendiger, da es hier nicht genügt, nur einen engen Zeitbereich nach der relevanten Taktflanke zu beobachten, sondern es muß eine kontinuierliche Analyse mit genügend kleinem Zeitraster erfolgen.

Für die Konstruktion von Automaten haben sich Diagramme wie das in Bild 3.18 dargestellte bewährt. Mit Kreisen stellt man die Zustände dar und versieht sie mit eindeutigen Namen. Die Ausgangssignale *a*, die der Automat im jeweiligen Zustand erzeugt, kann man neben dem Kreis mit einer geeigneten Zahlenbasis notieren, z.B. binär von ’000’ bis ’100’ – wie in Bild 3.18 dargestellt. Bei größeren Zahlen empfiehlt sich eine dezimale oder hexadezimale Darstellung. Die Zustandsübergänge werden mit Pfeilen symbolisiert, denen die Übergangsbedingungen in Form von booleschen Gleichungen zugeordnet werden. Im dargestellten Beispiel sind alle Übergänge abhängig von dem Eingangssignal e, das hier eine Breite von 2bit hat. Wenn ein Pfeil auf den Zustand zurückführt, von dem er ausgeht, bedeutet dies, daß unter den zugehörigen Bedingungen der Zustand erhalten bleibt.

Im einem Diagram nach Bild 3.18 ist noch nicht festgelegt, ob ein Moore- oder ein Mealy-Automat entstehen soll. Diese Festlegung erfolgt bei der Eingabe in ein Entwurfs- und Synthesewerkzeug – solche Abläufe werden in Kapitel 5 behandelt.

Im weiteren wenden wir die bislang erarbeiteten Grundkenntnisse über Automaten an, um das Steuerwerk eines Mikrorechner-Ausschnittes mit sehr wenigen Befehlen aufzubauen. Dabei wird ersichtlich, daß ein programmierbarer Automat entsteht, der insbesondere auf Eingangssignale, die den sogenannten Operationscode (Befehlscode) darstellen, so reagiert, daß die gewünschten Befehle Schritt für Schritt ausgeführt werden.

An der Box Bild 3.18 in oben rechts, die den Automaten mit seinen äußeren Anschlüssen symbolisiert, sind neben Ein- und Ausgangssignal noch ein Taktsignal (clk) und ein Resetsignal (res) zu sehen. Diese sind stets erforderlich – die Funktion des Taktes ist bereits bekannt. Das asynchrone Resetsignal (res) sorgt beim Einschalten der Stromversorgung dafür, daß der Automat einen definierten Grundzustand – hier st0 – einnimmt. Die Bedeutung dieser Maßnahme wird in Kapitel 5 noch tiefergehend verdeutlicht

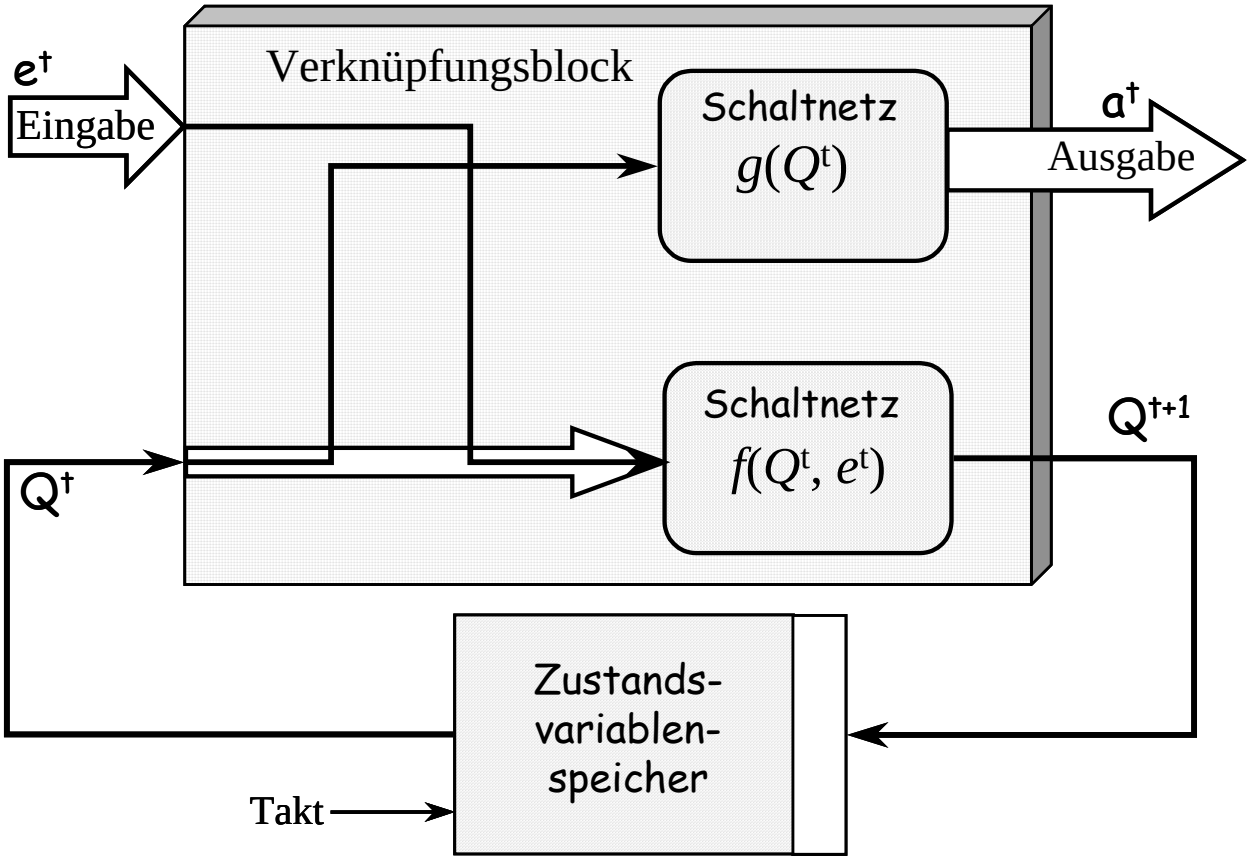



Bild 3.16: Der Moore–Automat

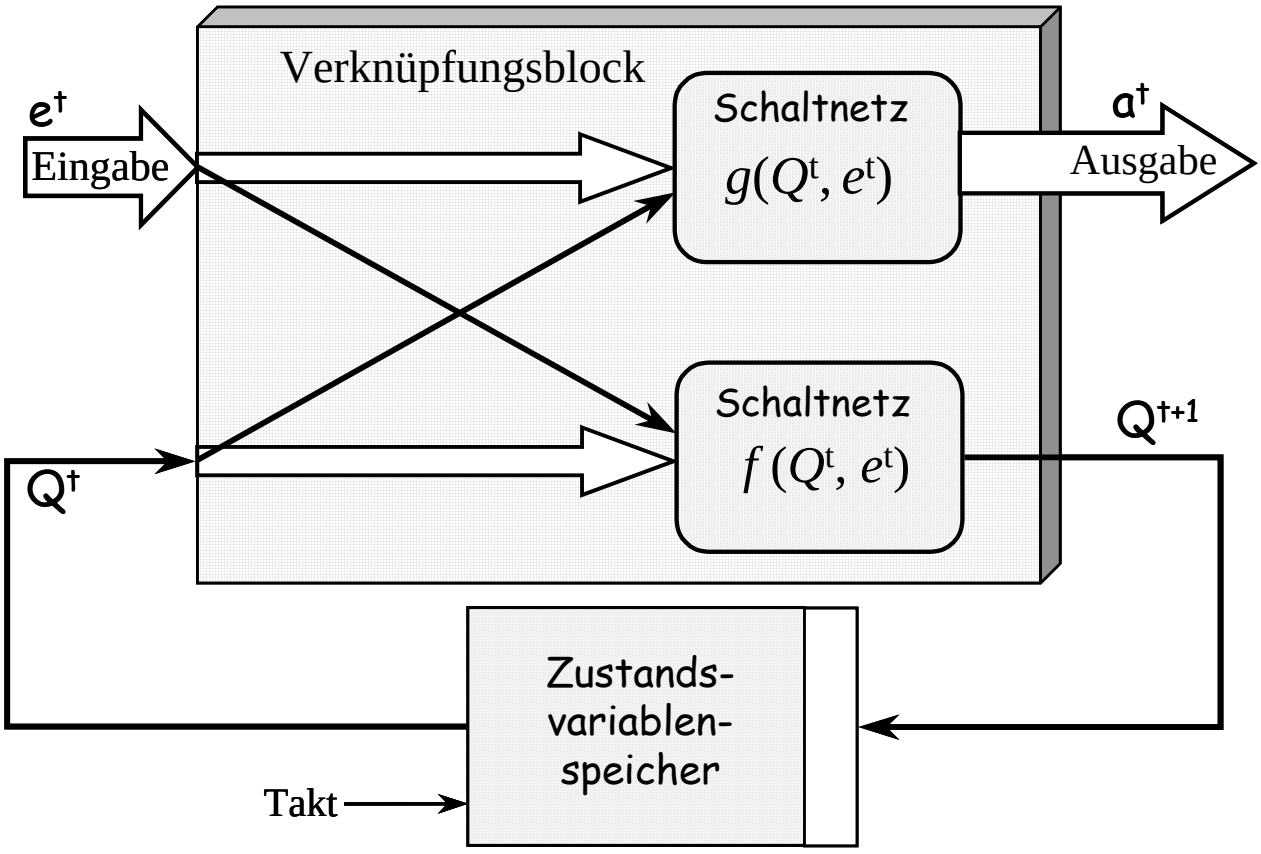



Bild 3.17: Der Mealy–Automat

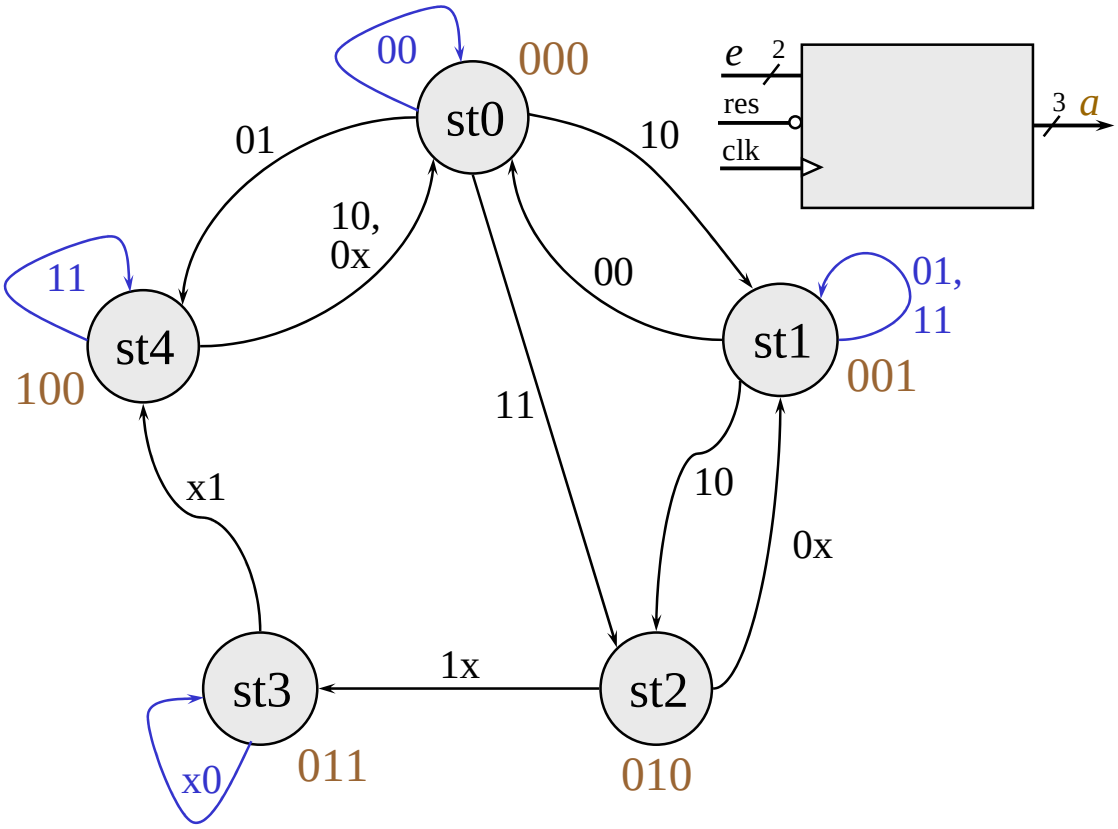



Bild 3.18: Zustands–Übergangsdiagramm für einen Automaten

# 4 Mikroprozessoren, -controller und digitale Signalprozessoren

## 4.1 Das Steuerwerk als programmierbarer Moore-Automat

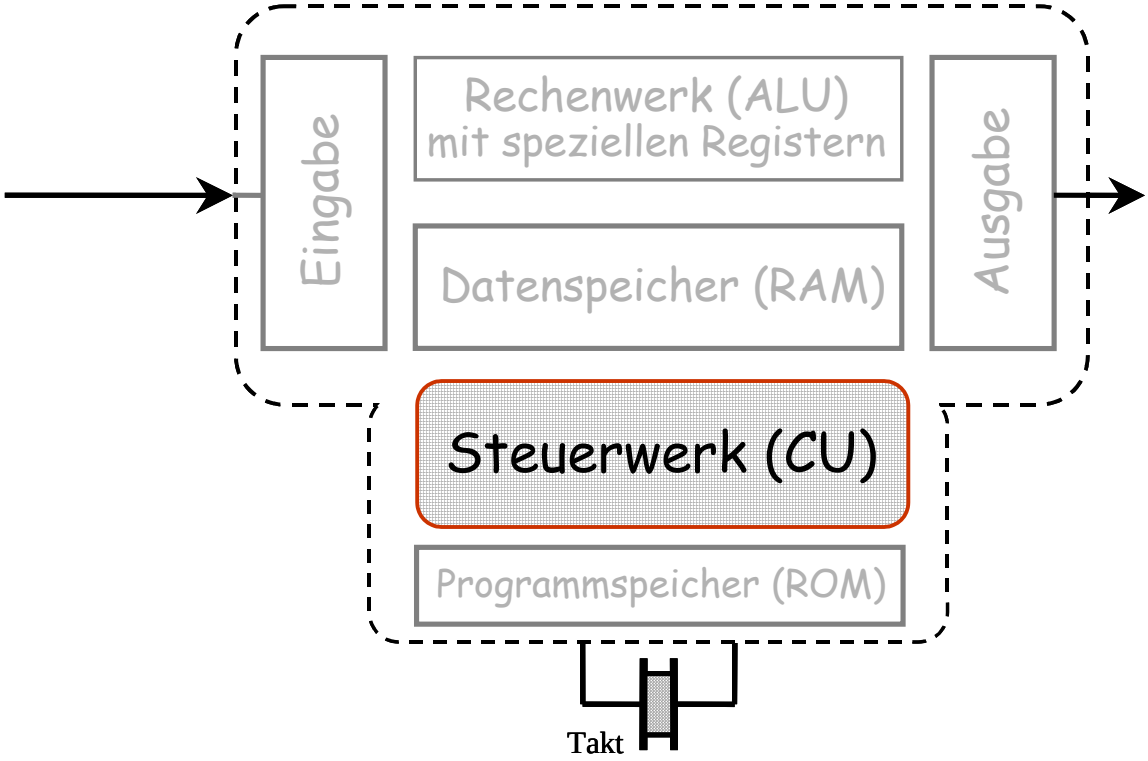



Bild 4.1: Das Steuerwerk ist ein übergeordneter Kernbestandteil eines Mikrorechners

In diesem Abschnitt werden Grundkonzepte der zeitlichen und logische Steuerung von Abläufen in Mikrorechnersystemen behandelt.

In Bild 4.1 ist das Steuerwerk (Control Unit ≡ CU) hervorgehoben. Es stellt in der Regel den komplexesten Bestandteil eines Mikrorechners dar. Der Entwurf, die hard- und softwareseitige Realisierung und die Verifikation des Steuerwerks sind sehr anspruchvolle Ingenieuraufgaben, die eine gewisse Einarbeitung in die Materie erfordern. Dieser Aufwand lohnt sich ganz allgemein, weil Ablaufsteuerungen wie wir sie hier betrachten Bestandteile vielfältiger technischer Systeme sind, d.h. sie kommen überall dort vor, wo ein Vorgang „automatisch“ abläuft. Einfache Beispiele sind Ampel- oder Fahrstuhlsteuerungen.

Zur Begrenzung der Komplexität wird hier ein überschaubarer Mirkorechnersystem-Ausschnitt betrachtet, der im weiteren als Mikrosequencer bezeichnet wird. Wie das Blockdiagram in Bild 4.2 zeigt, verfügt der Baustein über 5 binäre Eingänge: 3 für die Eingabe eines Befehlscodes (OP-Code), einen für eine Bedingung (COND) und eine für den Hardware-Reset. Eingang und Ausgang stellen *N*-bit breite Binärvektoren dar, die der „Maschinenwortbreite“ (8, 16, 32 bit) entsprechen.

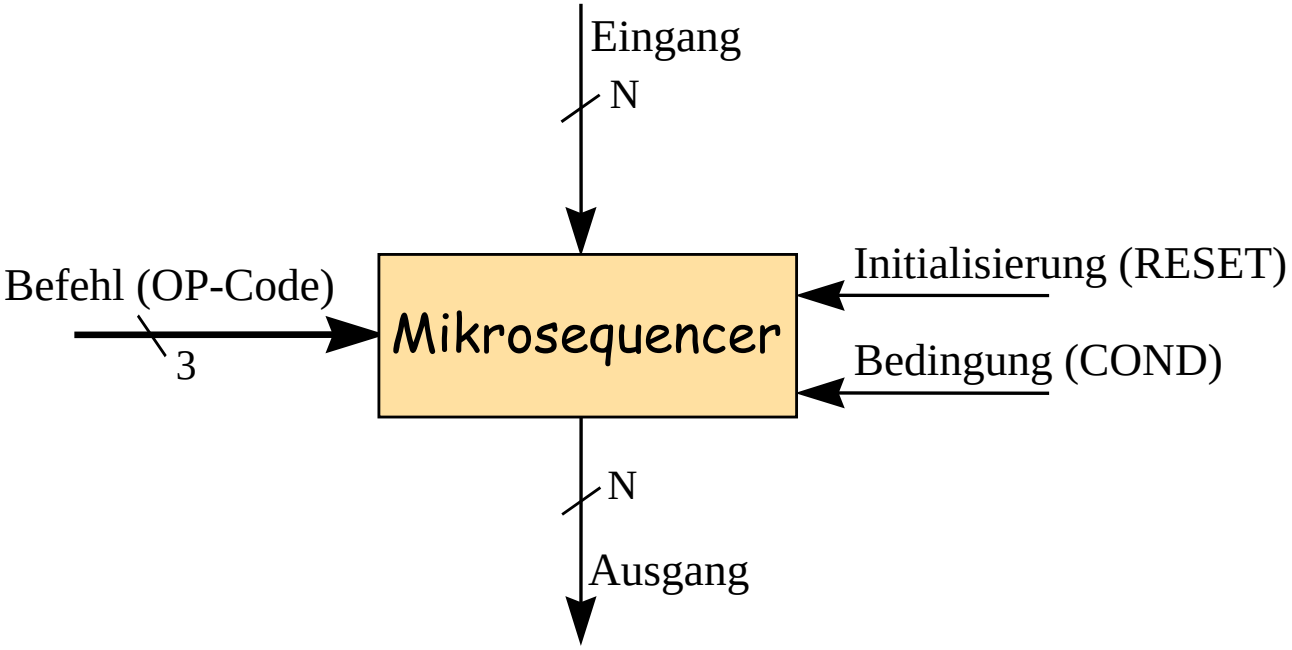



Bild 4.2: Blockschaltbild des Mikrosequencers zur „Portdefinition“

Tabelle 4.1: Liste der implementierten Befehle

| Befehl | Code | Mnemonic | Bedeutung |
|---|---|---|---|
| Reset | 000 | RES | Nullvektor an den Ausgang bringen |
| Weitermachen (continue) | 001 | CNT | Ausgang um eins erhöhen |
| Verzweigen (branch) | 010 | BRA | Ausgang:= Ausgang + Eingang |
| bedingtes Verzweigen | 011 | BRC | Ausgang:= Ausgang + Eingang wenn COND=1, sonst Ausg. +1 |
| Springen (jump) | 100 | JMP | Ausgang:= Eingang |
| bedingtes Springen | 101 | JMC | Ausgang:= Eingang, wenn COND = 1, sonst Ausg. +1 |
| Springe in Unterprogramm (Subroutine) | 110 | JMS | Ausgang:= Ausgang +1 ⇒ auf STACK, dann Ausgang:= Eingang |
| Rücksprung (return) | 111 | RET | obersten STACK-Eintrag holen und an den Ausgang bringen |

Befehlssatz nach Untersuchung der Bedingungen:
⇒ nur 6 verschiedene Befehle sind nötig

| OP-Code | COND | RESET | Mnemonic | Bemerkung |
|---|---|---|---|---|
| XXX | X | 1 | RES | RES ≡ B1 |
| 000 | X | 0 | RES | RES ≡ B1 |
| 001 | X | 0 | CNT | CNT ≡ B2 |
| 010 | X | 0 | BRA | BRA ≡ B3 |
| 011 | | | BRC | |
| 011 | 0 | 0 | CNT | CNT ≡ B2 |
| 011 | 1 | 0 | BRA | BRA ≡ B3 |
| 100 | X | 0 | JMP | JMP ≡ B4 |
| 101 | | | JMC | |
| 101 | 1 | 0 | JMP | JMP ≡ B4 |
| 101 | 0 | 0 | CNT | CNT ≡ B2 |
| 110 | X | 0 | JMS | JMS ≡ B5 |
| 111 | X | 0 | RET | RET ≡ B6 |

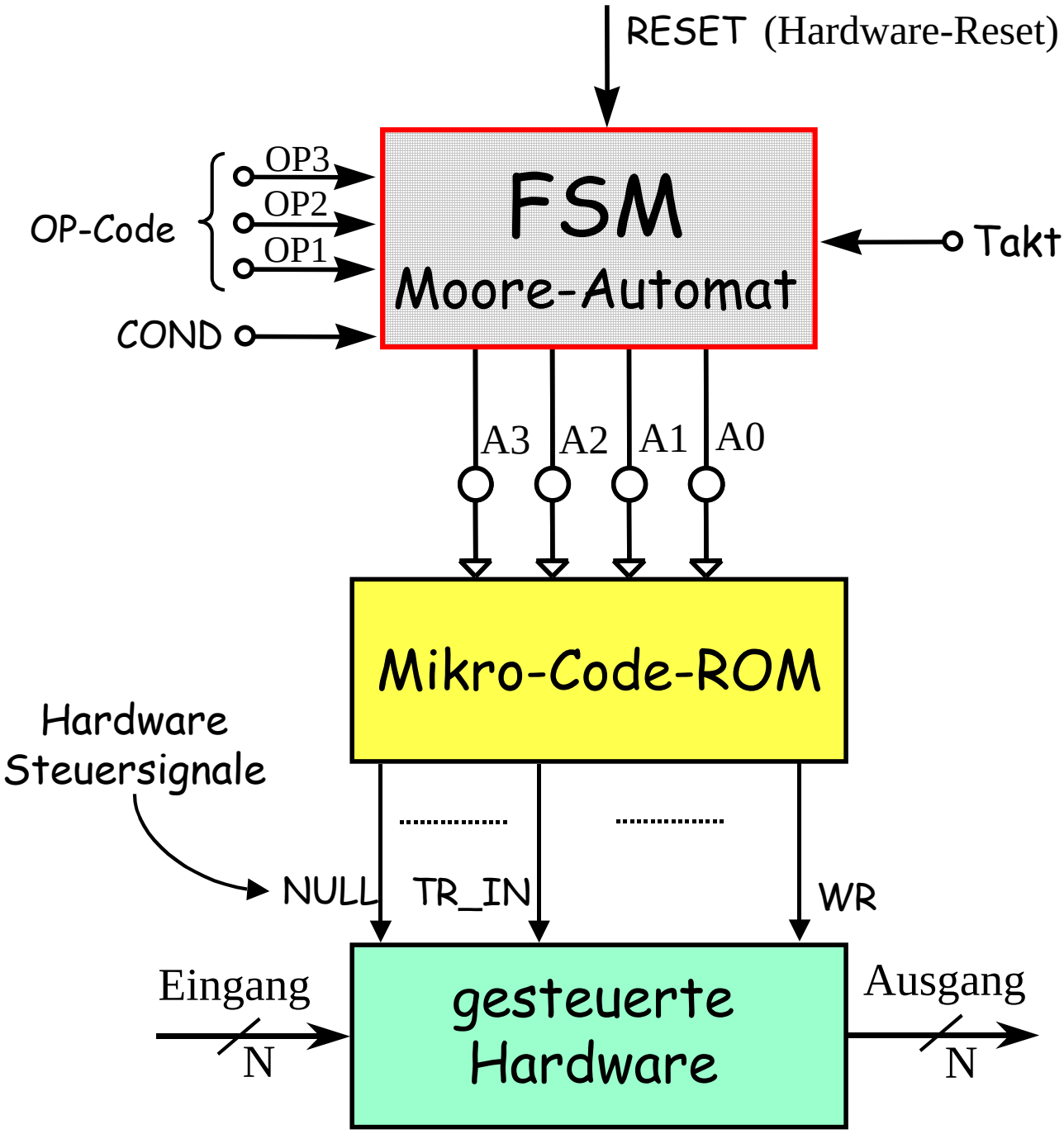



Bild 4.3: Das Mikrocode-ROM als Bindeglied zwischen FSM und gesteuerter Hardware des Mikrosequencers

## Das Mikroprogramm zur Hardwareansteuerung

In der Kopfzeile finden sich die Ansteuersignale der Hardware, die jeweils bei einem Eintrag ’1’ in der Tabelle aktiv sind

| | Null | TR_IN | TR_X | ADD | INC | S_INC | DEC | TR_Y | RD | WR |
|---|---|---|---|---|---|---|---|---|---|---|
| RES | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| CNT | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| BRA | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| JMP | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| JMS | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| RET | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |

Tabelle 4.2: Liste der Hardwaresteuersignale, die bei den Befehlen aktiv werden (Mikroprogramm)

## Der Mikroprogramm-Speicher: 13 Speicherplätze für je 10 bit

| Befehl | belegte Adressen |
|---|---|
| RES | 0000 |
| CNT | 0001<br>0010 |
| BRA | 0011<br>0100 |
| JMP | 0101 |
| JMS | 0110<br>0111<br>1000<br>1001 |
| RET | 1010<br>1011<br>1100 |

Die Hardware des Mikrosequencers in Register-Transfer-Darstellung

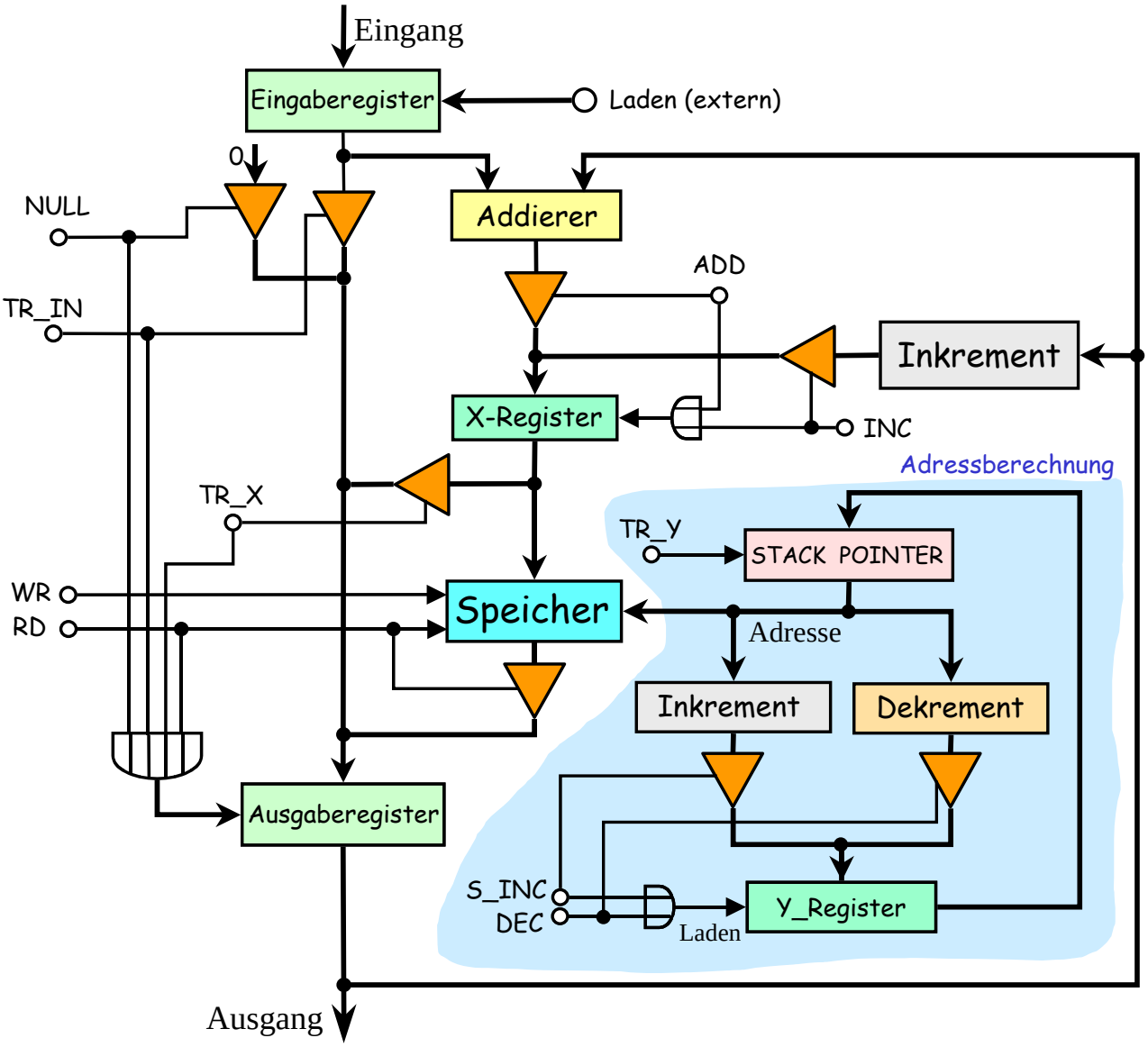



Bild 4.4: Die Mikrosequencer-Hardware in „Register-Transfer“-Beschreibung

Hier sind Register, Speicher, Addierer sowie Inkrementierer und Dekrementierer über verschiedene Signalwege miteinander verbunden, wobei die Aktivierung entsprechender Datentransporte über die Signalwege mittels Tristate-Treibern erfolgt. Aus Tabelle 4.2 geht hervor, bei welchen Befehlen und in welcher Reihenfolge die Aktivierung stattfindet.

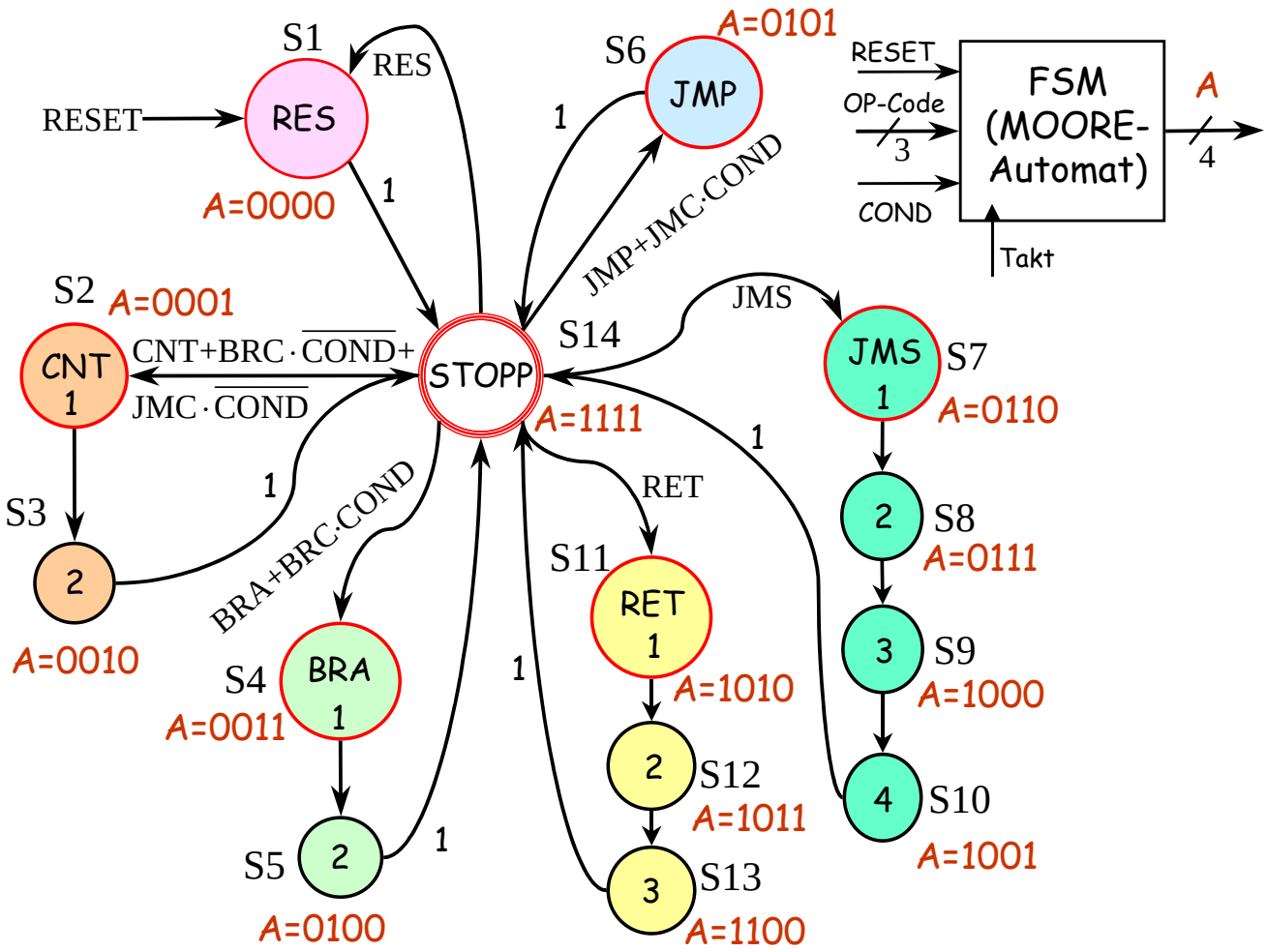



Bild 4.5: Zustandsdiagramm der FSM zur Ansteuerung des Mikro-Code-ROMs nach Bild 4.3

Beispiel einer Beschreibung der FSM zur Eingabe in ein „Synthese- und Minimierungs-Programm“, das die Programmierdaten für ein PLD (**P**rogrammable **L**ogic **D**evice) erzeugt

Hier wird die Syntax von **LOG/IC**[7] verwendet. Eine andere Möglichkeit wäre die Benutzung von **ABEL**[8] (Advanced Boolean Equation Language) oder von **VHDL**[9] (**V**ery High Speed Integrated Circuit **H**ardware **D**escription **L**anguage

[7] ein Produkt der ehemaligen Karlsruher Firma ISDATA
[8] ein Produkt der Firma Data I/O
[9] ein weltweit akzeptierter IEEE-Standard (IEEE ≡ Institute of Electrical and Electronics Engineers)

***identification** ;erläuternder Text

FSM zur Steuerung der MS-Hardware unter Verwendung des programmierbaren Logikbausteins „GAL 16V8“

Implementierte Befehle:

RES, CNT, BRA, BRC, JMP, JMC, JMS, RET
zusätzlich: RESET (Hardware-Reset), COND (Bedingung)

***declarations** ;Deklaration der benötigten Variablen
x-variables = 5 ; Eingänge
z-variables = 4 ;Ausgänge mit “Moore”-Charakter

***x-names** ;Namen der Eingangsvariablen
OP3=1, OP2=2, OP1=3, COND=4, RESET=5;

***z-names** ;Namen der Ausgangsvariablen
A3=1, A2=2, A1=3, A0=4;

***z-values** ;Definition der Zustands-Ausgabewerte

| Zustand | Ausgabewert | zugehöriger Befehl |
|---|---|---|
| s1 | 0000 | RES (RESET) |
| s2 | 0001 | CNT_1 |
| s3 | 0010 | CNT_2 |
| s4 | 0011 | BRA_1 |
| s5 | 0100 | BRA_2 |
| s6 | 0101 | JMP |
| s7 | 0110 | JMS_1 |
| s8 | 0111 | JMS_2 |
| s9 | 1000 | JMS_3 |
| s10 | 1001 | JMS_4 |
| s11 | 1010 | RET_1 |
| s12 | 1011 | RET_2 |
| s13 | 1100 | RET_3 |
| s14 | 1111 | STOPP |

***run-control** ;Ablaufkommandos mit gewünschten Ausgaben

listing = fuse-plot, pinout, equations, plot progformat = jedec

```
*flow-table                                          ;Flusstabelle => Kernstück der FSM
relevant = OP3  OP2  OP1  COND  RES.                 ;Eingangsvariablen
s14  , x  0    0    0    -     -    ,f1;   von Stopp zu Reset bei RES
s14  , x  -    -    -    -     1    ,f1;   von Stopp zu Reset bei RESET
s1   , x  -    -    -    -     0    ,f14;  nach Reset-Ausführung ⇒Stopp

s14  , x  0    0    1    -     0    ,f2 ;  CNT_1
s2   , x  -    -    -    -     0    ,f3 ;  CNT_2 fester Ablauf
s3   , x  -    -    -    -     0    ,f14;  CNT ausgeführt ⇒ Stopp

s14  , x  0    1    0    -     0    ,f4 ;  BRA-Befehl => BRA_1
s14  , x  0    1    1    1     0    ,f4 ;  BRC-Befehl => BRA_1
s4   , x  -    -    -    -     0    ,f5 ;  BRA_2
s5   , x  -    -    -    -     0    ,f14;  BRA ausgeführt ⇒ Stopp
s14  , x  0    1    1    0     0    ,f2 ;  bei COND=0 ⇒ CNT ausführen

s14  , x  1    0    0    -     0    ,f6 ;  JMP
s6   , x  -    -    -    -     0    ,f14;  JMP-Befehl ausgeführt ⇒ Stopp

s14  , x  1    0    1    1     0    ,f6 ;  JMC-Befehl mit COND=1
s6   , x  -    -    -    -     0    ,f14;  JMP-Befehl ausgeführt ⇒ Stopp
s14  , x  1    0    1    0     0    ,f2 ;  bei COND=0 ⇒ CNT ausführen

s14  , x  1    1    0    -     0    ,f7 ;  JMS_1 (Unterprogramm)
s7   , x  -    -    -    -     0    ,f8 ;  JMS_2
s8   , x  -    -    -    -     0    ,f9 ;  JMS_3
s9   , x  -    -    -    -     0    ,f10;  JMS_4
s10  , x  -    -    -    -     0    ,f14;  JMS-Befehl ausgeführt

s14  , x  1    1    1    -     0    ,f11;  RET_1
s11  , x  -    -    -    -     0    ,f12;  RET_2
s12  , x  -    -    -    -     0    ,f13;  RET_3
s13  , x  -    -    -    -     0    ,f14;  RET-Befehl ausgeführt

s[1..14]   , xrest ,f14   ;alle restlichen x-Eingaben führen zu Stopp!

*pal                                 ;zu verwendende Hardware (Bausteintyp)
type =GAL16V8
*pins                                ;Anschlussbelegung
OP1=2, OP2=3, OP3=4, COND=5, RESET=6, A3=19, A2=18, A1=17, A0=16;
*flipflops                           ;Art der zu verwendenden Flip-Flops
d-flipflops
*end
```

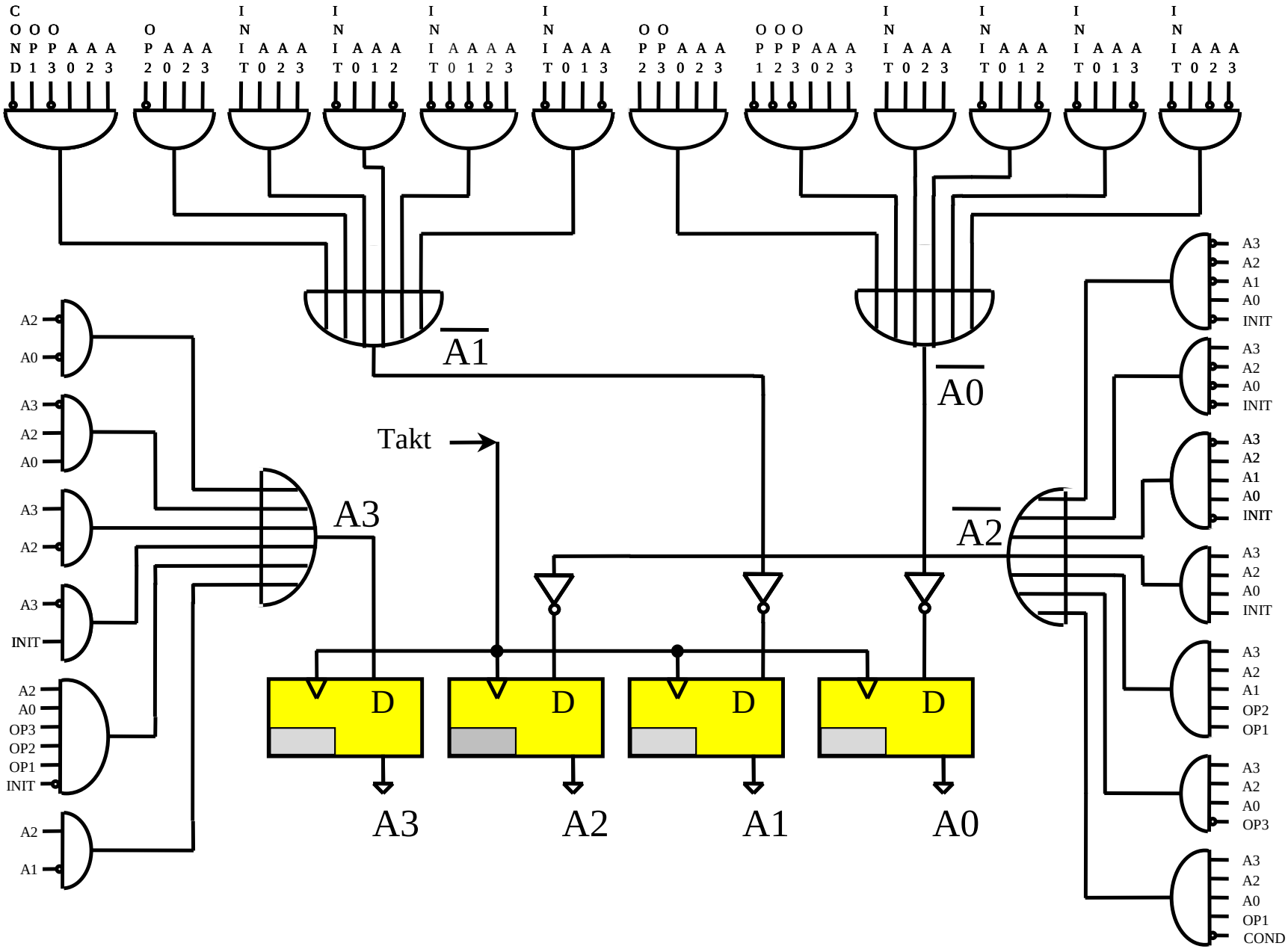



Bild 4.6: Die gesamte Hardware der FSM (Moore-Automat)

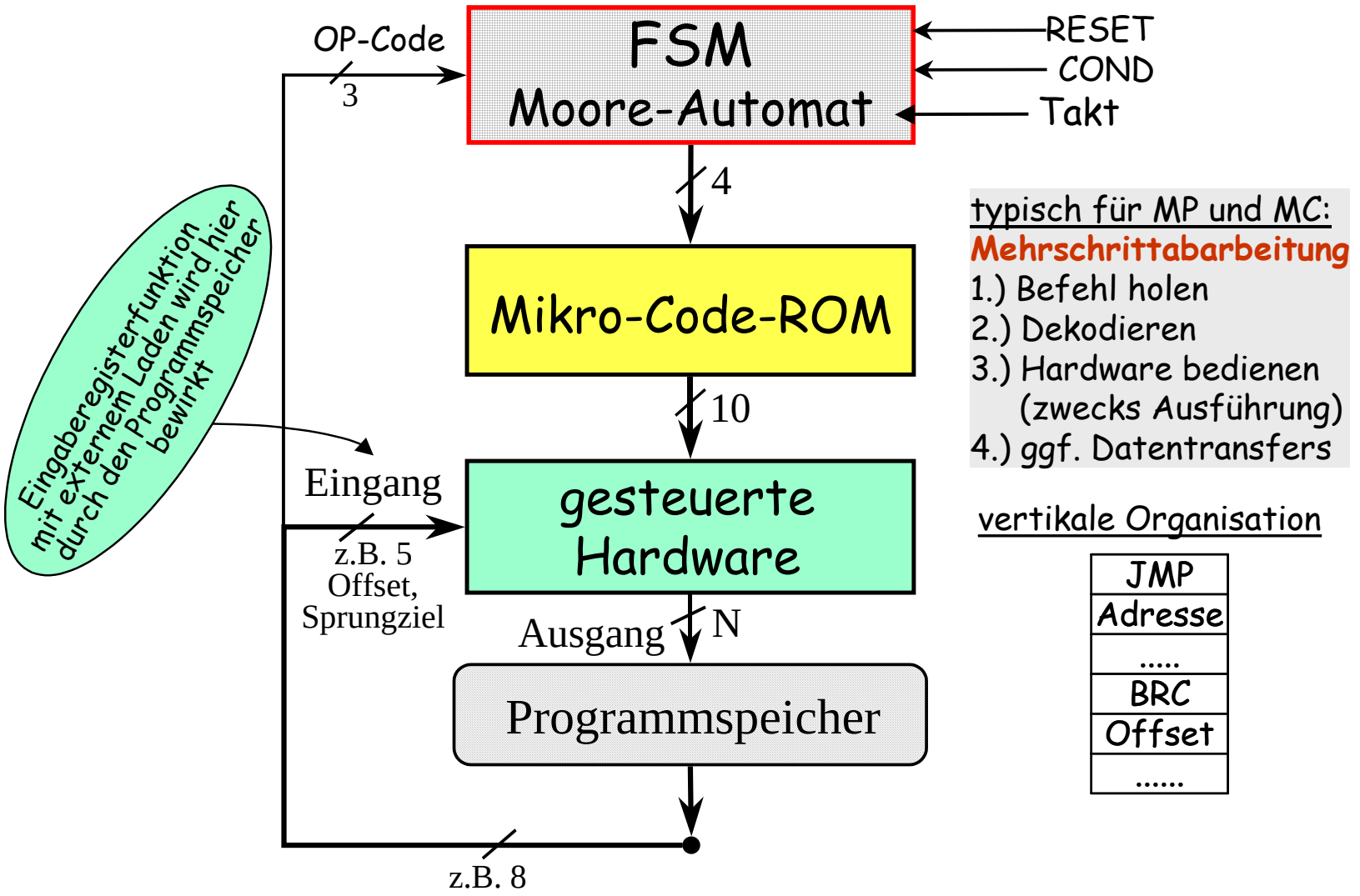



Bild 4.7: Einbindung des Mikrosequencers in ein Mikrorechnersystem

## 4.2 Realisierung arithmetischer und boolescher Oparationen - die Mikrorechner-ALU

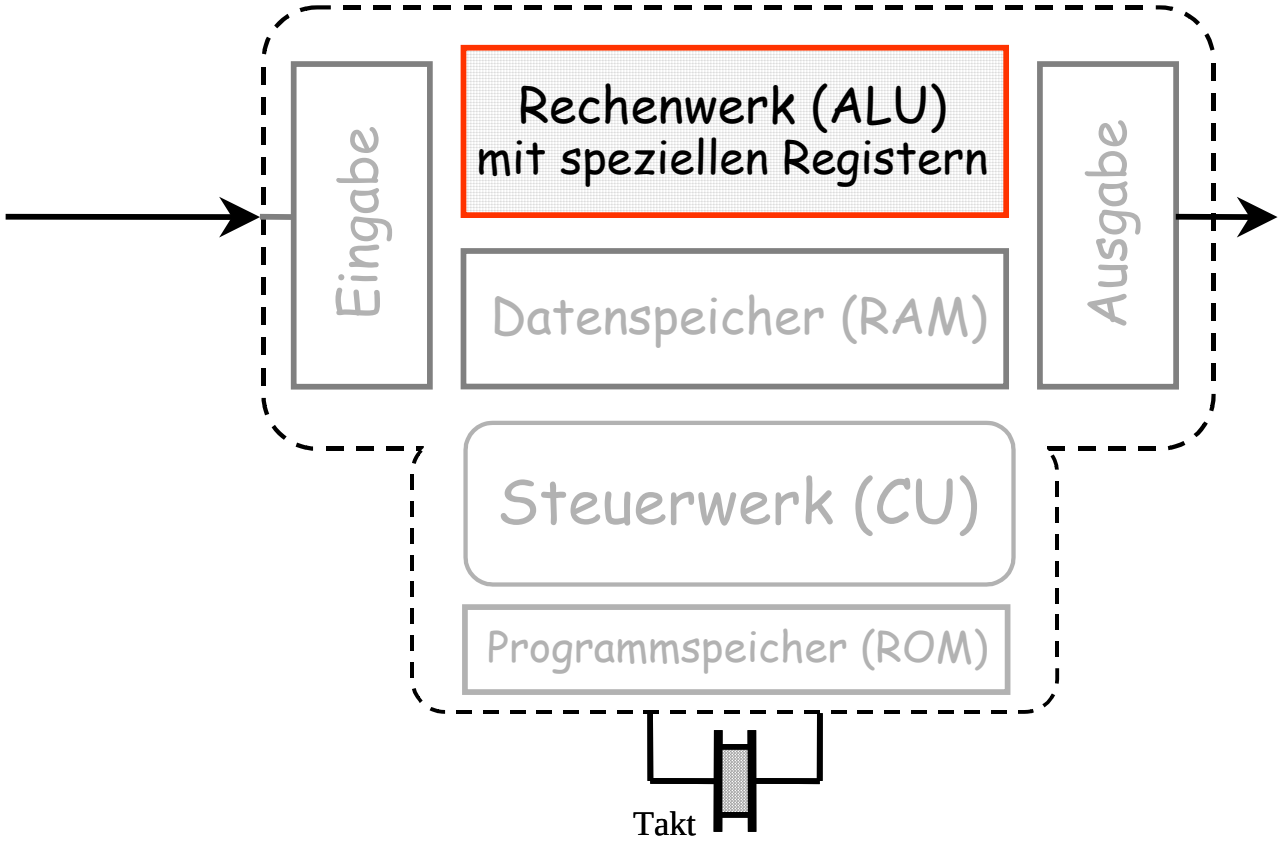



Bild 4.8: Die arithmetisch-logische Einheit (ALU) als weiteres Kernelement von Mikrorechnern

### 4.2.1 Boolesche Algebra, Logikpegel und Zahlenformate in der Mikrorechnertechnik

Die Schaltalgebra behandelt das Aufstellen logischer Verknüpfungen und ermöglicht so Analyse und Synthese logischer Schaltungen. Die Schaltalgebra ist aus der zweiwertigen (binären) Aussagelogik entstanden und erfaßt nur die logischen Funktionen und macht *keine* Aussagen über z.B.

- Technische Realisierung und Aufbau
- Arbeitsgeschwindigkeit
- Signallaufzeiten
- Spannungstoleranzen, Leistungsaufnahme, eventuelle EMV-Probleme
- Grenzdaten (maximale Taktfrequenz, Temperaturbereich)

Alle logischen Verknüpfungen, die die Schaltalgebra beschreibt, sind mit 3 Grundfunktionen realisierbar:

| | Symbole |
|---|---|
| ▪ UND | &, Λ, · |
| ▪ ODER | 1, ∀, + |
| ▪ NEGATION | $\overline{a}$ |

Die Realisierung dieser Grundfunktionen oder einfachen Kombination wie z.B.

- NAND aus UND mit nachfolgender NEGATION

bzw.

- NOR aus ODER mit nachfolgender NEGATION

mit elektronischen Bauelementen wird im weiteren in den heute gebräuchlichen verschieden Technologien behandelt. Andere Arten der Realisierung (mechanisch, elektromechanisch, pneumatisch) spielen heutzutage keine Rolle mehr.

In der elektronischen Digitaltechnik ordnet man den binären Zuständen

- „0“ oder LOW („L“)
- „1“ oder HIGH („H“)

elektrische Größen, d.h. Spannungen, Ströme oder auch Feldstärken zu, die möglichst gut und einfach unterscheidbar sein müssen. Zwischen den jeweiligen L- und H-Werten liegt ein *verbotener Bereich*, in dem weder ein Eingangssignal noch ein Ausgangssignal liegen darf.

Tabelle 4.3: Pegeldefinitionen für einige Beispiele digitaler Schaltkreistechnologien

| Spannung | $U_B$ | $U_{OH}$ | $U_{OL}$ | $U_{IH}$ | $U_{IL}$ | $U_{SCHW.}$ | verboten |
|---|---|---|---|---|---|---|---|
| TTL | 4,5...5,5V | 2,4V | 0,5V | 2,0V | 0,8V | 1,5V | 0,8…2V |
| CMOS | 4,5...5,5V | $U_B$-0,1V | 0,1V | 2,0V | 0,9V | ca. 2,5V | 0,9…2V |
| LVCMOS | 2,7...3,6 | $U_B$-0,1V | 0,1V | 0,7·$U_B$ | 0,2·$U_B$ | 0,5·$U_B$ | 0,2...0,7$U_B$ |
| LVTTL | 3,0...3,6V | 2,4V | 0,4V | 2,0V | 0,8V | 1,5V | 0,8...2V |
| ECL | -5,2V | -0,67V | -1,6V | -0,85V | -1,4V | -1,3V | -1,4...-0,85V |

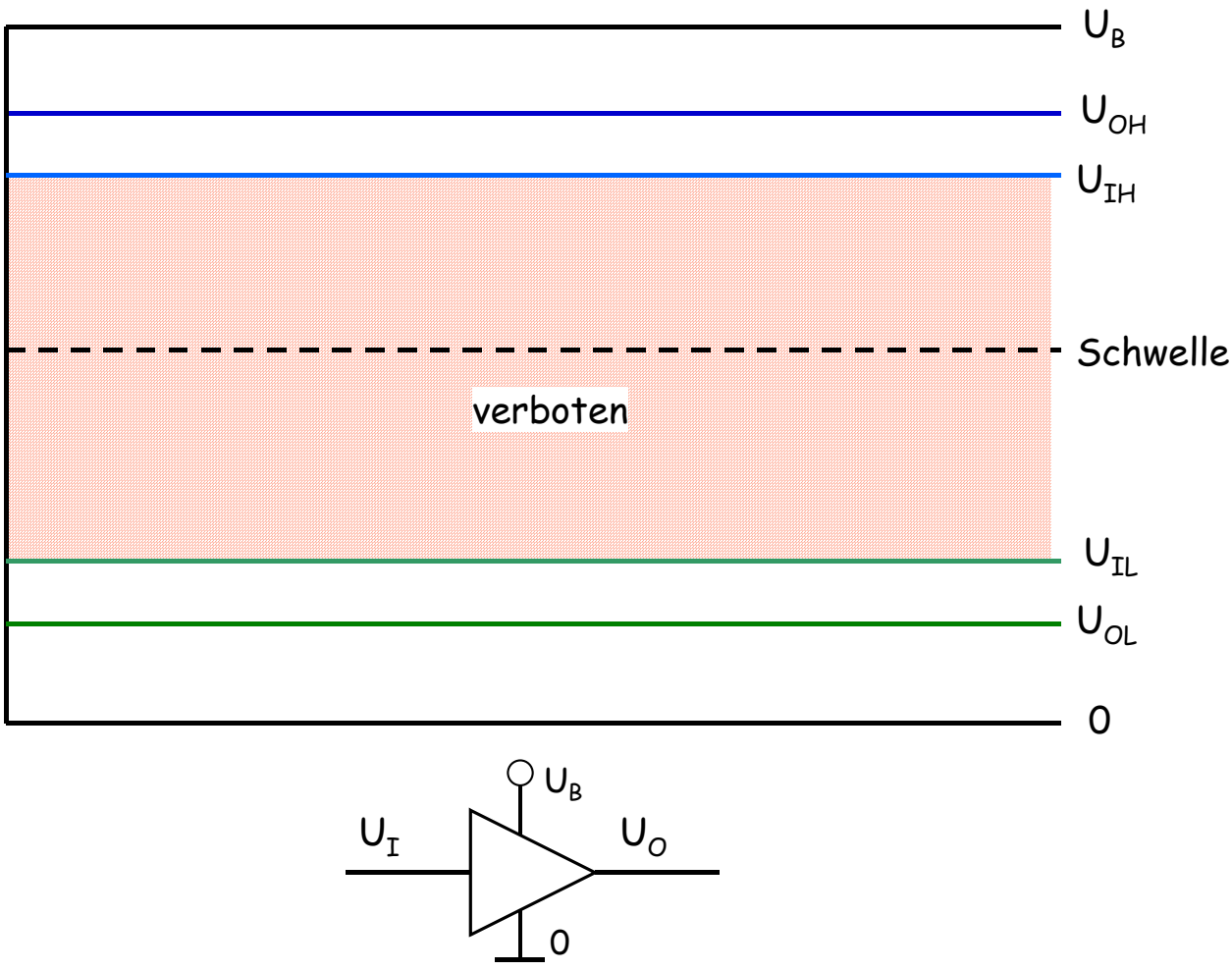



Bild 4.9: Veranschaulichung der Pegelbereichsdefinitionen digitaler Schaltungen

### 4.2.2 Zahlendarstellung und wichtige Rechenoperationen

Die allgemeine „polyadische“ Zahlendarstellung gemäß (3.1) beruht auf einer Zahlenbasis $\alpha$, wobei $\alpha$ eine beliebige ganze positive Zahl ist. Nicht jeder mögliche Wert von $\alpha$ hat praktische Bedeutung, sondern nur relativ wenige ausgezeichnete. In digitalen Rechenmaschinen sind dies $\alpha$=2, 8, 16. Die allgemein gebräuchliche und sozusagen dem Menschen angepaßte Zahlenbasis $\alpha$=10, d.h. das Dezimalsystem. Die Koeffizienten $x_i$ in (3.1) sind ebenfalls ganze Zahlen und stets auf den Wertebereich {0...$\alpha$–1} begrenzt.

$$Z_\alpha = \sum_{i=0}^{N-1} x_i \cdot \alpha^i = x_0 \cdot \alpha^0 + x_1 \cdot \alpha^1 + x_2 \cdot \alpha^2 + ... + x_{N-1} \cdot \alpha^{N-1} \tag{4.1}$$

$\alpha$=2 hat aufgrund der einfachen technischen Realisierbarkeit die größte Bedeutung innerhalb digitaler Rechner erlangt. Man spricht vom Dualsystem oder binärer Zahlendarstellung, weil nur zwei Zustände „0“ und „1“ unterschieden werden müssen. Oktal- und Hexadezimaldarstellung sind nicht wirklich in den Maschinen realisiert, sondern werden meist zur Abkürzung und Vereinfachung der Schreibweise benutzt.

Der gesamten heutigen Mikrorechnertechnik liegt somit die folgende Zahlendarstellung zu Grunde:

$$Z_b = \sum_{i=0}^{N-1} x_i \cdot 2^i, \text{ mit } x_i \in \{0,1\} \tag{4.2}$$

$Z_b$ nach (4.2) ist eine N–stellige Binärzahl (Index b), deren Dezimalwert sich wie folgt berechnet:

$$Z_b = x_0 \cdot 2^0 + x_1 \cdot 2^1 + x_2 \cdot 2^2 + ... + x_{N-1} \cdot 2^{N-1}, \quad \text{mit } x_i \in \{0,1\} \tag{4.3}$$

Für die Durchführung von Rechenoperationen genügt die Darstellung nach (4.2) nicht. Es wäre z.B. nur das Arbeiten mit positiven Zahlen möglich. Ein erster Erweitungsschritt ist notwendig, um auch negative Zahlen zu erfassen. In diesem Zusammenhang hat die sogenannte Zweierkomplementdarstellung die größte praktische Bedeutung. Zur Herleitung gehen wir von einer N–stelligen Binärzahl aus, die wir in Form eines N–bit breiten Binärvektors (Zeilenvektor) darstellen:

$$\mathbf{p} = \sum_{i=0}^{N-1} p_i \cdot 2^i \tag{4.4}$$

Wenn wir nun formal den Vektor **p** mit seinem Komplement $\overline{\mathbf{p}}$ ODER–verknüpfen, ergibt sich stets der Einsvektor (1111....1), d.h. bei einer N–stelligen Zahl gilt immer:

$$\mathbf{p} + \overline{\mathbf{p}} = 2^N - 1 \tag{4.5}$$

Ein einfaches Beispiel macht klar, daß die ODER–Verknüpfung hier mit der arithmetischen Addition identisch ist:

| | |
|---|---|
| **p** | 10011101011001 |
| $\overline{\mathbf{p}}$ | 01100010100110 |
| $\mathbf{p}+\overline{\mathbf{p}}$ | 11111111111111 |

Beim Addieren gibt es – wie man leicht sieht – niemals einen Übertrag in die nächsthöhere Stelle. Jetzt kann man rein formal arithmetisch weiterrechen und erhält aus (4.5):

$$-\mathbf{p} - \overline{\mathbf{p}} = -2^N + 1, \text{ oder } -\mathbf{p} = -2^N + \overline{\mathbf{p}} + 1 \tag{4.6}$$

Die negative Zahl **–p** ergibt sich also durch Komplementieren (umkehren) aller Bits von **p** und Addition von Eins. Das Ergebnis hat den Namen Zweierkomplement, während das Umkehren der ein-

zelnen Bits als Einerkomplementbildung bezeichnet wird. Aus (4.6) wird deutlich, daß das Zweierkomplement keineswegs der negativen Zahl **–p** entspricht, sondern erst dann, wenn $-2^N$ dazukommt ist das Ergebnis arithmetisch richtig. Daraus folgt, daß eine negative Zahl immer durch negative Gewichtung ihrer höchsten Potenz gekennzeichnet ist. Ist dagegen diese höchstwertige Stelle Null, liegt eine positive Zahl vor.

Tabelle 4.4: Einführungsbeispiel zur Zweierkomplementarithmetik

| | | | |
|---|---|---|---|
| 9 | 1001 | binäre Addition | |
| 7 | 0111 | 9 | 1001 |
| –7 | $1000+1-2^4$ | –7 | $-2^4+1001$ |
| | | Summe: | $+2^4-2^4+0010$ |
| | | 9–7=2 | 0010 |

Das Beispiel macht deutlich, daß es unbedingt erforderlich ist, die höchstwertige Stelle, d.h. $2^N$ mitzunotieren, denn sonst könnte man –7 nicht von +9 unterscheiden. Aus einer N–stelligen Zahl wird damit aber eine (N+1)–stellige. Allgemein gilt daher für eine Binärzahl im Zweierkomplement:

$$Z_\mathrm{b} = -x_N \cdot 2^N + \sum_{i=0}^{N-1} x_i \cdot 2^i \qquad (4.7)$$

Wie man aus (4.7) leicht erkennt, hat man für $x_N$=0 immer eine positive Zahl und für $x_N$=1 eine negative vor sich. Auf einen Mikrorechner mit z.B. 8bit Wortlänge bezogen bedeutet dies, daß der Zahlenbereich {0...255} für positive ganze Zahlen bei Anwendung der Zweierkomplementdarstellung zu {-128...+127} verschoben wird. Somit muß jetzt auch für den Wert +127 die gesamte Wortlänge von 8 bit benutzt werden, obwohl bei rein positiver Interpretation 7 bit genügen würden. An dieser Stelle wird eine wichtige Tatsache deutlich: Die Hardware einer digitalen Recheneinrichtung kann nicht zwischen positiven und negativen Zahlen unterscheiden. Das Kernstück der ALU (Arithmetic Logic Unit) einfacher Mikrorechner ist ein binärer Addierer, der stets in gleicher Weise die beiden zu verarbeitenden Bitvektoren verknüpft, d.h. er addiert stets Bit für Bit binär und gibt entstehende Überträge weiter. Es ist Aufgabe des Programmierers bzw. der Software die Zahlendarstellung korrekt zu wählen und die Ergebnisse richtig zu interpretieren. Bei vorzeichenbehafteten Rechenoperationen können zunächst durchaus fehlerhafte Resultate im Addierer entstehen, die erkannt und passend korrigiert werden müssen. Dazu werden geeignete Marker (Flags) generiert, von denen wir diejenigen, die zur korrekten Handhabung von Zweierkomplementzahlen nötig sind, nämlich Carry- (C), Negativ- (N) und Overflow-Flag (OV) jetzt im Detail betrachten. Daneben verfügt eine Mikrorechner–ALU meist noch über ein Auxiliary-Carry- (AC) und ein Parity-Flag - s. Bild 4.10.

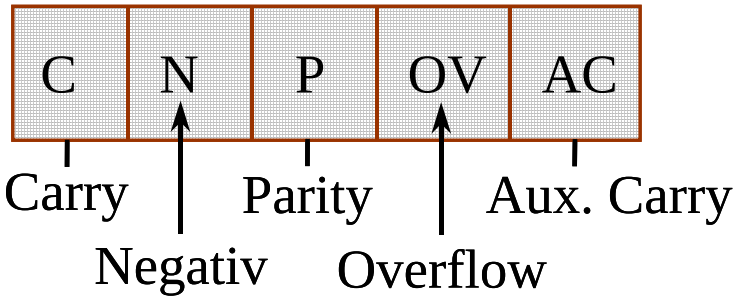



Bild 4.10: Typische Flags in einer Mikrorechner–ALU

Das Carry–Flag wird z.B. gesetzt, wenn bei einer Addition ein Überlauf über die Wortlänge der Maschine hinaus entsteht, d.h. bei einer 8 bit–ALU ins 9. Bit. Das AC–Flag ist eine Hilfsgröße, um mit vierstelligen Binärzahlen (sogenannten BCD[10]-Zahlen) zu rechnen.

[10] Binary Coded Decimals

Es wird immer dann gesetzt, wenn ein Übertrag von der vierten in die fünfte Stelle auftritt. Beim Parity–Flag unterscheidet man zwischen gerader und ungerader Parität. Es dient zum Erkennen von Fehlern bei der Datenübertragung, z.B. über eine integrierte serielle Schnittstelle eines Mikrorechners. Bei gerader Parität wird P stets so gesetzt, daß die Summe der „1"-Bits im Akkumulator zusammen mit P eine gerade Zahl ergibt. Entsprechend muß bei ungerader Parität die Summe der „1"-Bits stets ungerade werden. Auch beim Programmieren von Codier– und Decodierroutinen für die Datensicherung ist das Parity–Flag nützlich. Hier werden jetzt die drei für die Zweierkomplementarithmetik (ZKA) wichtigen Flags C, N und OV genauer betrachtet.

Sinn und Funktion des Negativ-Flags sind sehr einfach zu verstehen: Das N-Flag wird immer dann gesetzt (es nimmt den Wert logisch „1" an) wenn das höchstwertige Bit des Akkumulators oder ggf. auch eines anderen Registers der ALU ebenfalls „1" ist. Entsprechend (4.7) bedeutet dieses ja, daß in Zweierkomplementinterpretation eine negative Zahl vorliegt – daher der Name. Ein einfaches Beispiel zeigt, daß ein N-Flag aber keineswegs genügt, um korrekte ZKA zu betreiben: Wenn z.B. in einer 8 bit-ALU zu der Zahl $127_d$ Eins addiert wird, ergibt sich binär 1000 0000. Das Ergebnis muß natürlich +128 sein, aber in der ZKA wird es jetzt als –128 (1000 0000) angesehen. Offenbar genügt die Wortlänge von 8bit (1Byte) nicht mehr, um +128 darzustellen. Eine Lösung wäre das Anfügen eines weiteren Bytes, so daß eine 16bit-Zahl ($0000\ 0000\ 1000\ 0000_b = 00\ 80_h$) entsteht. Die Auswertung gemäß (4.7) liefert jetzt das korrekte Ergebnis +128. Im Grunde hätte es auch schon genügt, nur ein einziges Bit hinzuzufügen, um +128 korrekt darzustellen. Andererseits können aber ganz offensichtlich beliebig viele Null-Bits vorangestellt werden, ohne das der Wert sich verändert. Wir werden sehen, daß das im Falle negativer Zahlen genauso für vorangestellte „1"–Bits gilt, was aber nicht so unmittelbar einsichtig scheint. Man nennt dieses „Phänomen" Vorzeichenerweiterung (engl.: sign extension) und macht bei verschiedenen Algorithmen der DSV gerne Gebrauch davon. Weil „sign extension" praktisch wichtig ist, machen wir uns den Sachverhalt auch für negative Zahlen klar:

Ähnlich wie beim obigen Additionsbeispiel würde das Subtrahieren von Eins von der Zahl –128 in der ALU eines 8bit-Mikrorechners so ablaufen:

1000 0000 –1 = 0111 1111,

d.h. das Ergebnis wäre +127 und nicht –129. Auch hier gilt wieder, daß das Ergebnis in ZK-Darstellung nicht mehr in 1Byte paßt. Es muß auch hier mindestens ein weiteres Bit hinzugefügt werden, so daß man 1 0111 1111 erhält. Jetzt liefert die Auswertung nach (4.7) das korrekte Ergebnis –256+127=–129. Eine Aufstellung analog zu Tabelle 4.4 liefert:

Tabelle 4.5: Zur Vorzeichenerweiterung bei negativen Zahlen

| | | | |
|---|---|---|---|
| $-128_d$ | 1000 0000 | binäre Addition | |
| $-1_d$ | 1111 1111 | $-128_d$ | 1000 0000 |
| $-1_d$ | $-2^7+127_d$ | $-1_d$ | 1111 1111 |
| | | Summe: $-256_d+127_d$ | 1 0111 1111 |
| | | $= -129_d$ | Carry wird gesetzt |

Es ist unmittelbar einzusehen, daß das Hinzufügen weiterer führender „1"Bits den Zahlenwert nicht verändert, denn im Beispiel nach Tabelle 4.5 ergibt 11 0111 1111 ≡ –512+256+127=–129, oder 111 0111 1111 ≡ –1024+512+256+127=–129.

Über das Beispiel hinaus wird bei Betrachtung von (4.7) allgemein klar, daß auch eine beliebige Anzahl führender „1"-Bits den Zahlenwert nicht verändert, denn jedes neue Bit liefert den Beitrag $\mathbf{-2^{N+1}+2^N = -2^N-2^N+2^N = -2^N}$, d.h. es ergibt sich keine Veränderung.

Wie wir gesehen haben, sind weder Negativ- noch Carry Flag, und auch nicht beide zusammen in der Lage, alle bei ZKA auftretende Fehlerfälle in den Griff zu bekommen. Es wird ein weiteres, nämlich das Overflow-Flag benötigt. Anstatt mit OV wird es manchmal auch nur mit V abgekürzt. Das OV-Flag darf – trotz der Namensähnlichkeit – auf keinen Fall mit dem Carry–Flag (C) verwechselt werden. Wenn OV nach einem binären Addiervorgang (nichts anderes leistet die ALU) gesetzt ist, muß das Ergebnis korrigiert werden, da es mit der Standard-Wortlänge der Maschine nicht mehr als Zweierkomplementzahl darstellbar ist. Die folgende Tabelle 4.6 listet die möglichen Konstellation von Summanden einer binären Addition auf und die Konsequenzen, die sich in den drei Flags C, OV und N widerspiegeln.

Tabelle 4.6: Fallunterscheidung bei der binären Addition und die zugehörigen Flags

| Summanden | nach Addition | | Bemerkung |
|---|---|---|---|
| **+ +** | **Carry ist immer 0** | | **Fall I** |
| | **wenn $N^+$=0 ⇒ OV=0** | **wenn $N^+$=1⇒ OV=1 ⇒ Korrektur nötig** | |
| **– –** | **Carry ist immer 1** | | **Fall II** |
| | **wenn $N^+$=1 ⇒ OV=0** | **wenn $N^+$=0 ⇒ OV=1 ⇒ Korrektur nötig** | |
| **+ –** | **C=1 ⇒ Summe positiv** | **immer OV=0** | **Fall III** |
| | **C=0 ⇒ Summe negativ** | | **stets korrekt** |

Wie man sieht, ist der Fall gemischter Operanden (+ –) problemlos, da die binäre Addition stets korrekte Ergebnisse im Sinn der ZKA liefert. OV bleibt immer „0“ und das Carry-Flag signalisiert, ob das Ergebnis positiv oder negativ ist. Es folgen einige Beispiele:

| | | | |
|---|---|---|---|
| 10000000 = –128 | 11111111 = –1 | 10000000 = –128 | 10000011 =–125 |
| 01111111 = +127 | 00000001 = +1 | 00000001 = +1 | 01111110 = +126 |
| 1111238 = –1 (C=0) | 00000000 = 0 (C=1) | 2 = –127 (C=0) | 00000001 =+1 (C=1) |

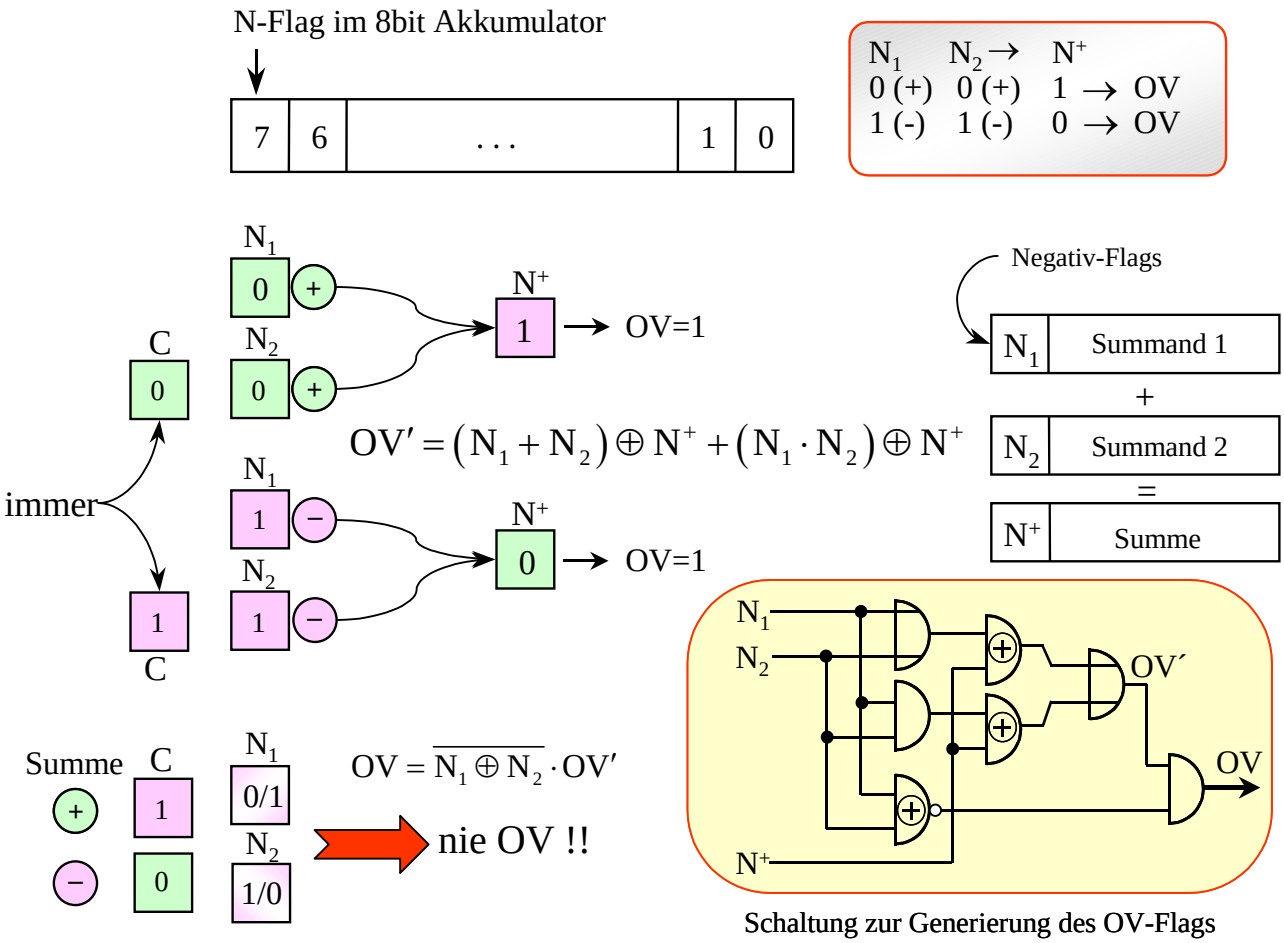



Bild 4.11: Zur Entstehung des Overflow-Flags

Es ist immer wieder wichtig, sich vor Augen zu halten, daß die ALU den Zahlen nicht die Bedeutung der einzelnen Bits „ansieht“, insbesondere nicht, daß die höchstwertigen negativ sind, d.h. sie addiert stur binäre positive Zahlen, also z.B. für die letzte Spalte der obigen Tabelle 131+126=257 gerechnet wird. Dadurch, daß der Übertrag von 256 sozusagen „unter den Tisch fällt“, also implizit abgezogen wird, erhält man das korrekte Ergebnis von 1 im Akkumulator. Ebenso wird für die zweite Spalte 255+1=256 gerechnet, was durch Wegfall des Übertrags korrekt auf zu Null führt. Die beiden weiteren Fälle I und II nach Tabelle 4.6 sind etwas kniffliger und werden jetzt untersucht.

Bild 4.11 zeigt die Zusammenhänge in einer Übersicht. Aus Tabelle 4.6 ist bereits klar, daß immer dann eine Korrektur erforderlich wird, wenn bei positiven Summanden (d.h. $N_1$ =0 UND $N_2$ =0) nach der Addition das Negativ-Flag $N^+$ gesetzt ist. Ebenso bei negativen Summanden, wenn nach der Addition $N^+$=0 geworden ist.

Die Korrektur kann im einfachsten Fall durch Anfügen eines Bits als neues MSB erfolgen, wobei unmittelbar der Wert des Carry-Bits übernommen werden kann – s. Bild 4.11. Wenn ein ganzes Byte angefügt wird, erhalten alle Bits darin den gleichen Wert. Wie man sieht, ist bei positiven Summanden stets C=0, so daß in diesem Fall bei OV=1 ein Null-Bit oder $00_h$ als Byte voranzustellen wäre. Bei negativen Summanden ergibt sich immer C=1, so daß hier bei OV=1 ein Eins-Bit oder $FF_h$ als Byte vorangestellt werden muß.

Des weiteren sind in Bild 4.11 logische Verknüpfungen zur Generierung des OV-Flags angegeben. Das Zwischenergebnis OV’ ergibt sich als EXOR–Verknüpfung von $N^+$ mit den UND– bzw. ODER–Verknüpfungen der Negativ-Flags $N_1$ und $N_2$, die vor der Addition vorhanden sind. OV’

darf aber nicht unmittelbar genommen werden, da eine einfache Überlegung zeigt, daß bei gemischten Summanden OV' immer eins wird, obwohl es hier nie etwas zu korrigieren gibt:

$$(1+0)\oplus N^{+}+(0)\oplus N^{+}=\overline{N^{+}}+N^{+}=1$$

Dieser Fall muß somit ausgeschlossen werden, was in einfacher Weise z.B. mit einem Äquivalenzgatter und einem UND-Gatter - wie unten rechts in Bild 4.11 dargestellt - geschehen kann. Jetzt wird OV' nur dann noch weitergeleitet, wenn $N_1$ und $N_2$ gleich waren.

Mit Hilfe der Komplementdarstellung kann auf einfache Weise Addition und Subtraktion mit demselben Rechenwerk durchgeführt werden. Das folgende Beispiel zeigt, daß man mit einfachen „Rechentricks" oft erheblichen Programmieraufwand sparen kann.

Wenn $X_\alpha$ z.B. eine N-stellige Zahl mit der Basis α ist, von der eine weitere N-stellige Zahl $Y_\alpha$ zu subtrahieren ist, dann kann mittels Komplementbildung wie folgt gerechnet werden:

$$X_\alpha - Y_\alpha \quad \text{mit} \quad X_\alpha=\sum_{i=0}^{N-1} x_i\cdot\alpha^i,\quad Y_\alpha=\sum_{i=0}^{N-1} y_i\cdot\alpha^i$$

$$\text{Komplement von} \quad X_\alpha \;\hat{=}\; \alpha^N - X_\alpha$$

$$\xrightarrow{\text{Subtraktion}} \quad \alpha^N-\left[\alpha^N - X_\alpha + Y_\alpha\right]=X_\alpha - Y_\alpha$$

Erstaunlicherweise wird nicht das Komplement des Subtrahenden *Y* sondern das des Minuenden *X* gebildet. Dann wird der Subtrahend *Y* zu diesem Komplement addiert und von der Summe wird wieder das Komplement zu $\alpha^N$ gebildet. Wie man sieht, führt dies sofort zum Ergebnis *X*-*Y*. Da in der ALU von Mikrorechnern die Komplementbildung praktisch immer implementiert ist, jedoch die Subtraktion im Befehlssatz nicht immer vorhanden ist, kann man sich mit dem dargestellten Rechentrick sehr gut behelfen.

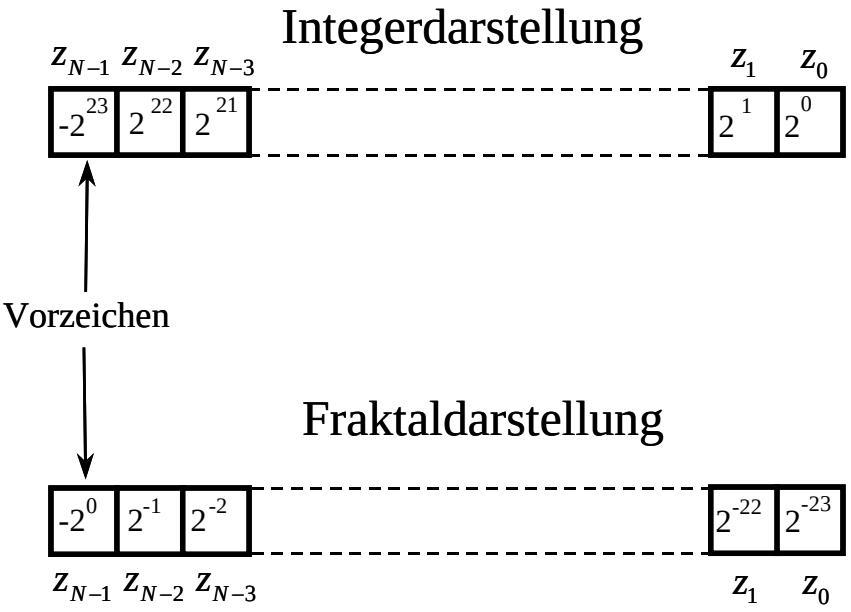



Bild 4.12: Fraktalzahlenbeispiel für 24 bit (DSP 56000, Motorola)

Während in Mikrocontrollern und in einfachen Mikroprozessoren in der Regel nur mit vorzeichenbehafteten Ganzzahlen (engl. Signed Integers) gerechnet wird, hat sich in digitalen Signalprozessoren die sogenannte Fraktaldarstellung durchgesetzt. Der Unterschied besteht aber lediglich in einer Verschiebung des Zahlenbereiches um eine (große) Zweierpotenz. Das bedeutet, daß Rechenwerke für Integer- und Fraktalzahlen grundsätzlich gleich aufgebaut sind – lediglich die Zahlenwertigkeit wird anders interpretiert. Der Übergang von einer vorzeichenbehafteten Ganzzahlendarstellung im Zweierkomplement $Z_i$ zur entsprechenden Fraktaldarstellung $Z_{Fi}$ erfolgt in einfacher Weise durch Division mit der höchsten Zweierpotenz, d.h. mit der des Vorzeichens, so daß das Vorzeichen jetzt die Potenz $2^0$ hat.

$$Z_{Fi} = \frac{Z_i}{2^{N-1}} = \frac{-z_{N-1}\cdot 2^{N-1} + \sum_{i=0}^{N-2} z_i \cdot 2^i}{2^{N-1}} = -z_{N-1}\cdot 2^0 + \sum_{i=0}^{N-2} z_i \cdot 2^{i-N+1} \qquad (4.8)$$

Die Fraktaldarstellung hat in der DSV[11] aus verschiedenen Gründen besondere Bedeutung. Einige davon seien kurz angeführt:

- In der Filtertheorie werden Filterkoeffizienten gerne in den Zahlenbereich {–1...+1} abgebildet. Wie aus Bild 4.12 ersichtlich ist, hat die Fraktalzahl den Minimalwert $-2^0 = -1$ und den Maximalwert $1\text{-}2^{23}$, d.h. ≈ 1.
- Viele Algorithmen der DSV beruhen auf fortgesetzten Multiplikationen mit Additionen. Dadurch, daß die höchstwertige Stelle in der Fraktaldarstellung $2^0$ ist, bleibt sie bei der Multiplikation erhalten, d.h. es ist keine Anpassung durch Schiebeoperationen nötig.

In der DSV kommen häufig Rechenoperationen vor, bei denen sehr große und sehr kleine Zahlen miteinander verrechnet werden müssen, z.B. bei einer Matrixinversion. So kann es selbst bei einer Wortlänge von 24bit zu erheblichen Rundungsfehlern kommen. Für solche Probleme kann die Gleitkommadarstellung Abhilfe schaffen.

Die Gleitkommadarstellung von Zahlen im Dezimalsystem ist uns von der Grundschule an vertraut. Eine Dezimalzahl $Z_d$ setzt sich aus einer Mantisse $M_d$ und dem Exponenten $E_d$ zusammen. Der gleiche Aufbau gilt auch für andere Zahlenbasen wie das folgende Beispiel demonstriert:

| Dezimal: | $Z_d = M_d \cdot 10^{E_d}$ | Binär: | $Z_b = M_b \cdot 2^{E_b}$ |
|---|---|---|---|
| | $3{,}7581 \cdot 10^4$ | | $1{,}1011 \cdot 2^{111}$ |

Während man im allgemeinen eine gewisse Freiheit bei der Wahl der Vor– und Nachkommastellen, d.h. bezüglich der Position des Kommas zuläßt, ist es in der Mikrorechnertechnik vorteilhaft, einen Standard zu verwenden, um die Portabilität von Software zu erleichtern.

Ein solcher Standard für Gleitkommazahlen existiert in Form des *IEEE-P754 Floating Point Standard*, der im folgenden anhand von Bild 4.13 vorgestellt wird. Man unterscheidet einfache und doppelte Genauigkeit, wobei die Wortlänge 32 bzw. 64bit ist. Das am weitesten links stehende Bit repräsentiert das Vorzeichen. Es ist unbedingt zu beachten, daß es sich hier nicht um eine Zweierkomplementdarstellung handelt (bei der das Vorzeichenbit mit der höchsten Zweierpotenz gewichtet ist), sondern eine „1“ im Vorzeichenbit bezeichnet per Definition eine negative Zahl, während bei „0“ die Zahl als positiv zu interpretieren ist. Man kann auch die Schreibweise $(-1)^V$ wählen.

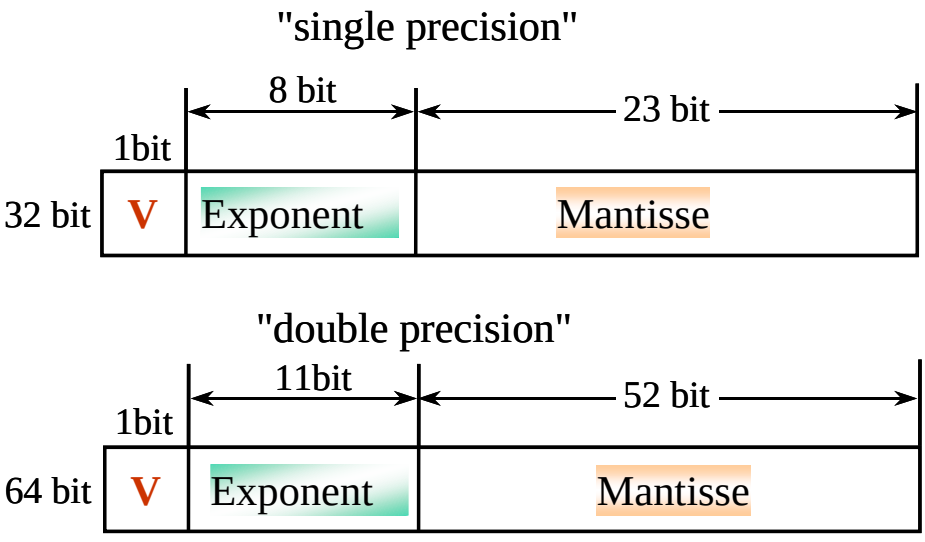



Bild 4.13: Der IEEE 754 Floating Point Standard

Rechts neben dem Vorzeichen folgt der Exponent, für den einige besondere Regeln eingeführt wurden, um zum einen positive und negative Zahlen und insbesondere die Werte ±0 und ±∞ darstellen zu können.

[11] digitale Signalverarbeitung

Mit dem 8bit breiten Exponenten (single precision) kann ein Zahlenbereich von 0...255 erfaßt werden. Um nun aber auch **negative Zahlen** darstellen zu können, gibt der Standard einen „Offset“ oder **„Bias“** von 127 vor, der implizit immer vom eingetragenen Wert $Z_E$ des Exponenten abgezogen wird. Wenn z.B. $Z_E$=1 ist, liefert die Interpretation gemäß *IEEE-P754 Floating Point Standard* den effektiven Exponenten $E_{eff}=1-127=-126$. Dies ist der größte zulässige negative Wert, weil $Z_E$=0 als „Ausnahme“ zur Definition von $\pm 0$ reserviert ist. Entsprechend wurde $Z_E$=255 als Ausnahme zur Beschreibung von $\pm\infty$ festgelegt. Für den effektiven Wert des Exponenten steht somit der Bereich $-126 \le E_{eff} \le 127$ zu Verfügung. Bei doppelter Genauigkeit hat man 11bit, d.h. den Zahlenbereich 0...2047 zur Verfügung. In analoger Weise sind auch hier wieder Offset und Ausnahmen definiert:

Bias: 1023 $\qquad Z_E=0 \Rightarrow \pm 0 \qquad Z_E=2047 \Rightarrow \pm\infty \qquad -1022 \le E_{eff} \le 1023$

An Beispielen wird der jeweils insgesamt darstellbare Zahlenbereich weiter unten erläutert.

Zunächst muß aber noch die Mantisse betrachtet werden. Sie ist grundsätzlich so aufgebaut, daß es nur **eine Stelle vor dem Komma** gibt, die immer den Wert „1“ hat. In der Darstellung kann daher diese Stelle eingespart, d.h. einfach **weggelassen** werden. Hingeschrieben wird somit nur der gebrochene oder Fraktalanteil der Mantisse.

$$M = 1 + \sum_{i=1}^{N} m_i \cdot 2^{-i} \; ; \quad \text{z.B. } N=23 \textit{ bzw. } 52$$

$$\uparrow \text{ erscheint nicht in der Darstellung} \qquad (4.9)$$

$$M = 1 + 0,5m_1 + 0,25m_2 + ...m_N \cdot 2^{-N}$$

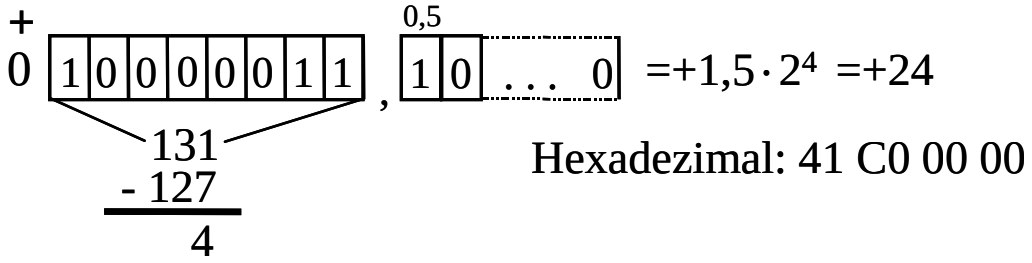
Wie aus (4.9) ersichtlich ist, liegt der dezimale Zahlenbereich der $N$ bit Mantisse zwischen 1 und „fast“ 2, d.h. genau $2–2^{–N}$. Für einfache Genauigkeit also $2–2^{–23}$ bzw. $2–2^{–52}$ für doppelte.

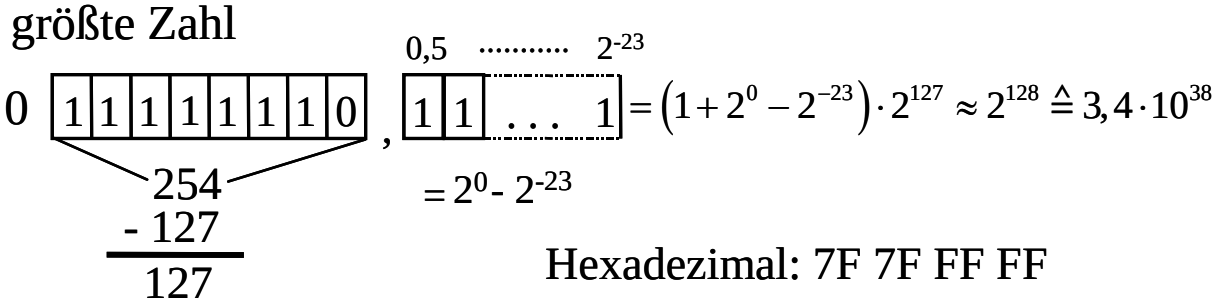
Zusammenfassend berechnet sich somit der Dezimalwert einer Gleitkommazahl nach IEEE-754 zu

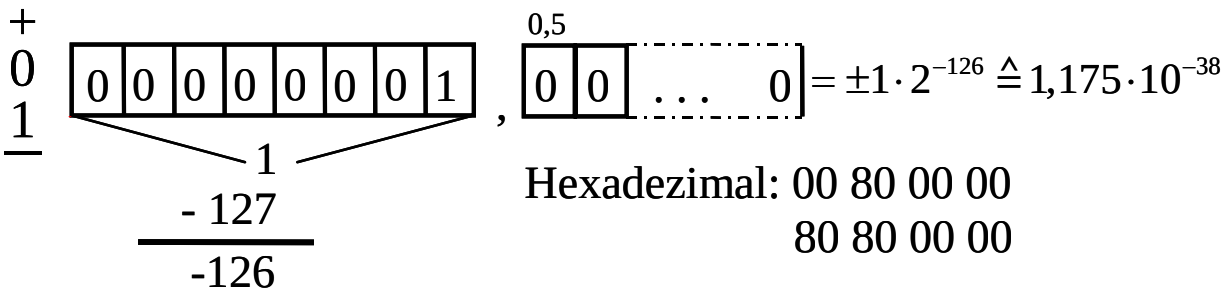
$$Z_d = (-1)^V \cdot M \cdot 2^{E_{eff}}, \quad mit\ E_{eff} = Z_E - Bias \qquad (4.10)$$

Es folgen einige Beispiele für den Umgang mit Gleitkomma- und Fraktalzahlen.

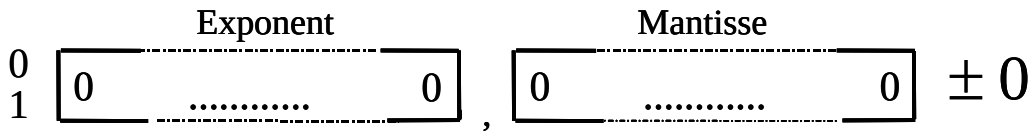
Beispiel zur Dezimalwertberechnung aus der Fraktaldarstellung:




## Ausnahmen



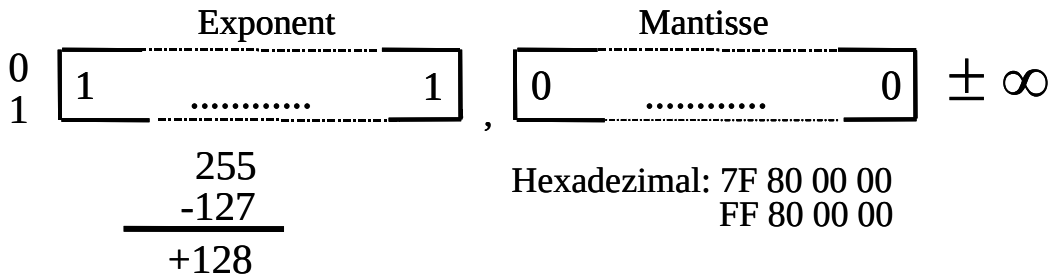




Bild 4.14: Beispiele für den Umgang mit Gleitkommazahlen nach IEEE 754

Gegeben: 24bit Fraktalzahl in Hexadezimalschreibweise: 99 30 7F

Binär: 1001 1001 0011 0000 0111 1111

Da das höchstwertige Bit „1“ ist, handelt es sich um eine negative Zahl, deren Dezimalwert sich dann wie folgt berechnet:

$$Z_d = \frac{1}{2^{23}}\left(-2^{23} + \text{Dezimalwert}\left\{19\ 30\ 7\text{F}_\text{h}\right\}\right) = -1 + \frac{1650815}{8388608} = -1 + 0{,}196792 = -0{,}8032075$$

Multiplikation von zwei Zweierkomplementzahlen *A* und *B*

$$A \cdot B = \left(-a_{N-1} \cdot 2^{N-1} + \sum_{i=0}^{N-2} a_i 2^i\right) \cdot \left(-b_{N-1} \cdot 2^{N-1} + \sum_{i=0}^{N-2} b_i 2^i\right) =$$

$$= a_{N-1} \cdot b_{N-1} \cdot 2^{2N-2} + \sum_{j=0}^{N-2} \sum_{i=0}^{N-2} a_i \cdot b_j \cdot 2^{i+j} - a_{N-1} \cdot 2^{N-1} \cdot \sum_{i=0}^{N-2} b_i 2^i - b_{N-1} \cdot 2^{N-1} \cdot \sum_{i=0}^{N-2} a_i 2^i$$

Entsprechend der eingeführten Zweierkomplementdarstellung wird die Beziehung

$$\text{A} + \overline{\text{A}} = 2^\text{N} - 1 \quad \Rightarrow \quad -\text{A} = -2^\text{N} + \overline{\text{A}} + 1$$

auf die negativen Mischterme

$$-a_{N-1} \cdot 2^{N-1} \cdot \sum_{i=0}^{N-2} b_i 2^i = -2^{N-1} \cdot \sum_{i=0}^{N-2} a_{N-1} b_i \cdot 2^i = 2^{N-1} \cdot \left[-\sum_{i=0}^{N-2} a_{N-1} b_i \cdot 2^i\right]$$

und

$$-b_{N-1} \cdot 2^{N-1} \cdot \sum_{i=0}^{N-2} a_i 2^i = -2^{N-1} \cdot \sum_{i=0}^{N-2} b_{N-1} a_i \cdot 2^i = 2^{N-1} \cdot \left[-\sum_{i=0}^{N-2} b_{N-1} a_i \cdot 2^i\right]$$

angewandt. Damit erhält man:

$$2^{N-1}\left[-2^{N-1} + \sum_{i=0}^{N-2} \overline{a_{N-1} b_i} \cdot 2^i + 1\right] = -2^{2N-2} + \sum_{i=0}^{N-2} \overline{a_{N-1} b_i} \cdot 2^{i+N-1} + 2^{N-1}$$

und

$$2^{N-1}\left[-2^{N-1} + \sum_{i=0}^{N-2} \overline{b_{N-1} a_i} \cdot 2^i + 1\right] = -2^{2N-2} + \sum_{i=0}^{N-2} \overline{b_{N-1} a_i} \cdot 2^{i+N-1} + 2^{N-1},$$

woraus sich die Summe

$$-2^{2N-1} + 2^N + \sum_{i=0}^{N-2} \overline{a_{N-1} b_i} \cdot 2^{i+N-1} + \sum_{i=0}^{N-2} \overline{b_{N-1} a_i} \cdot 2^{i+N-1}$$

ergibt. Das gesamte Produkt stellt sich dann wie folgt dar:

$$A \cdot B = -2^{2N-1} + 2^N + a_{N-1} \cdot b_{N-1} \cdot 2^{2N-2} + \sum_{j=0}^{N-2} \sum_{i=0}^{N-2} a_i \cdot b_j \cdot 2^{i+j} + \\ + \sum_{i=0}^{N-2} \overline{a_{N-1} b_i} \cdot 2^{i+N-1} + \sum_{i=0}^{N-2} \overline{b_{N-1} a_i} \cdot 2^{i+N-1} \qquad (4.11)$$

Beispiel für die Multiplikation von zwei 8bit „signed“ Integerzahlen *A* und *B* mit 16bit–Ergebnis:

$$A \cdot B = -2^{15} + 2^8 + a_7 \cdot b_7 \cdot 2^{14} + \sum_{j=0}^{6} \sum_{i=0}^{6} a_i \cdot b_j \cdot 2^{i+j}$$

$$+ \sum_{i=0}^{6} \overline{a_7 b_i} \cdot 2^{i+7} + \sum_{i=0}^{6} \overline{b_7 a_i} \cdot 2^{i+7}$$

Die Produktbildung der Komponenten $a_i b_j$ erfolgt mittels UND- und der negierten $\overline{b_i a_j}$ mittels NAND-Gattern. Wenn man die einzelnen Binärprodukte nach ihrer Wertigkeit ordnet, entsteht ein Matrixschema, das z.B. mit zeilenweiser Addition schrittweise abgearbeitet werden kann. Da es bei den Multiplizier-Akkumulier-Operationen (MAC) in der DSV aber auf Geschwindigkeit ankommt, müssen weitgehend Methoden der Parallelberechnung angewandt werden. Näheres dazu folgt in Abschnitt 4.9.4.

## 4.3 Aufbau der ALU und ihre Integration in den Mikrorechner

### 4.3.1 Addier/Subtrahierwerke

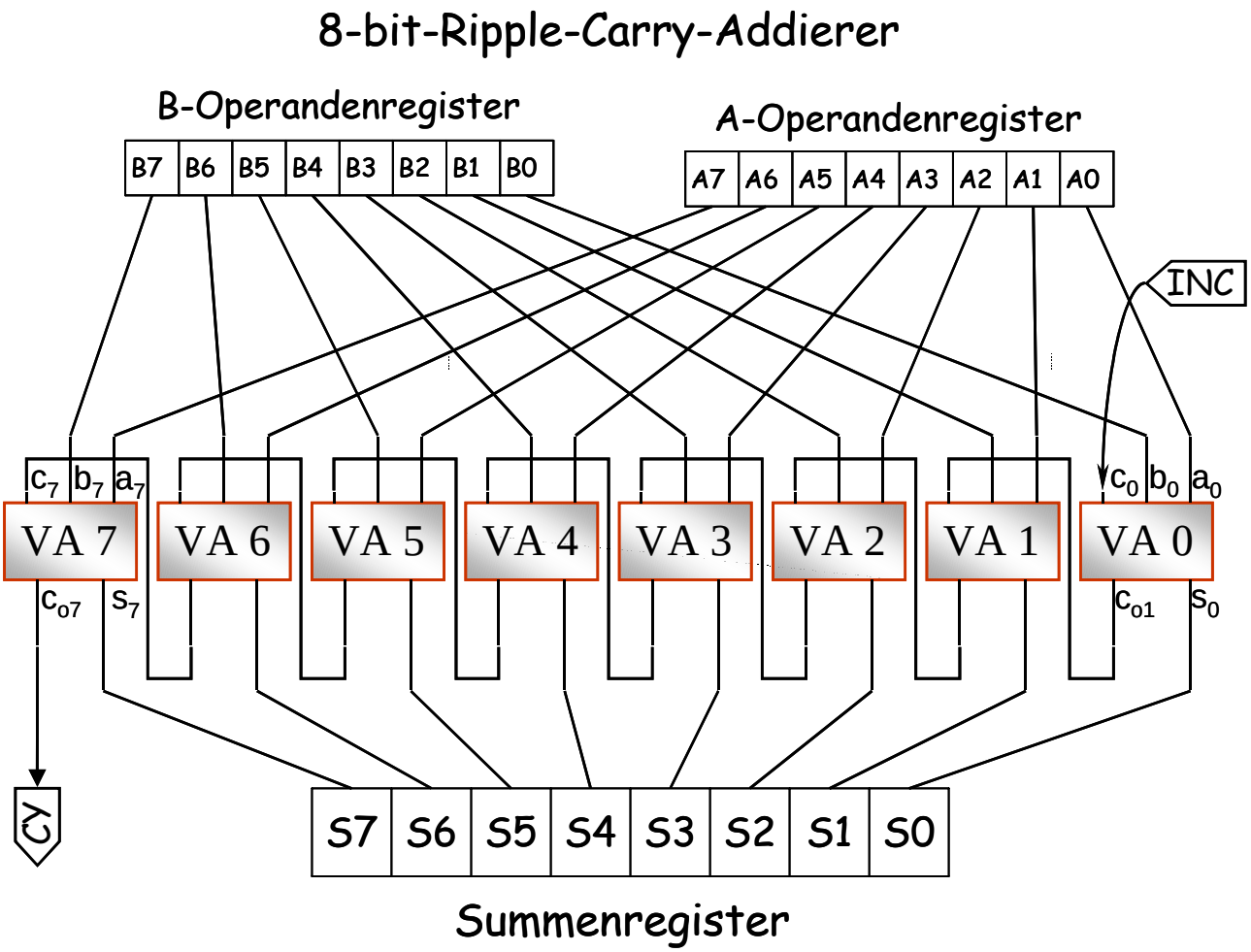



Bild 4.15: Aufbau eines 8-bit-Ripple–Carry Addierers mittels einer Volladdiererkette

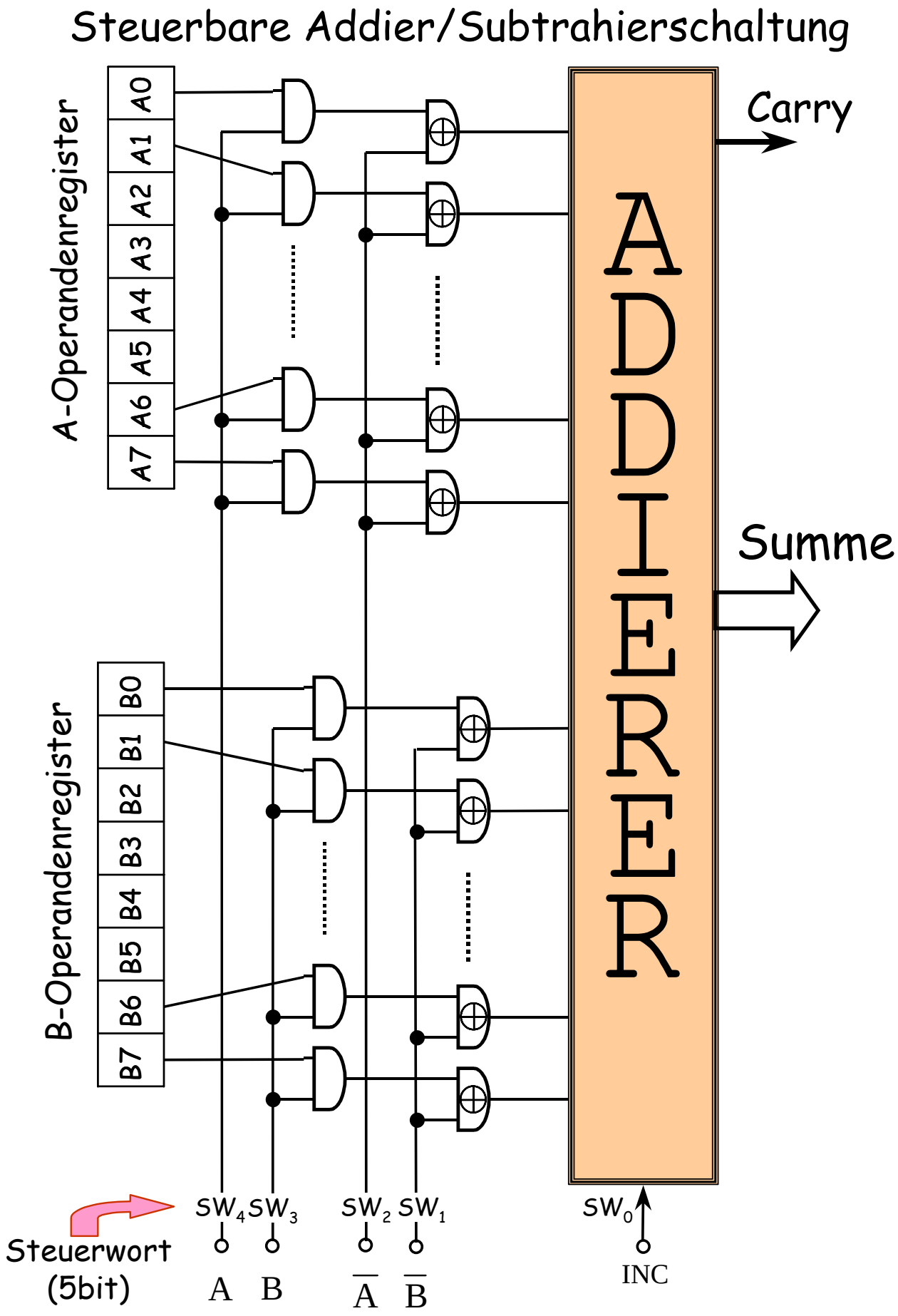



Bild 4.16: Das Addier/Subtrahier-Rechenwerk als Kernstück einer Mikrorechner ALU

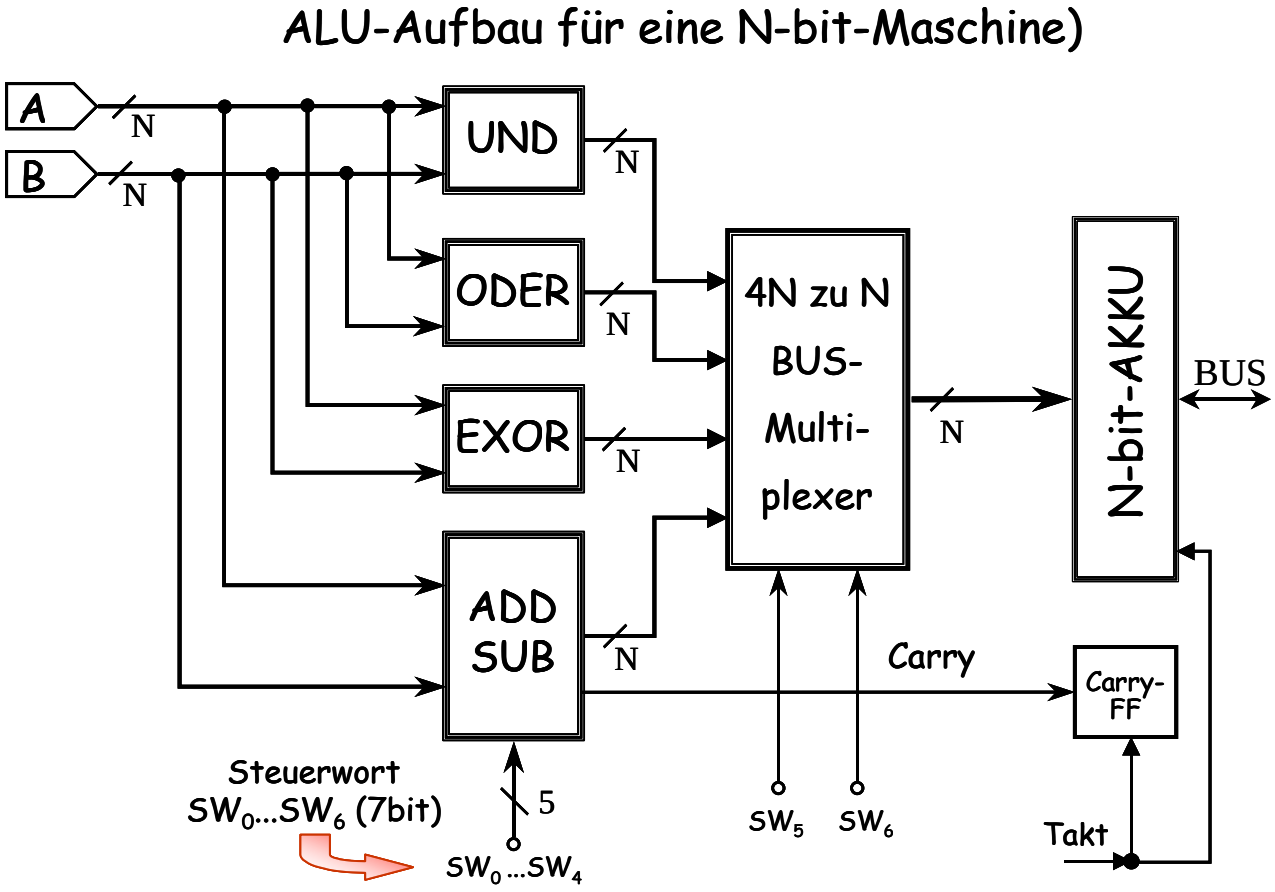



Bild 4.17: Die komplette ALU eines N–bit–Mikrorechners

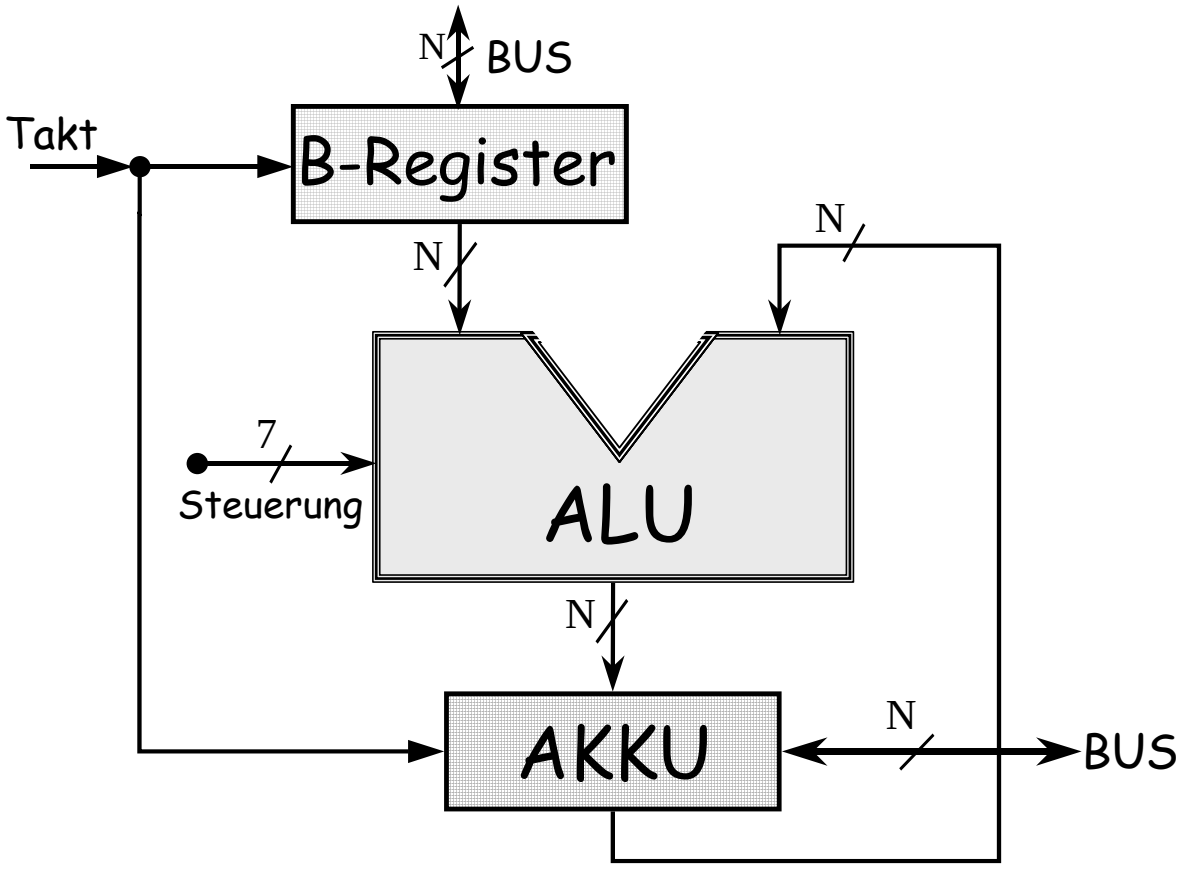



Bild 4.18: Eine Mikrorechner-ALU in Verbindung mit den wichtigsten Arbeitsregistern

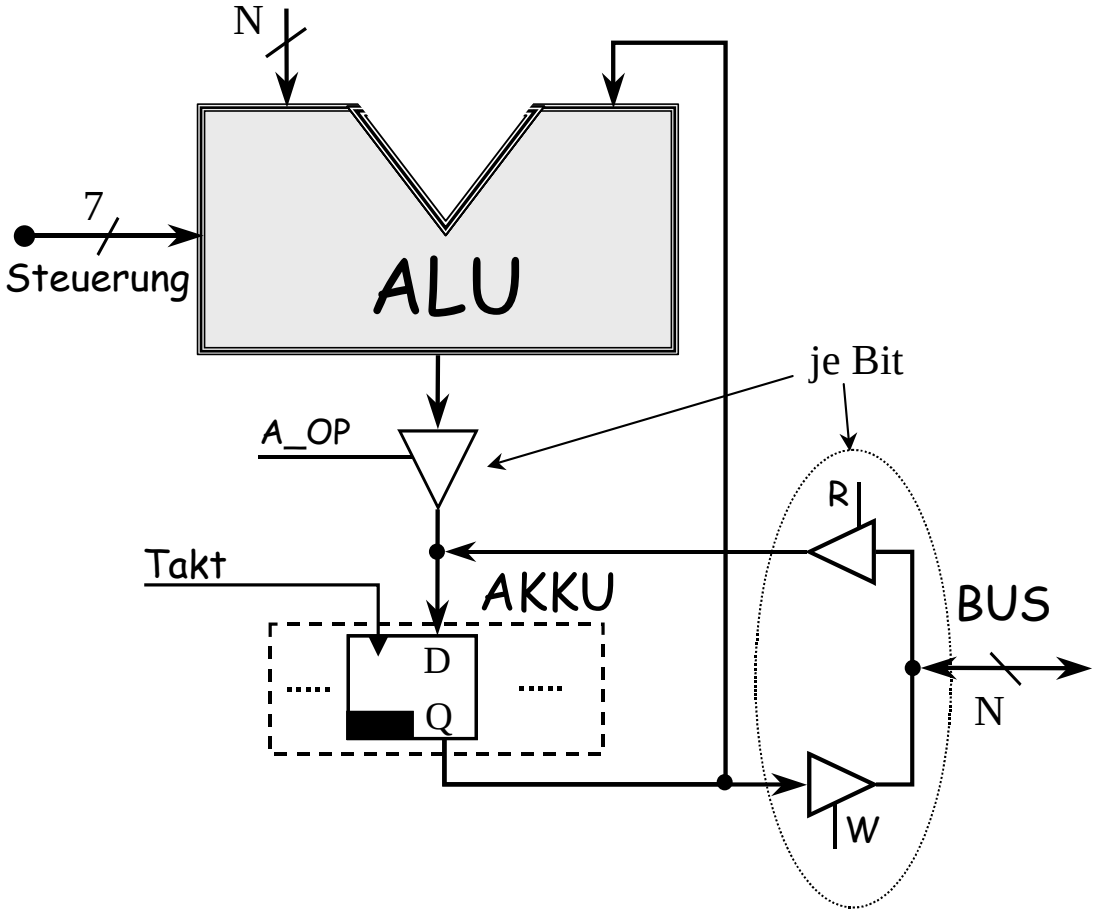



Bild 4.19: Einzelheiten der Datenflußsteuerung am Akkumulator

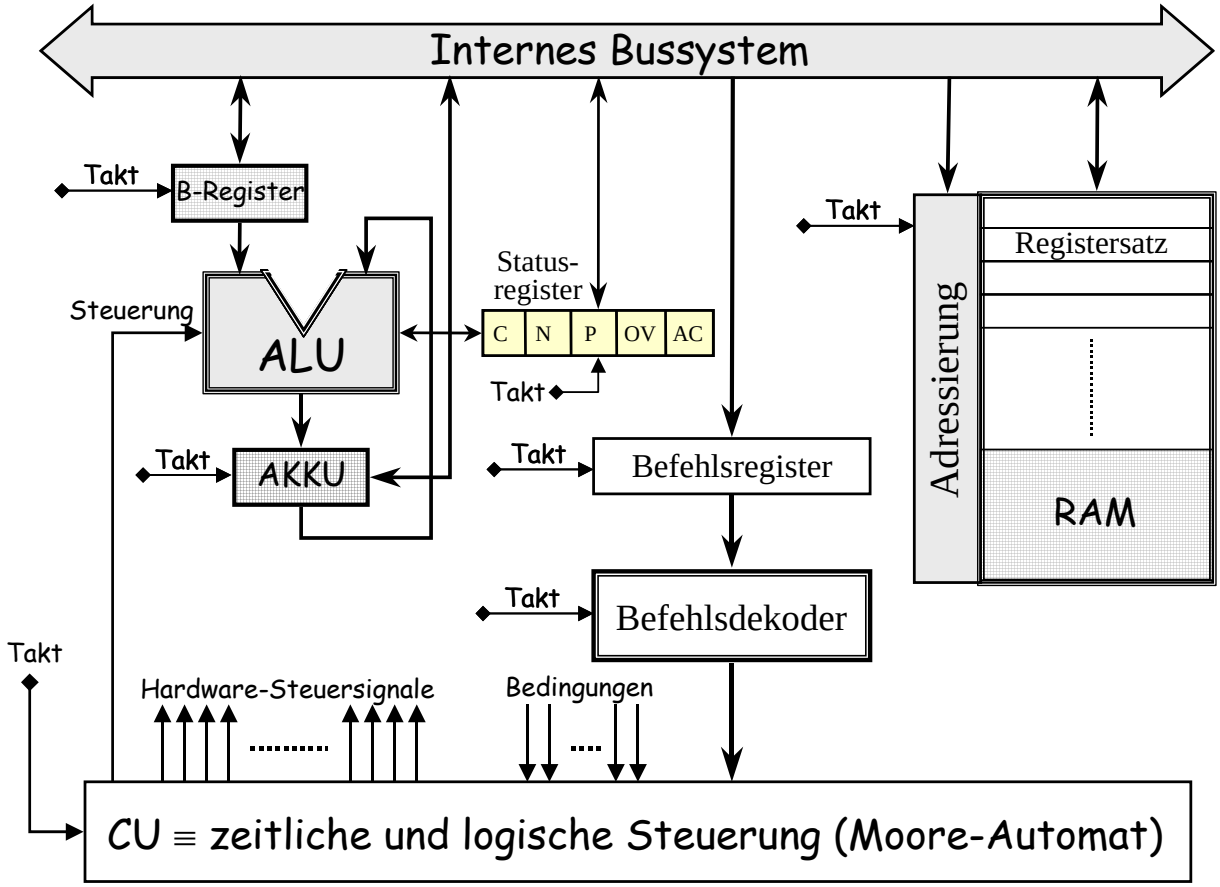



Bild 4.20: Innerer Grundaufbau eines Mikroprozessors

## 4.3.2 Addierschaltungen mit höherer Leistungsfähigkeit

### 4.3.2.1 Das Carry-Look-Ahead-Addiererprinzip

Zweck: "Vorrausschauende" Berechnung aller Überträge in einem Addierer, so daß der Fall des wellenartigen Durchlaufens (Ripple) nicht mehr vorkommt. Der Carry-Look-Ahead-Addierer wird damit um Größenordnungen schneller als z.B. ein Ripple-Carry-Addierer.

**Grundüberlegungen:**

1. Wann entsteht bei einem *N*-bit-Binäraddierer mit den Eingangsoperanden $\boldsymbol{A}=a_0...a_{N-1}$ und $\boldsymbol{B}=b_0...b_{N-1}$ ein Übertrag $c_{i+1}$ in der Addierstufe *i*, unabhängig von den Werten aller Binärvariablen der Vorstufen, d.h. unabhängig von $a_0...a_{i-1}$ $b_0...b_{i-1}$ sowie $c_0$ ?
   **Antwort**: Falls $a_i \cdot b_i = 1$
   Dieser Fall wird als Carry-Erzeugung oder Carry-Generate bezeichnet, d.h.:
   $g_i = a_i \cdot b_i$
2. Wann wird bei einem *N*-bit-Binäraddierer mit den Eingangsoperanden $\boldsymbol{A}=a_0...a_{N-1}$ und $\boldsymbol{B}=b_0...b_{N-1}$ ein Übertrag $c_{i+1}$, über die Addierstufe *i* weitergegeben, der nicht in dieser Stufe entstanden ist. Dabei muß also **abhängig** von den Werten der Binärvariablen der Vorstufen, d.h. **abhängig** von $a_0...a_{i-1}$, $b_0...b_{i-1}$ und $c_0$ ein Eingangsübertrag $c_i=1$ entstanden sein.
   **Antwort:** Der Übertrag wird weitergeleitet, falls $a_i \cdot + b_i = 1$
   Dieser Fall wird als Carry-Weiterleitung oder Carry-Propagate bezeichnet, d.h.: $p_i = a_i \cdot + b_i$

Im weiteren wird die Schreibweise

$$g_i = a_i \cdot b_i \quad \text{für Carry-Generate} \tag{4.12}$$

und

$$p_i = a_i \cdot + b_i \quad \text{für Carry-Propagate} \tag{4.13}$$

verwendet, womit sich der Übertrag in die jeweils nächsthöhere Zweierpotenz zu

$$c_{i+1} = g_i + c_i \cdot p_i \tag{4.14}$$

ergibt.

Mit den obigen Überlegungen lassen sich jetzt die booleschen Gleichungen für die einzelnen Überträge angeben:

$$\begin{aligned}
c_1 &= g_0 + p_0 \cdot c_0 \\
c_2 &= g_1 + p_1 \cdot c_1 = g_1 + p_1 \cdot \left(g_0 + p_0 \cdot c_0\right) \\
c_3 &= g_2 + p_2 \cdot c_2 = g_2 + p_2 \cdot \left[g_1 + p_1 \cdot \left(g_0 + p_0 \cdot c_0\right)\right] \\
c_4 &= g_3 + p_3 \cdot c_3 = g_3 + p_3 \cdot \left[g_2 + p_2 \cdot \left[g_1 + p_1 \cdot \left(g_0 + p_0 \cdot c_0\right)\right]\right] \\
c_5 &= g_5 + p_5 \cdot c_5 = .....
\end{aligned} \tag{4.15}$$

....

Man sieht, daß die Ausdrücke für die Carry-Vorausberechnung mit zunehmenden Zweierpotenzen rasch wachsen. In der Praxis wird man deshalb für lange Addierer eine Partitionierung einführen, die Geschwindigkeitseinbussen hinnimmt, aber dafür mit weniger Hardwareaufwand sehr überschaubar realisiert werden kann. Es gibt z.B. Standard-Logik-Bausteine, in denen 4 Summenberechnungen und die Bestimmung des $c_4$-Bits vorgenommen werden. Zum Aufbau längerer Addierer lassen sich die Bausteine kaskadieren, wobei zur Carry-Berechnung für die höchste Zweierpotenz jetzt natürlich alle Bausteine der Kaskade durchlaufen werden müssen. D.h. für einen 16-bit-Addierer erhält man genau die vierfache Laufzeit eines 4-bit-Bausteins. Da die einzelnen Bausteine aber durch das Carry-Look-Ahead-Prinzip laufzeitoptimiert sind, verbleiben immer noch deutliche Vorteile im Vergleich mit einem einfachen Ripple-Carry-Addierer. Im folgenden wird dies genauer quantifiziert.

Mit der folgenden einfachen Manipulation (Hinzufügen von Nulltermen) erhält man eine alternative Teilsummenberechnung (Halbaddiererfunktion), in der nun die Ausdrücke für Carry-Propagate und für Carry-Generate vorkommen:

$$\begin{aligned} a_i \oplus b_i &= a_i \cdot \overline{b_i} + b_i \cdot \overline{a_i} + a_i \cdot \overline{a_i} + b_i \cdot \overline{b_i} = \\ &= \left(a_i + b_i\right) \cdot \left(\overline{a_i} + \overline{b_i}\right) = \left(a_i + b_i\right) \cdot \overline{a_i \cdot b_i} \\ &= p_i \cdot \overline{g_i} \end{aligned} \tag{4.16}$$

Des weiteren kann die allgemeine Beziehung zur Carry-Berechnung $c_{i+1} = g_i + c_i \cdot p_i$ auch in der Form $\boldsymbol{c_{i+1} = g_i \cdot p_i + c_i \cdot p_i = p_i \cdot (g_i + c_i)}$ geschrieben werden, da

$$p_i \cdot g_i = \left(a_i + b_i\right) \cdot a_i \cdot b_i = a_i \cdot b_i \,. \tag{4.17}$$

Damit kann der Gleichungssatz für die Carry-Vorausberechnung in folgender Form alternativ dargestellt werden:

$$\begin{aligned} c_1 &= p_0 \cdot \left(g_0 + c_0\right) \\ c_2 &= p_1 \cdot \left(g_1 + c_1\right) = p_1 \cdot \left[ \underbrace{g_1}_{A} + \underbrace{p_0}_{B} \cdot \left( \underbrace{g_0 + c_0}_{C} \right) \right] \\ &= p_1 \cdot \left[ \left(g_1 + p_0\right) \cdot \left(g_1 + g_0 + c_0\right) \right] \\ c_3 &= p_2 \left(g_2 + c_2\right) = p_2 \cdot \left[ \underbrace{g_2}_{A} + \underbrace{p_1}_{B} \cdot \left( \underbrace{g_1 + p_0}_{C} \right) \cdot \left( \underbrace{g_1 + g_0 + c_0}_{D} \right) \right] \\ &= p_2 \cdot \left(g_2 + p_1\right) \cdot \left(g_2 + g_1 + p_0\right) \cdot \left(g_2 + g_1 + g_0 + c_0\right) \end{aligned} \tag{4.18}$$

Obwohl der Ausdruck für $c_4$ schon recht umfangreich wird, läßt er sich schnell mit der Bildungsregel aus dem obigen Gleichungssystem (4.18) angeben:

$$c_4 = p_3\left(g_3 + c_3\right) =$$
$$p_3 \cdot \left(g_3 + p_2\right) \cdot \left(g_3 + g_2 + p_1\right) \cdot \left(g_3 + g_2 + g_1 + p_0\right) \cdot \left(g_3 + g_2 + g_1 + g_0 + c_0\right) \tag{4.19}$$

Die Berechnung der Summenbits $s_i$ ist sehr einfach. Wenn man von der Teilsumme $a_i \oplus b_i = p_i \cdot \overline{g_i}$ ausgeht, ergibt sich

$$s_i = a_i \oplus b_i \oplus c_i = p_i \cdot \overline{g_i} \oplus c_i \,. \tag{4.20}$$

Es müssen also die vorausberechneten Carrybits lediglich mit dem Produkt $p_i \cdot \overline{g_i}$

EXOR-verknüpft werden.

Aus den obigen Gleichungen kann unmittelbar die Vorschrift für den Hardwareaufbau abgeleitet werden. Aus der Gleichung für $c_4$ ersieht man den Geschwindigkeitsvorteil: Da alle $g_i$ und $p_i$ parallel erzeugt werden erhält man maximal nur drei Gatterlaufzeiten für die Carry-Berechnung – für $c_0$ sind es sogar nur zwei. Bei einem kaskadierten 16-bit-Addierer benötigt somit der Durchlauf von $c_0$ nach $c_{16}$ nur 8 Gatterlaufzeiten.

Das folgende Schaltungsbeispiel zeigt einen 4-bit-Addierer, in dem das beschriebene Carry-Look-Ahead-Prinzip realisiert ist. Bei handelsüblichen Bausteinen (z.B. 74x283, wobei x{S, LS, AS, ALS, C, HC, HCT..} die Technologie kennzeichnet), kann die Innenschaltung abweichen, z.B., wenn NAND- und NOR-Gatter technologiebedingt gegenüber UND- bzw. ODER-Gattern Geschwindigkeitsvorteile bieten.

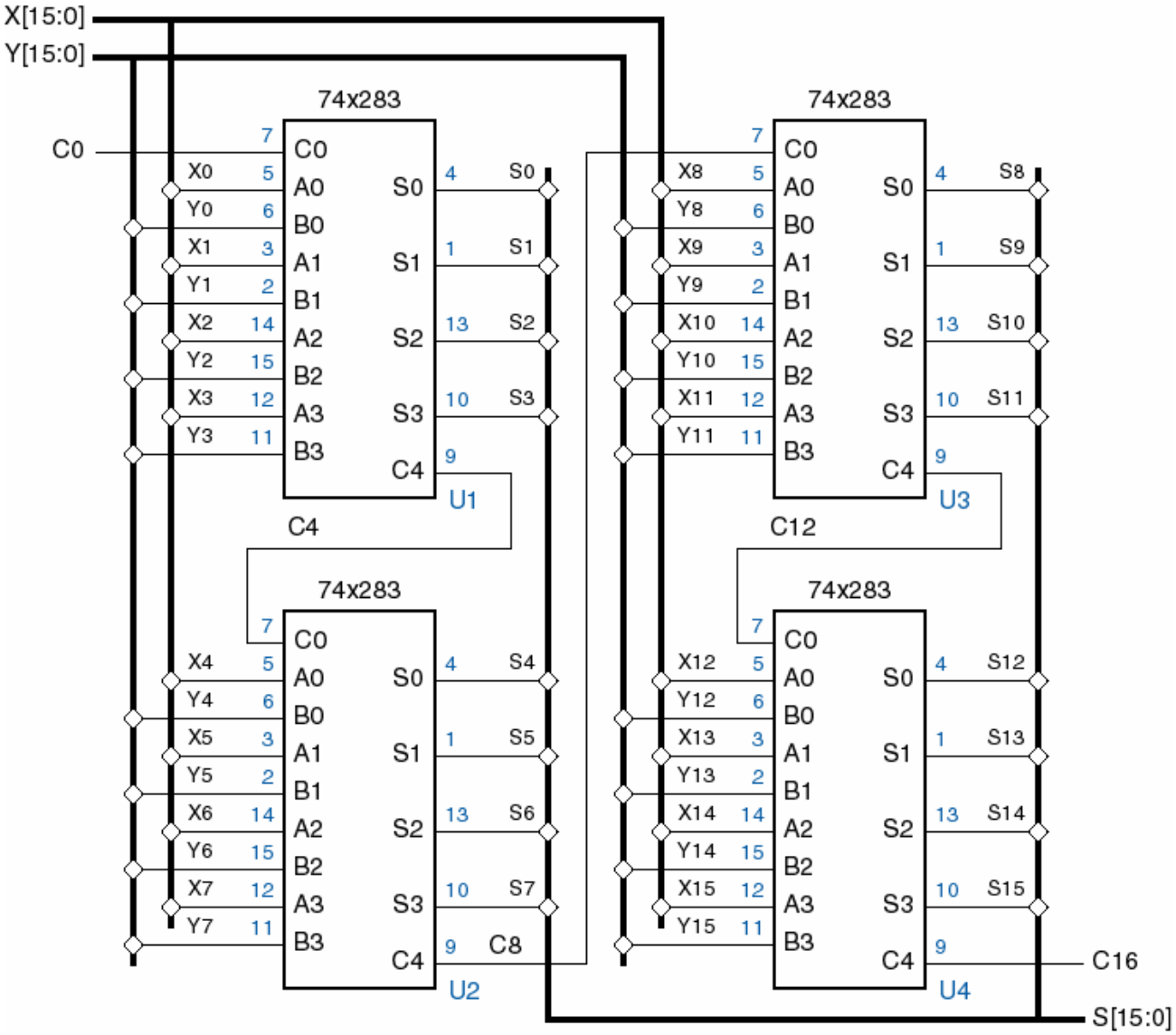



Bild 4.21: Anwendungsbeispiel für einen kaskadierten 16-bit-Addierer aus 4 Standard-Carry-Look-Ahead Addierbausteinen für je 4 bit.

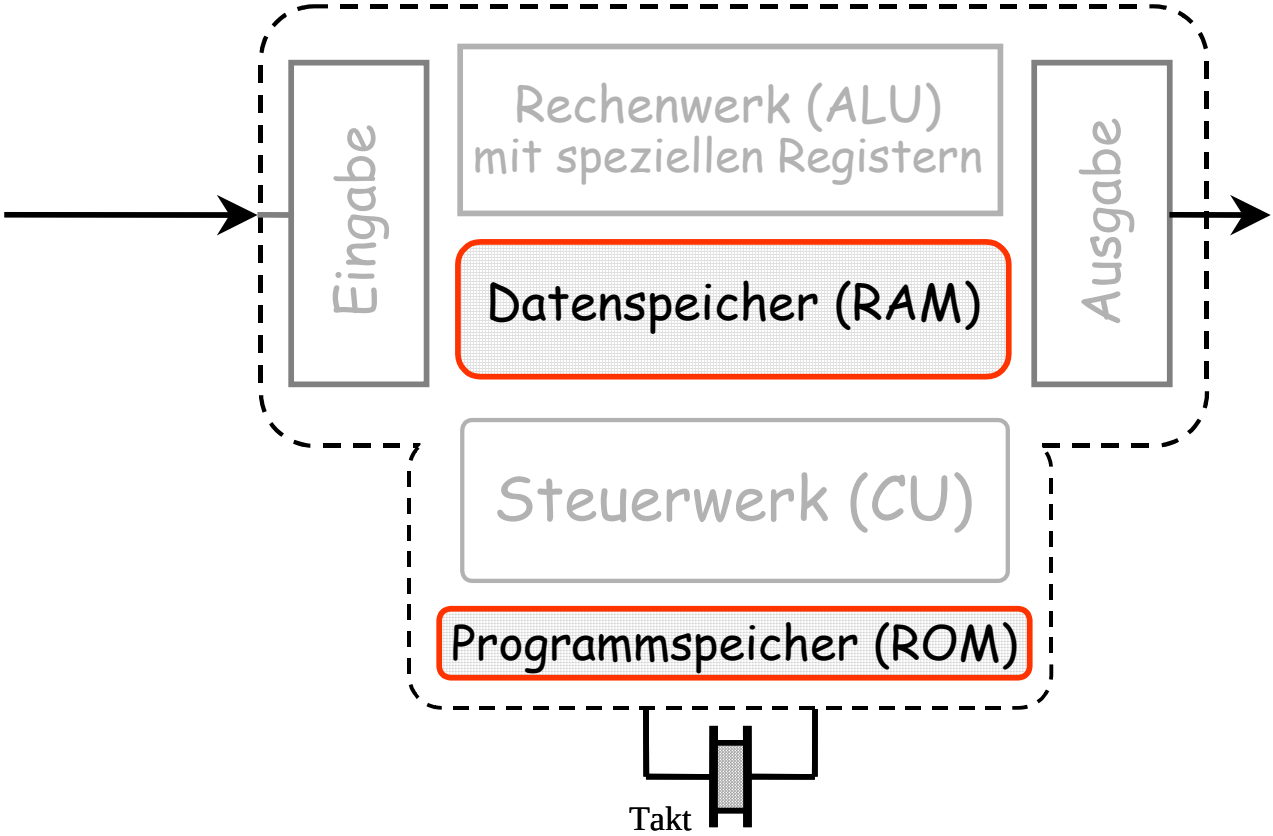



Bild 4.22: Programm- und Datenspeicher als weitere wichtige Mikrorechnerbestandteile

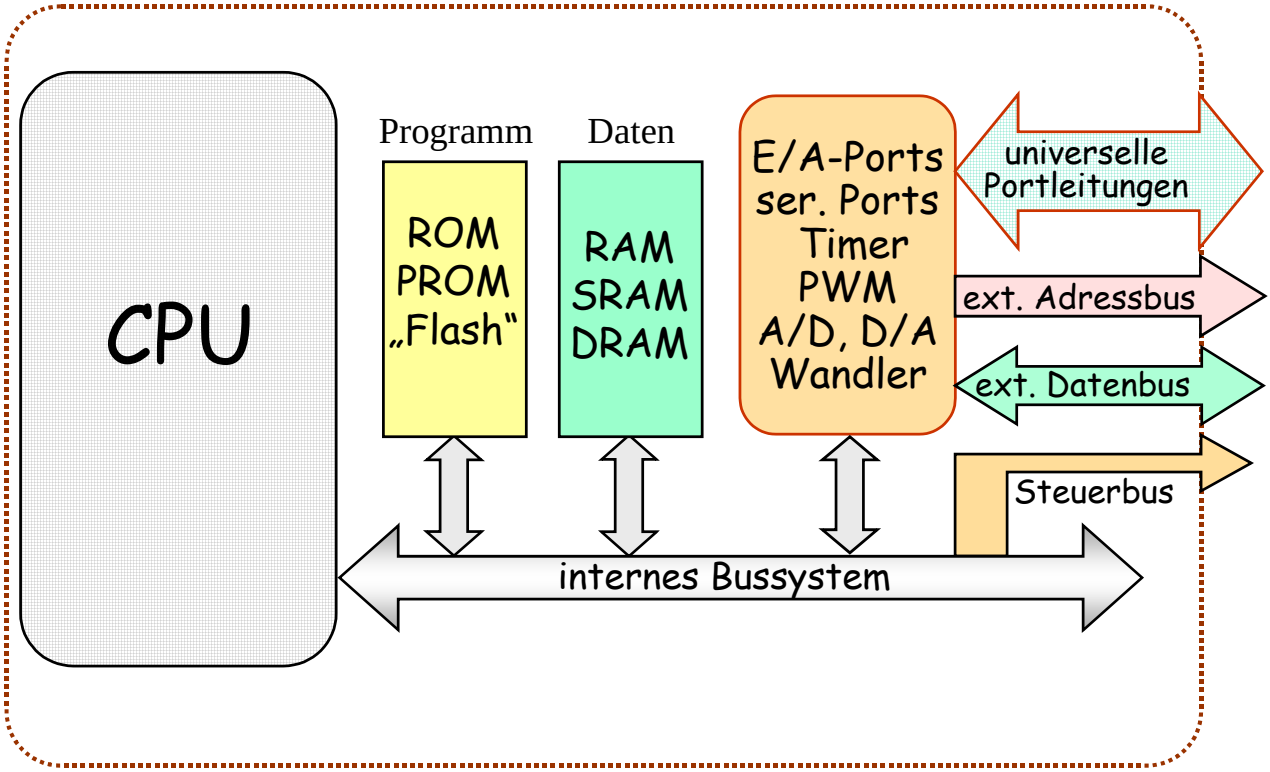



Bild 4.23: Einbindung der Speicher beim Aufbau eine kompletten Mirkorechnersystems

### 4.4.1 Speicherarten in Mikrorechnersystemen – vgl. Bild 4.23

## Festwertspeicher

behalten Information auch ohne Stromversorgung (meist nur lesbar)

ROM ≡ **R**ead **O**nly **M**emory

## Programmierbare Festwertspeicher

„schnell" lesbar, löschbar und„langsam" beschreibbar

PROM ≡ **P**rogrammable **R**ead **O**nly **M**emory

EPROM ≡ **E**lectrically **P**rogrammable **R**ead **O**nly **M**emory (Löschen mit UV-Licht)

EEPROM ≡ **E**lectrically **E**rasable **P**rogrammable **R**ead **O**nly **M**emory

elektrisch lösch- und wiederbeschreibbar

FLASH ≡ EEPROM mit besonderer Blockstruktur, die sehr schnelles Löschen ermöglicht

## Schreib/Lesespeicher mit wahlfreiem Zugriff

RAM ≡ **R**andom **A**ccess **M**emory

- Register, Register-File (Akkumulator) aus Flip-Flops aufgebaut
- statisches RAM
- dynamisches RAM

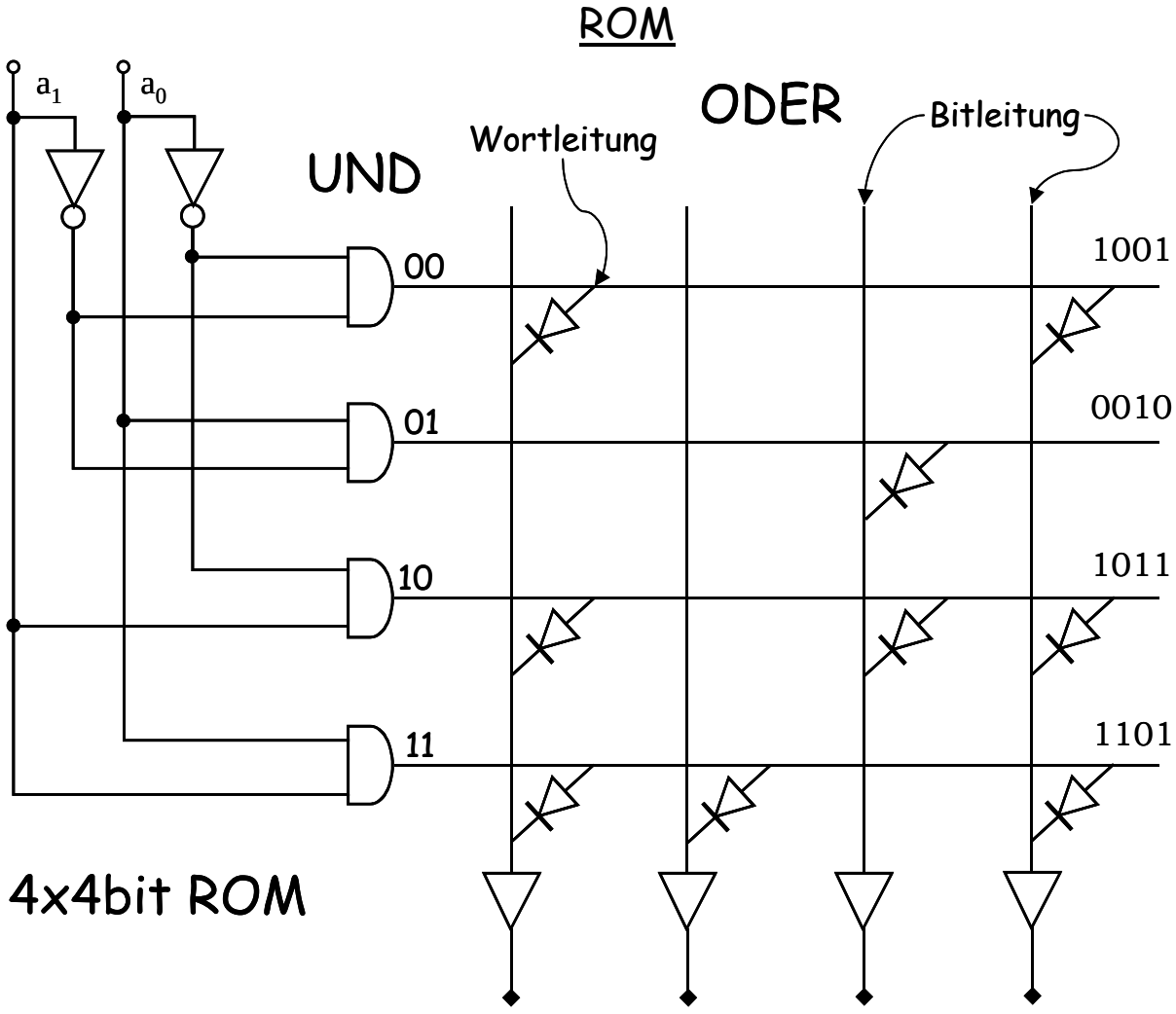



Bild 4.24: Prinzipaufbau eines **R**ead **O**nly **M**emory (ROM)

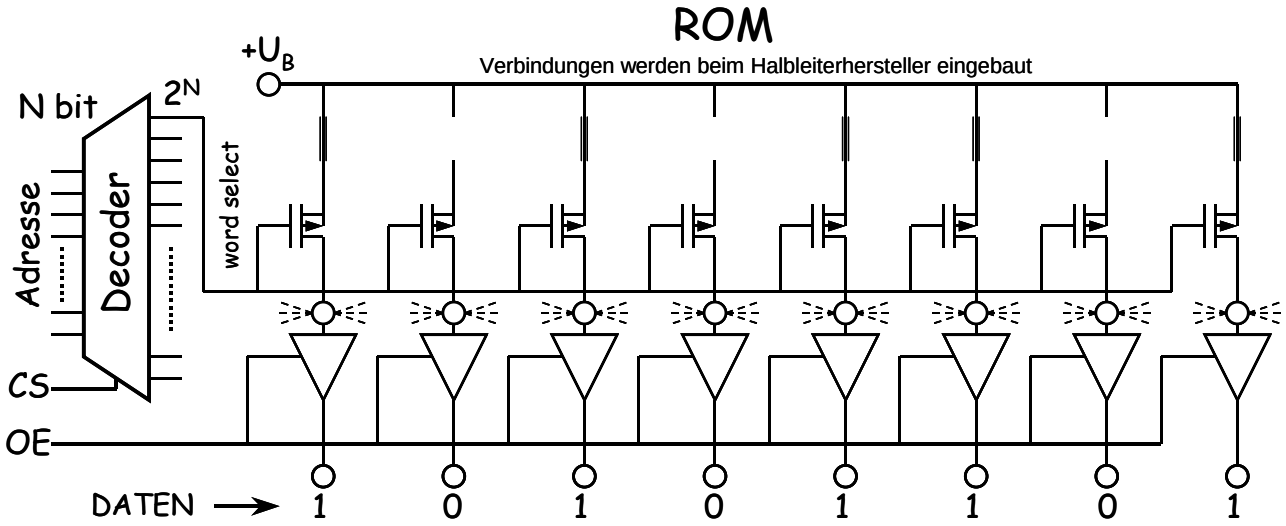



Bild 4.25: Realer ROM–Aufbau

CS ≡ Chip-Select (Bausteinauswahl)
OE ≡ Output Enable (aktivieren der „Tristate“–fähigen Ausgangstreiberstufen)

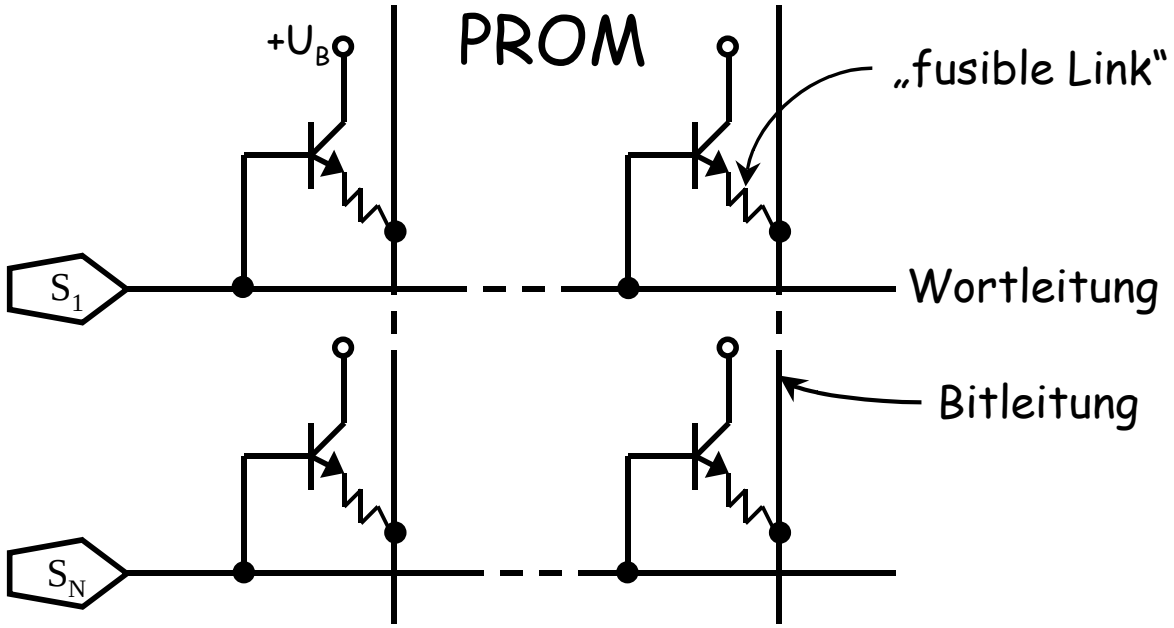



Bild 4.26: Irreversible Programmierbarkeit durch Trennen von metallischen Verbindungen

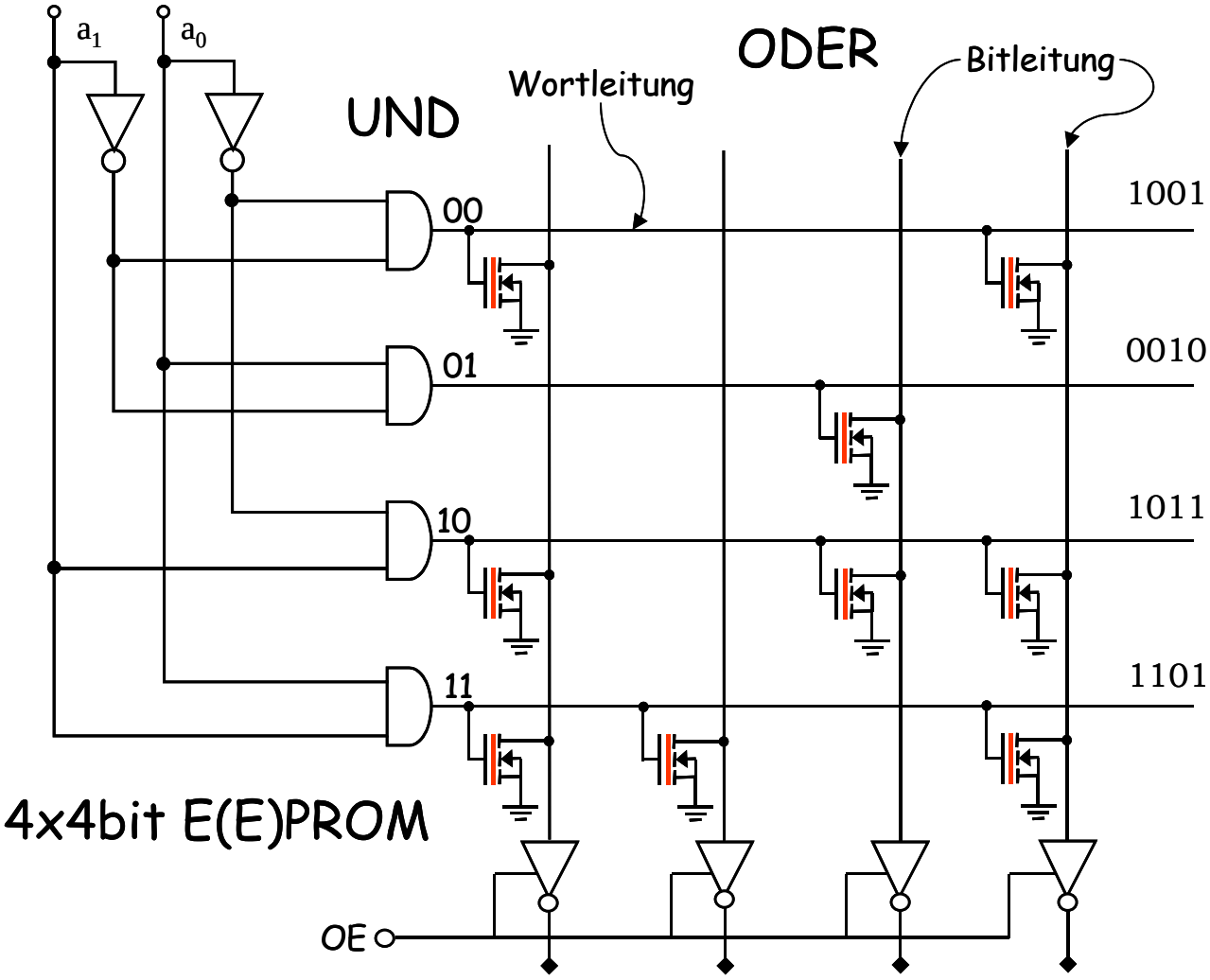



Bild 4.27: Prinzip des lösch- und wiederbeschreibbaren PROM auf der Basis von MOSFETs mit „Floating Gate“

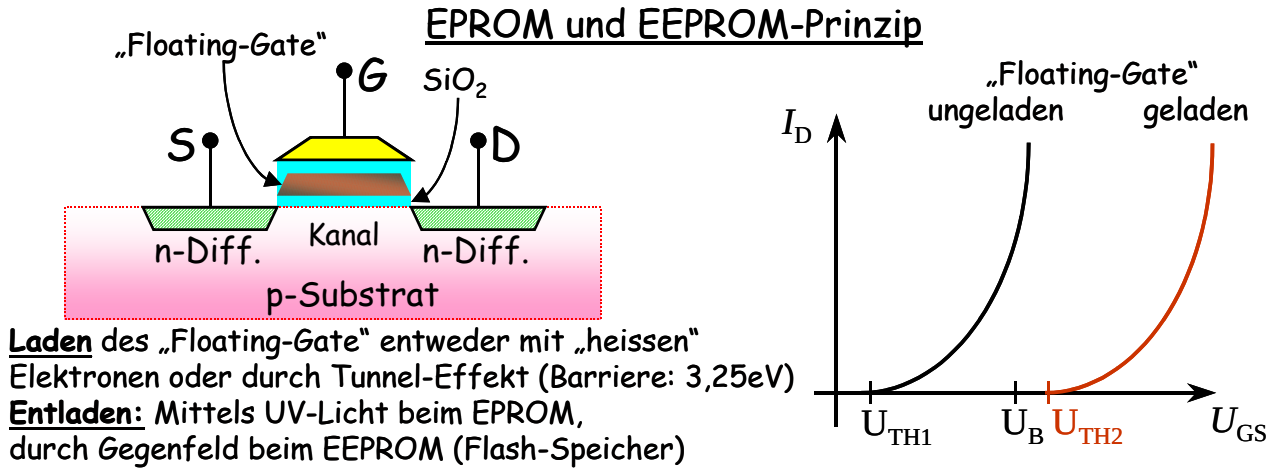



Bild 4.28: Aufbau der E(E)PROM Zelle

Gate und Floating-Gate bestehen aus Polysilizium und sind in isolierendem $SiO_2$ eingebettet. Es gibt 2 Mechanismen, durch die Elektronen die Energiebarriere von 3,5eV überwinden können:
1. Durch **hohe Feldstärke im Kanal** erreichen die Elektronen eine genügend hohe **kinetische Energie**. Im Transistor fließt dabei ein Drain-Source-Strom.
2. Bei stromlosem Transistor wird eine **hohe Spannung an das Gate** angelegt, so daß aufgrund extremer Feldstärke das sogenannte **„Fowler-Northeim-Tunneling“** einsetzt, wobei Elektronen den Isolator „durch-tunneln“ und auf das Floating-Gate gelangen.

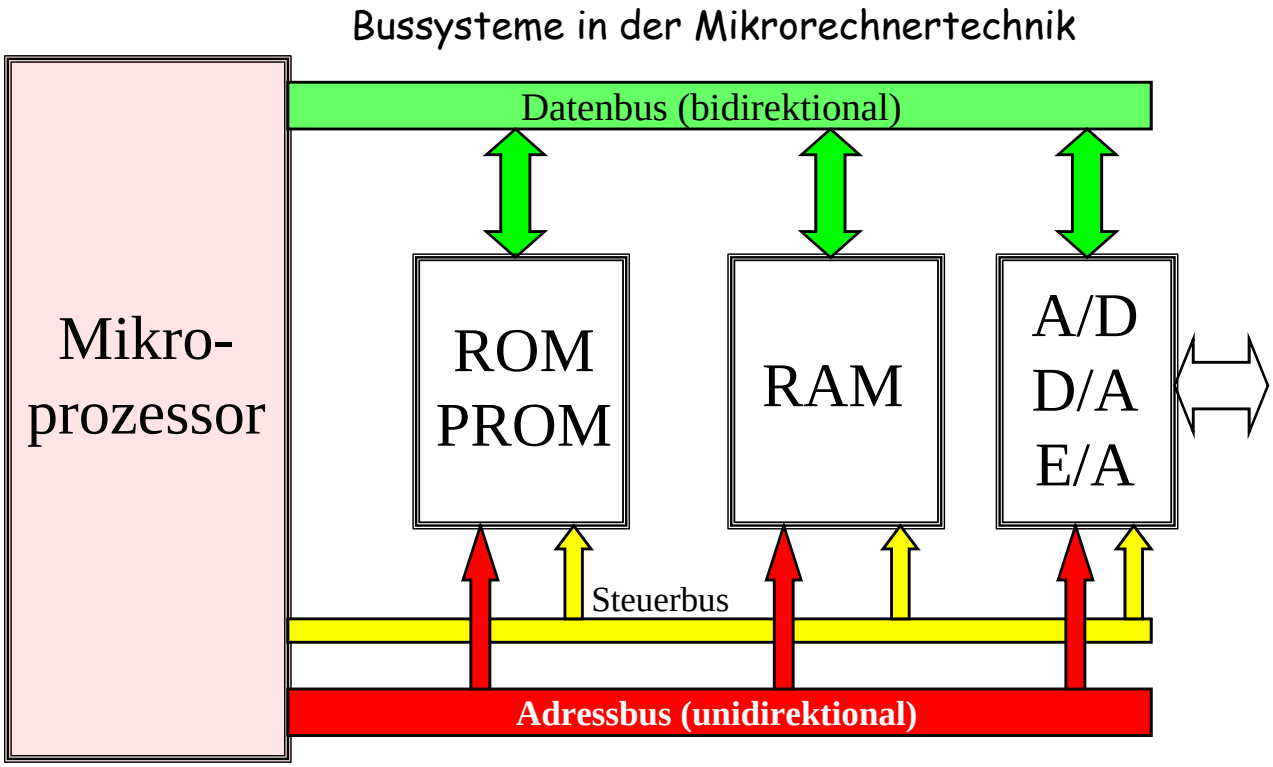



Bild 4.29: Aufgaben der verschiedenen Bussysteme eines Mikrorechners

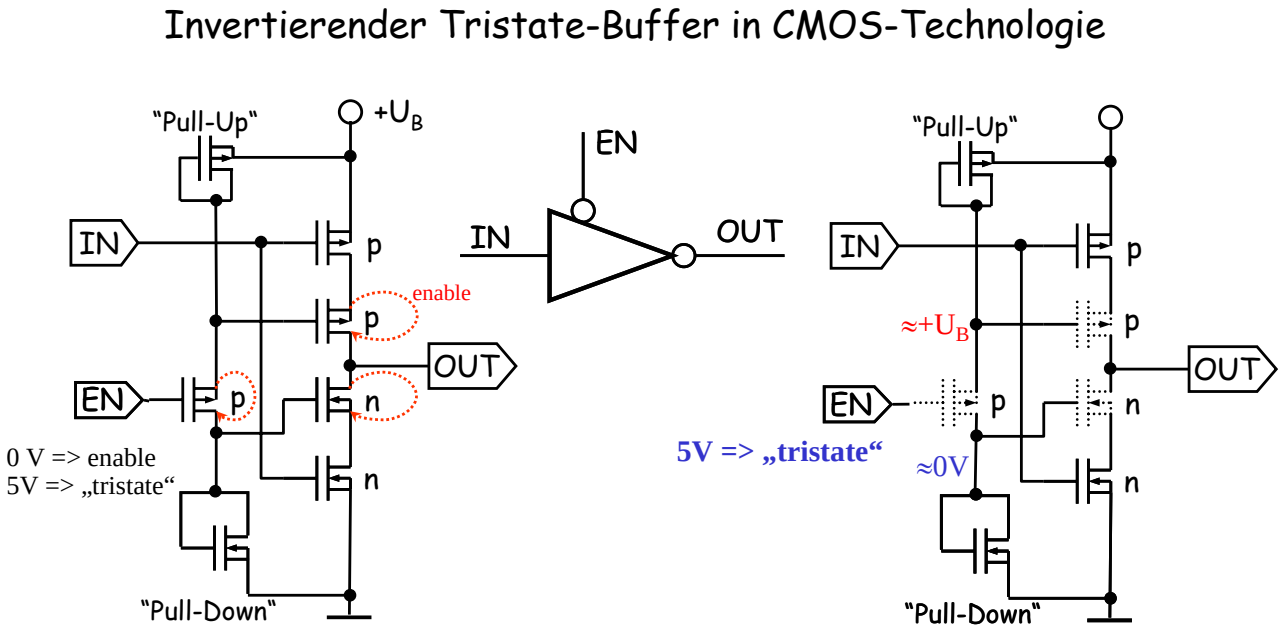



Bild 4.30: Aufbau eines CMOS–Tristate–Buffers

Tristate-Buffer haben wir bereits als wichtige Steuerwerksbestandteile des kennengelernt, ohne uns dort näher mit der elektrischen Funktionsweise zu befassen. Das ist in Bild 4.30 nachgeholt. Wenn der EN-Eingang auf ’0’-Pegel ist sind die markierten Transistoren leitend, so daß eine normale Inverterfunktion realisiert wird. Bei EN=5V (’1’-Pegel) hingegen sind sie wie man links im Bild leicht sieht gesperrt und isolieren sozusagen den Ausgang (OUT), so daß dieser „hochohmig“ erschient und von äußeren Signalen – ohne diesen einen Widerstand entgegenzusetzen - beliebig im zulässigen Betriebsspannungsbereich einer Digitalschaltung bewegt werden kann.

Invertierender
BUS-Transceiver
mit
z.B. N=8, 16...32

| EN | DIR | A→B | B→A |
|---|---|---|---|
| 0 | 0 | „E" | „T" |
| 0 | 1 | „T" | „E" |
| 1 | 0 | „T" | „T" |
| 1 | 1 | „T" | „T" |

„E" ⇒ Enable
„T" ⇒ Tristate

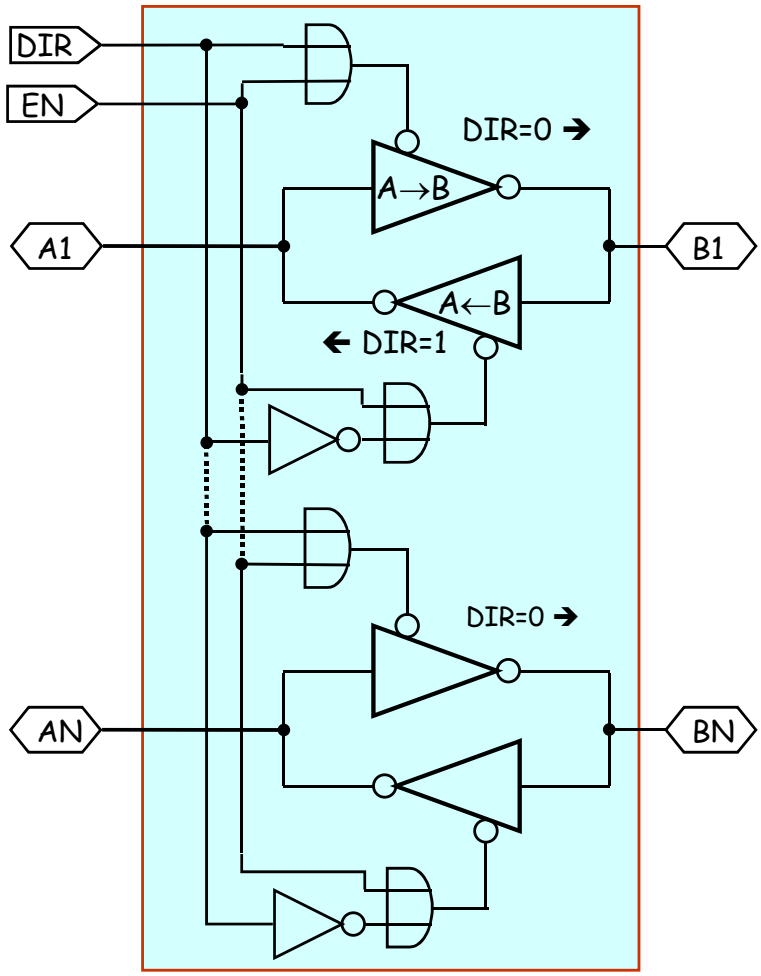



Bild 4.31: Der bidirektionale Bus–Transceiver als wichtiges Bauelement für den Datenaustausch zwischen Mikrorechnern und Speichern

#### 4.4.1.1 Schreib/Lesespeicher mit wahlfreiem Zugriff

- bilden die Arbeitsspeicher in Mikrorechnersystemen
- können sehr schnell gelesen und wieder beschrieben werden
- einzelne Speicherstellen können geändert werden
- Zahl der Lösch- und Schreibvorgänge ist praktisch unbegrenzt

Der Kernbereich einer CPU (Central Processing Unit ≡ Mikroprozessor) verfügt in der Regel über mindestens 8 Register, die jeweils ein Maschinenwort (8, 16, 32bit) oder gelegentlich auch das Doppelte davon fassen können.

Praktisch alle Rechen- und Datenverarbeitungsvorgänge nutzen Register:

Beispiel für einen einfachen Datentransfer MOV R1,A ;A nach R1 bringen

Beispiel für eine Addition ADD A,R2 ; A:=A+R2

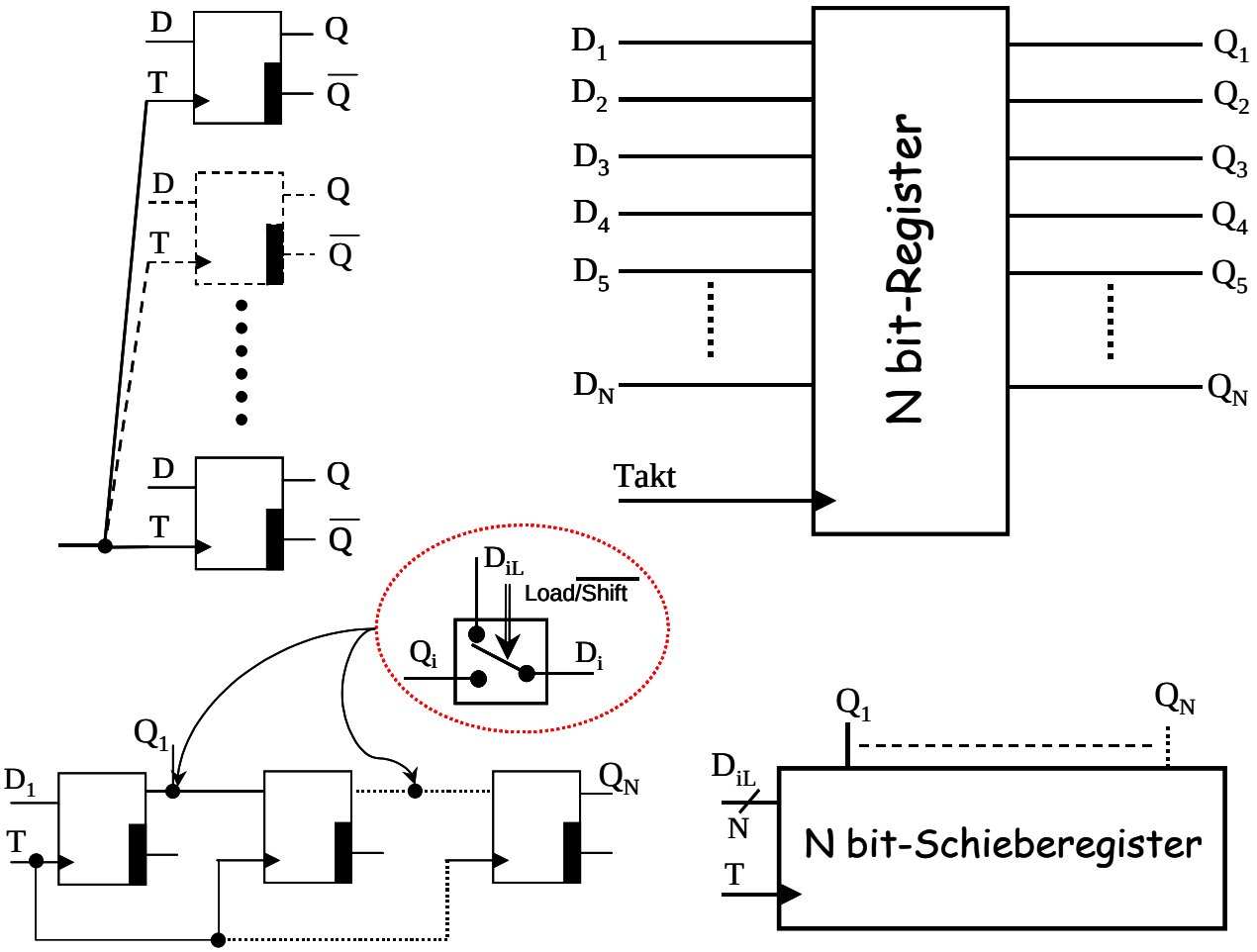



Bild 4.32: Der Aufbau von Registern

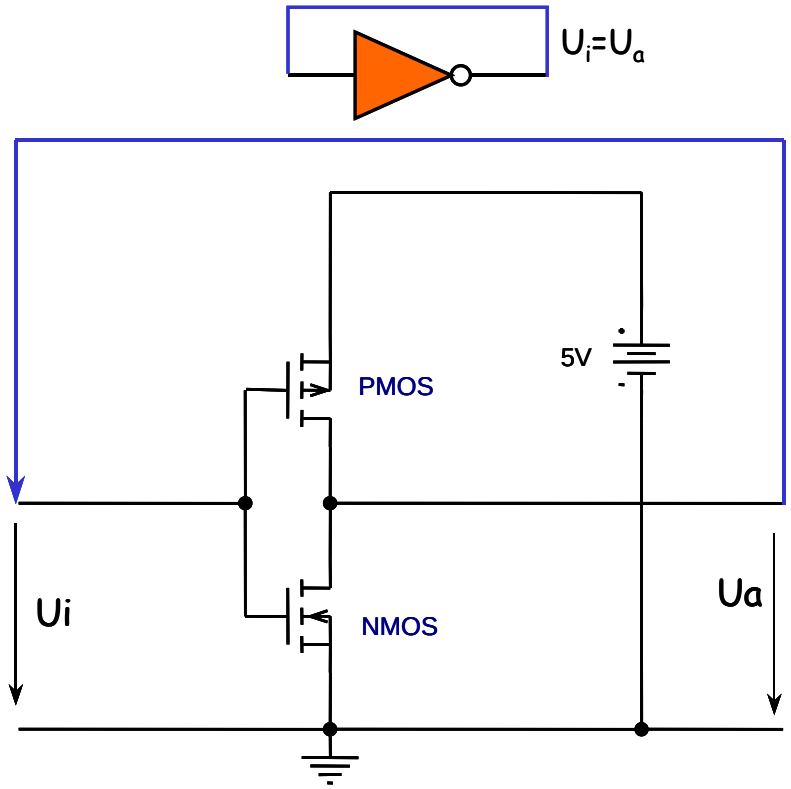



Bild 4.33 CMOS–Inverter mit direkter Rückkopplung

Da „registertaugliche“ Flip-Flops auch in Transfergatetechnik mindestens 16 Transistoren brauchen ist die Zahl der in dieser Form vorhandenen Speicherplätze relativ klein und wird auf das absolut Notwendige begrenzt.

Arbeitsspeicher und Datenspeicher von Mikrorechnern müssen weitaus einfacher aufgebaut sein, um die nötige Kapazität (gezählt in kBit oder MBit bzw. kByte oder MByte) bereitstellen zu können.

Ein wichtiger Schritt hin zum „Massenspeicher“ ist der Aufbau statischer RAM-Zellen.

Ausgangspunkt ist dabei der einfache CMOS–Inverter, nach Bild 4.35, der im Gedankenexperiment mit einer direkten Drahtverbindung vom Ausgang zum Eingang verse-

hen werden soll. Würde man dabei das mechanische Schaltermodell zugrunde legen, käme es zu einem zerstörenden Kurzschluß.

Wie wir sehen werden, liefert auch die elektronische Realisierung noch kein brauchbares Resultat. Denn anstatt eines Speichers, der zwei stabile und gut unterscheidbare Zustände annehmen sollte, bekommen wir einen Oszillator, den wir an dieser Stelle absolut nicht brauchen können.

Das folgende Bild 4.36 zeigt die Inverterkennlinie (rot) und die (triviale) Kennlinie der Rückkopplung $U_a = U_i$ (blau). Wie man sofort sieht, führt die grafische Lösung der zwei Gleichungen mit zwei Unbekannten hier zu einem Schnittpunkt in Bildmitte, der jedoch instabil ist. Eine geringfügige Auslenkung nach links (lila Pfeil) läßt $U_a$ auf die Betriebsspannung ($U_B$=5V) ansteigen. Dies wird ohne Verzögerung über die Rückkopplung an den Eingang weitergegeben ($U_i$=5V). Der Inverter bringt nun nach seiner Durchlaufverzögerung $\tau$ den Ausgang auf Null, was sich sofort wieder auf den Eingang fortpflanzt. Jetzt kippt der Inverterausgang wiederum nach der Durchlaufverzögerung $\tau$ auf $U_B$. Damit ist die dargestellte Schwingung angefacht.

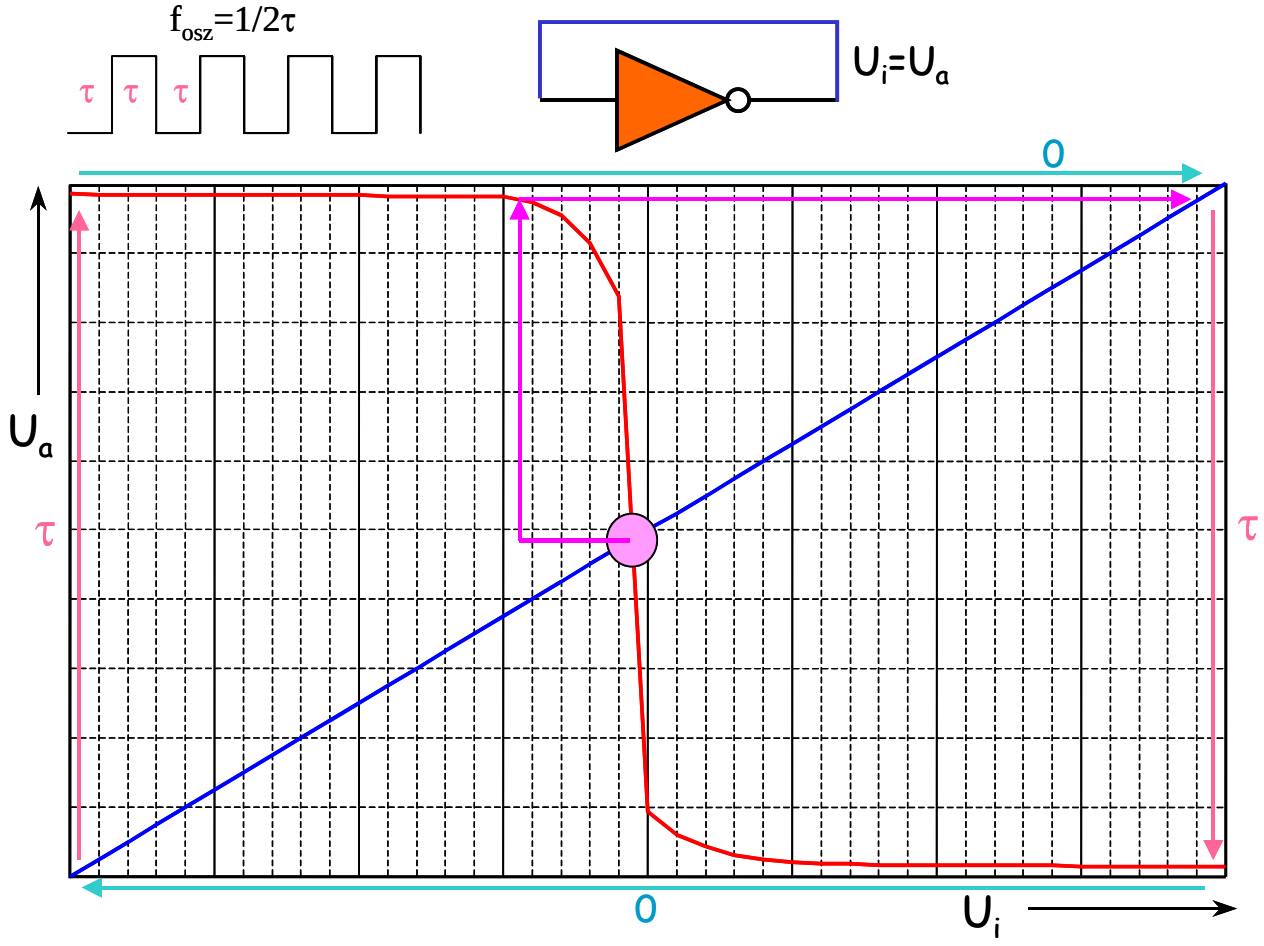



Bild 4.34: Ungewolltes Entstehen einer Schwingung beim rückgekoppelten Inverter

Eine Lösung ergibt sich jedoch durch Verwendung eines Buffers in Form von zwei aufeinanderfolgenden Invertern - s. Bild 4.35. Hierbei stellen sich genau die zwei gewünschten stabilen Arbeitspunkte ein.

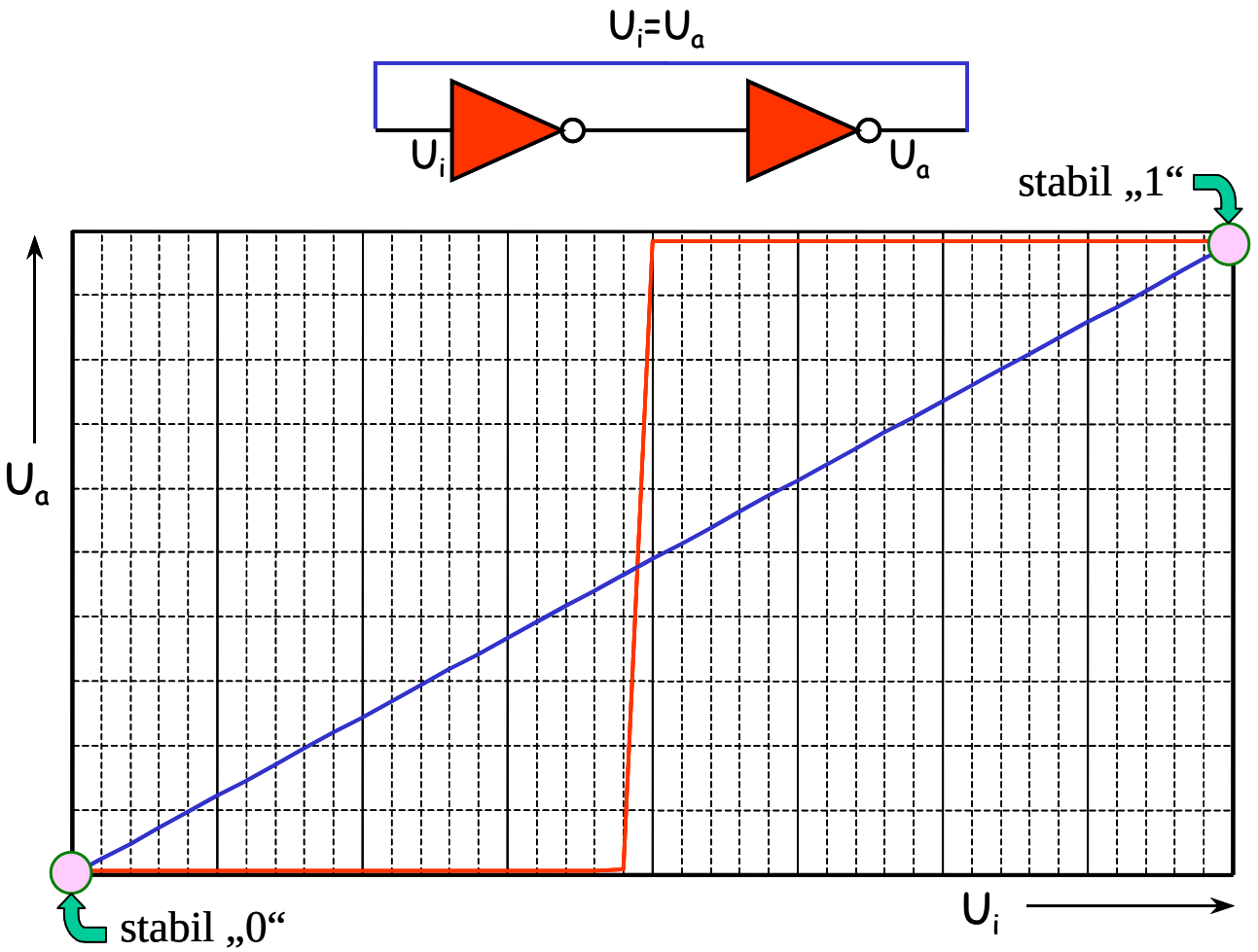



Bild 4.35: Lösung durch Rückkopplung eines Buffers aus zwei aufeinanderfolgenden Invertern

Bild 4.36 zeigt den Gesamtaufbau einer statischen RAM-Zelle, die im Kern eine Schaltung nach Bild 4.35 enthält. Ein wichtiges Prinzip ist die Symmetrie, die sowohl für den Lese- als auch für den Schreibvorgang vorliegt. Beim Schreiben muß die Speicherzelle zum Kippen gebracht werden, indem der „H“–Pegel führende Inverterausgang auf ca. $U_B/2$ heruntergezogen wird. Wie wir bei der Untersuchung des elektrischen Verhaltens eines CMOS–Inverters gesehen haben, ist dazu nicht besonders viel Strom erforderlich. Ein schneller Kippvorgang nach Maßgabe des zu schreibenden Datenbits kann mit Hilfe der Schreibbuffer ausgelöst werden.

Beim Lesen werden die beiden Ausgangbuffer im unteren Bildteil aktiviert, die das gespeicherte Bit normal und invertiert zur Verfügung stellen.

Die Realisierung auf Transistorebene wird anhand einer Powerpoint-Darstellung detailliert behandelt.

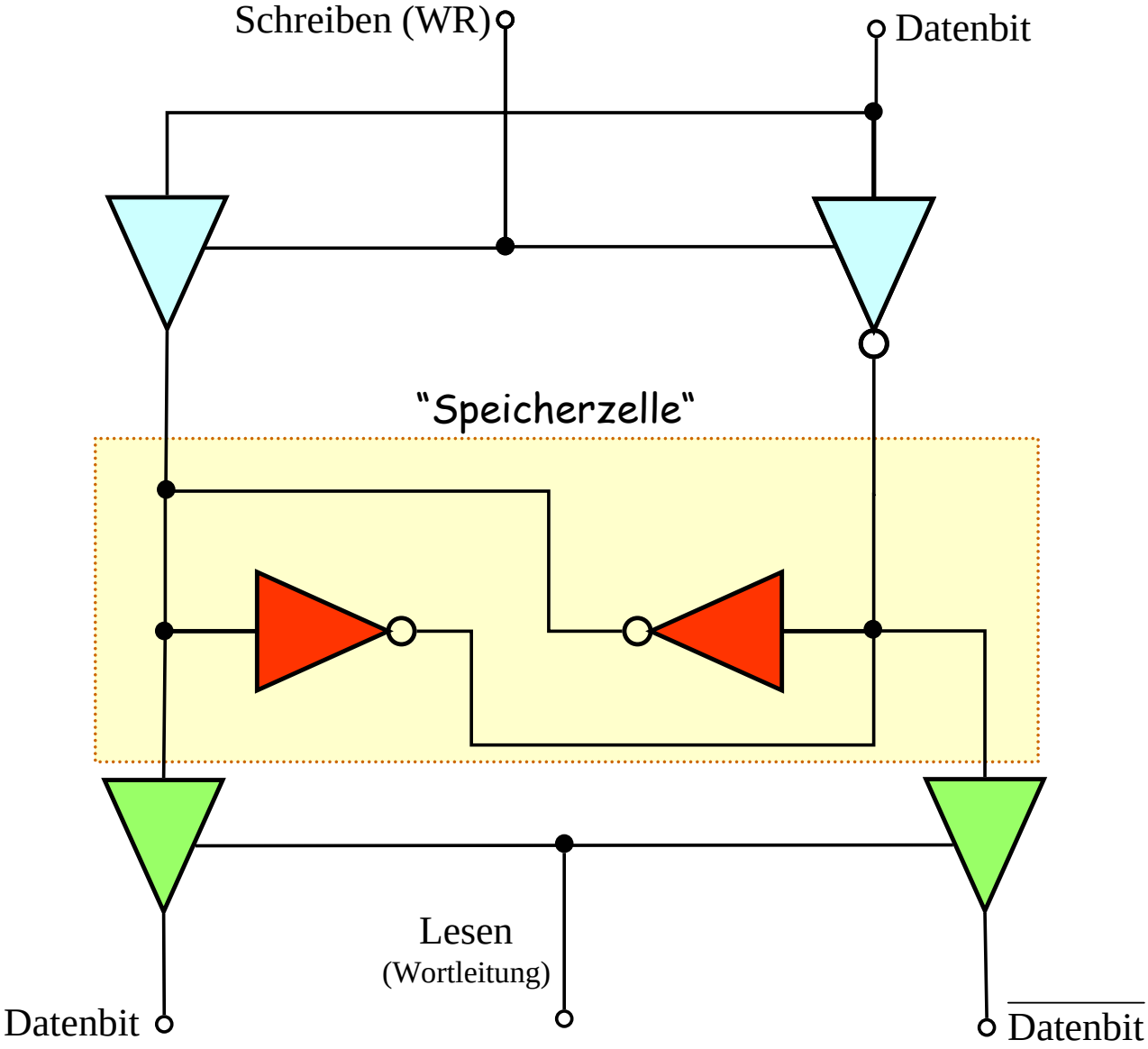



Bild 4.36: Blockschaltbild einer kompletten statischen RAM-Zelle

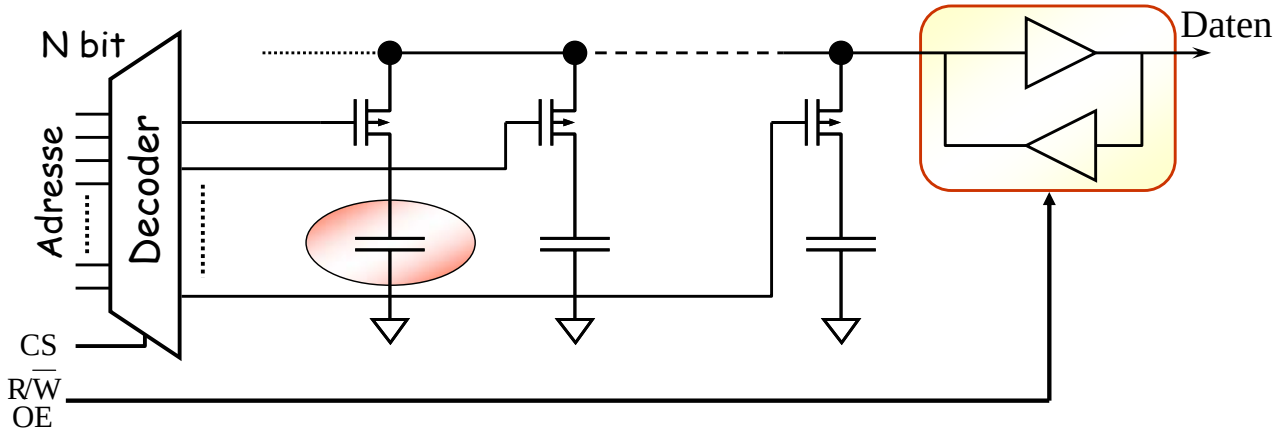



Bild 4.37: Funktionsprinzip dynamischer RAM-Zellen

Die dynamische RAM-Zelle ist der einfachste Speicher, der rein analog in Form eines Kondensators auf Silizium (z.B. Poly–Silizium - $SiO_2$ - Substrat) aufgebaut ist. Die Kapazität beträgt höchstens einige Femtofarad ($fF=10^{-15}F$). Bei 1 fF und bei einer Spannung von 3,3V sind immerhin noch ca. 20.000 Elektronen nötig, um einen „H"–Pegel zu speichern (Elektronenladung: $e \approx 1{,}6 \cdot 10^{-19}$ C). Beim Lesen wird die Information zerstört und muß deshalb nach Regenerierung sofort wieder eingeschrieben werden. Das besorgt die in Bild 4.37 angedeutete bidirektionale Verstärkeranordnung, in der der Zwischenspeicher nicht eingezeichnet ist. Dadurch, daß die Kapazität der Bitleitung viel größer als die eines Speicherkondensators ist, müssen sehr kleine Pegeländerungen erkannt werden.

Wegen des nichtidealen Schaltverhaltens der Lesetransistoren, die in gesperrtem Zustand „Leckströme" aufweisen, verlieren die Speicherkondensatoren Ladung und müssen daher regelmäßig in bestimmten Zeitabständen nachgeladen werden. Diese Regenerierung wird „Refresh" genannt und muß ständig aktiv sein, soll aber den normalen Speicherbetrieb nicht einschränken.

Moderne DRAM–Zellen müssen nur ca. alle 60ms aufgefrischt werden. Was das physikalisch für die „Transistorqualität" bedeutet, macht ein Zahlenbeispiel klar:

Unter der Annahme, das der Abfluß der halben Ladung einen Speicherverlust zur Folge hätte, dürften höchstens $1{,}6 \cdot 10^{-15}$C (10.000 Elektronen) in 60ms „verschwinden". Der zulässige Leckstrom muß dafür unter 0,03pA bleiben!

Das Auffrischen einer Zelle geht sehr schnell, d.h. in weniger als einer Mikrosekunde (auch schon in „alten" Technologien). Bei einem 64k-RAM käme man bei bitweisem Refresh mit rund 64ms aber schon in Bedrängnis. Deshalb läuft der Vorgang in der Regel zeilenweise ab, so daß bei einer Organisation 256x256 insgesamt weniger als 256µs benötigt werden. Lesen und Schreiben dürfen durch den Refresh-Vorgang nicht behindert werden. Das ist problemlos möglich, da Lesen und Schreiben automatisch die Regenerierung bewirken, so daß angesprochene Zellen vom Refresh ausgenommen werden können. Bei heutigen DRAMs ist die komplette Refresh–Logik inklusive Timing mitintegriert, so daß der Anwender sich nur noch um das zeitrichtige Anlegen der äußeren Signale (RAS, CAS, $R/\overline{W}$, OE) kümmern muß.

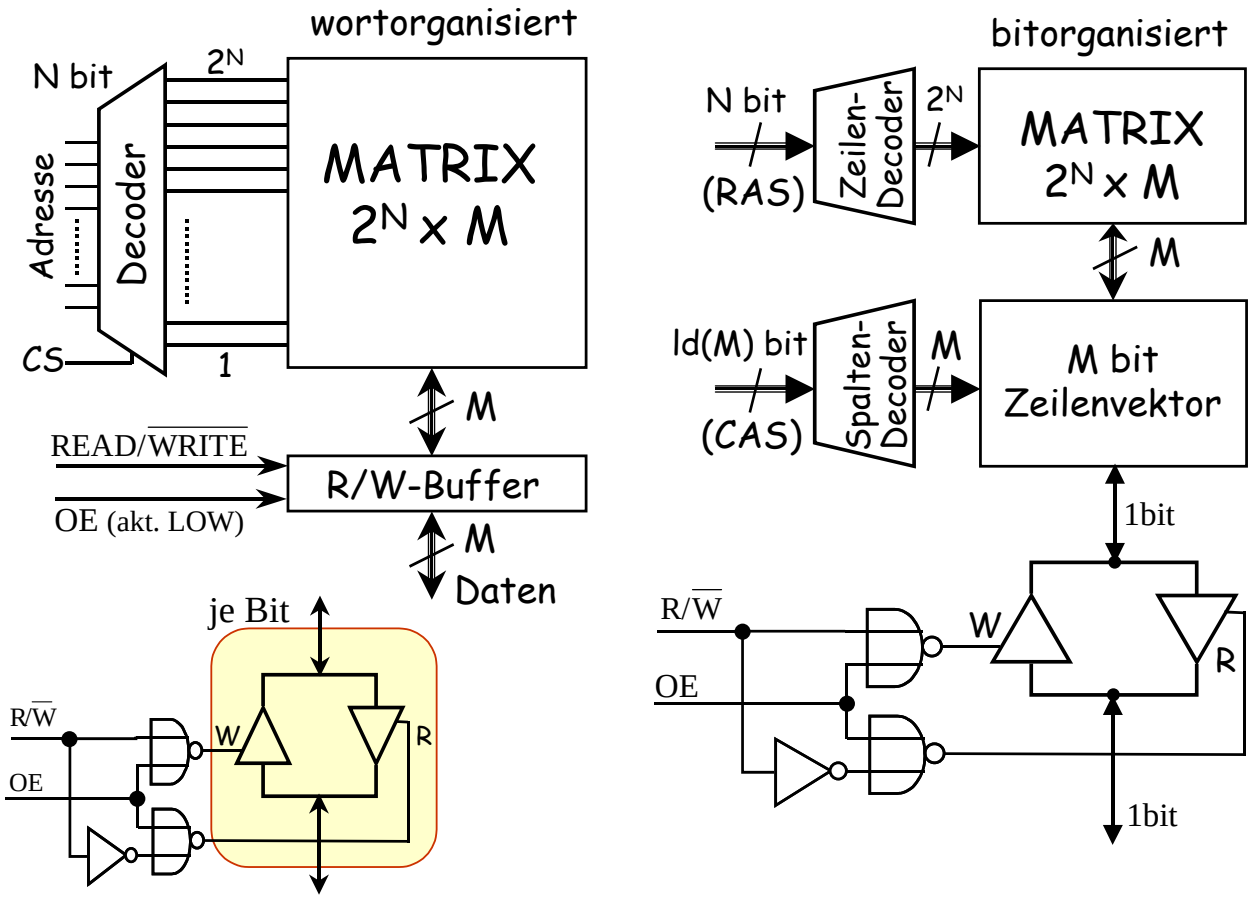



Bild 4.38: Übliche Organisationsformen von Speichern

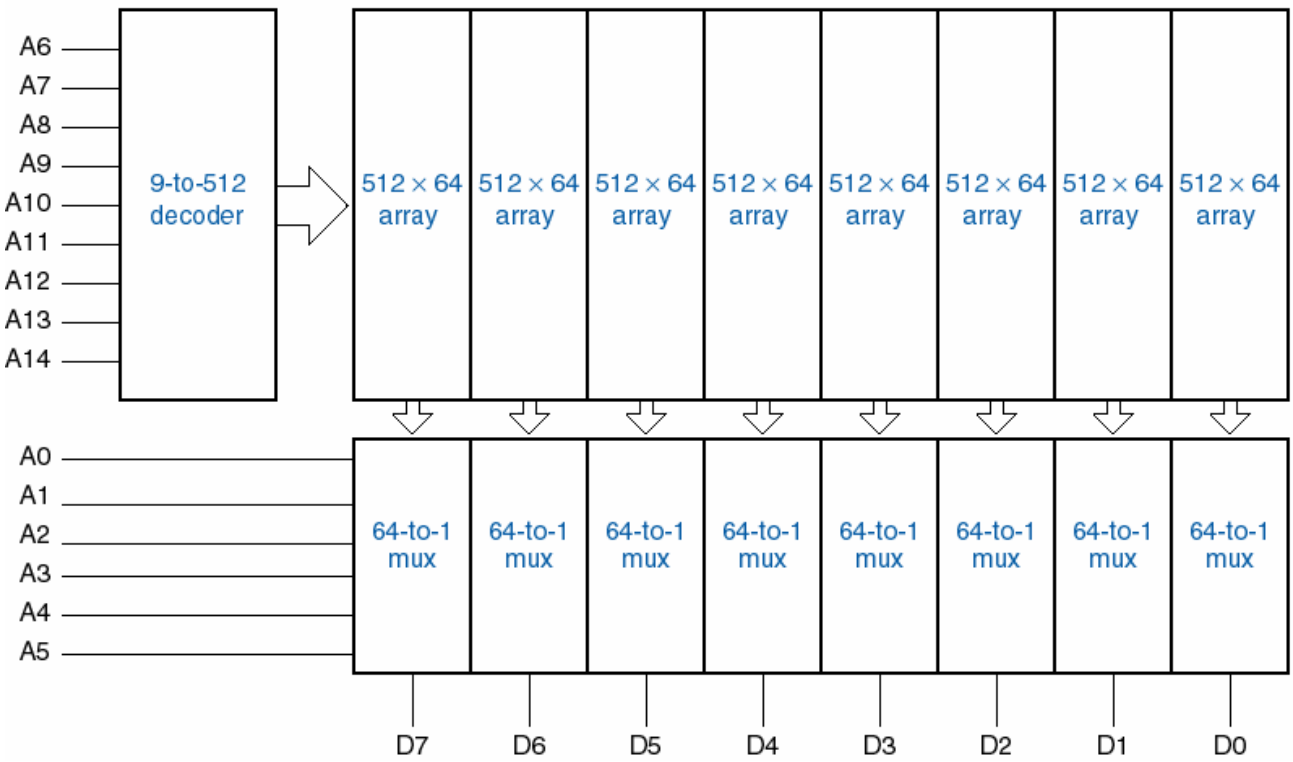
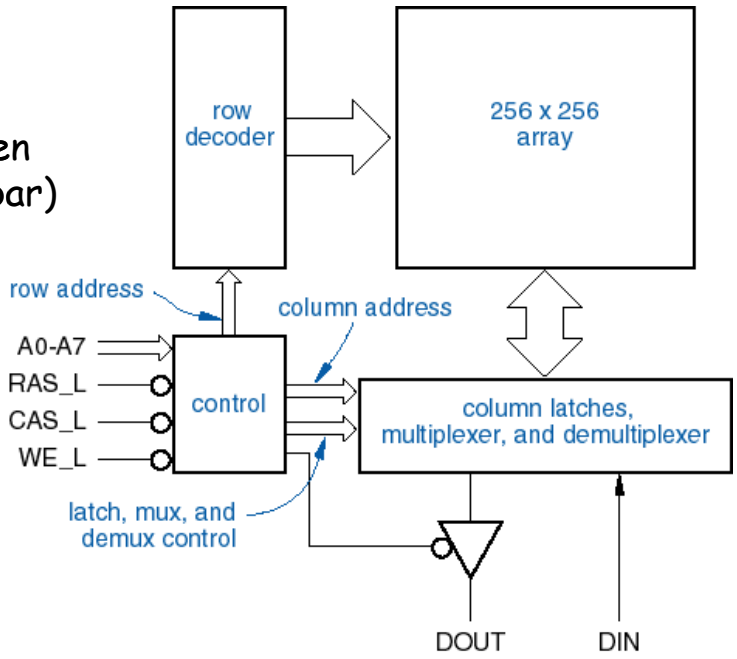



Bild 4.39: Beispiel der Block-Organisation eines 32Kx8 Speichers: Je 8 Blöcke (oben) liefern jeweils 512 „Worte“ der Länge 64 bit, die unten in jeweils 64 Worte der Länge 8 bit zerlegt und ausgegeben werden

Beispiel für den Aufbau eines dynamischen RAMs mit den zum Betrieb benötigten äußeren Signalen (Refresh nicht sichtbar)




Zur Erläuterung des Refresh-Vorgangs bei dynamischen RAMs

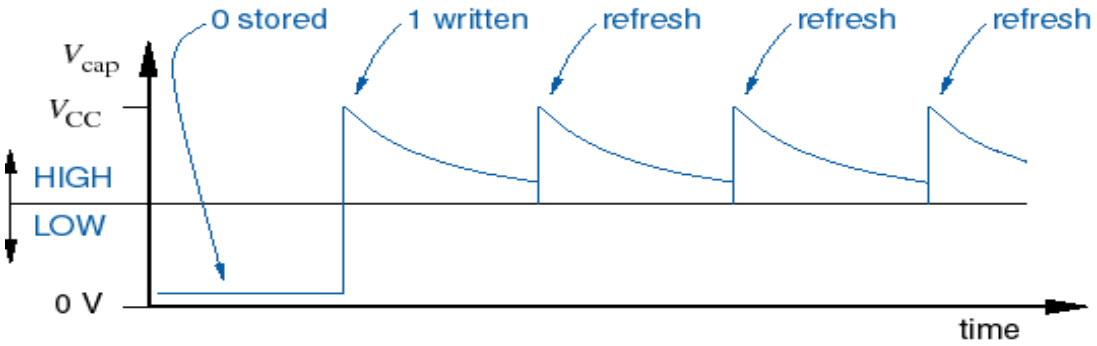



Bild 4.40: Signale für den Zugriff auf ein dynamisches RAM und Erläuterung des Refreshvorgangs

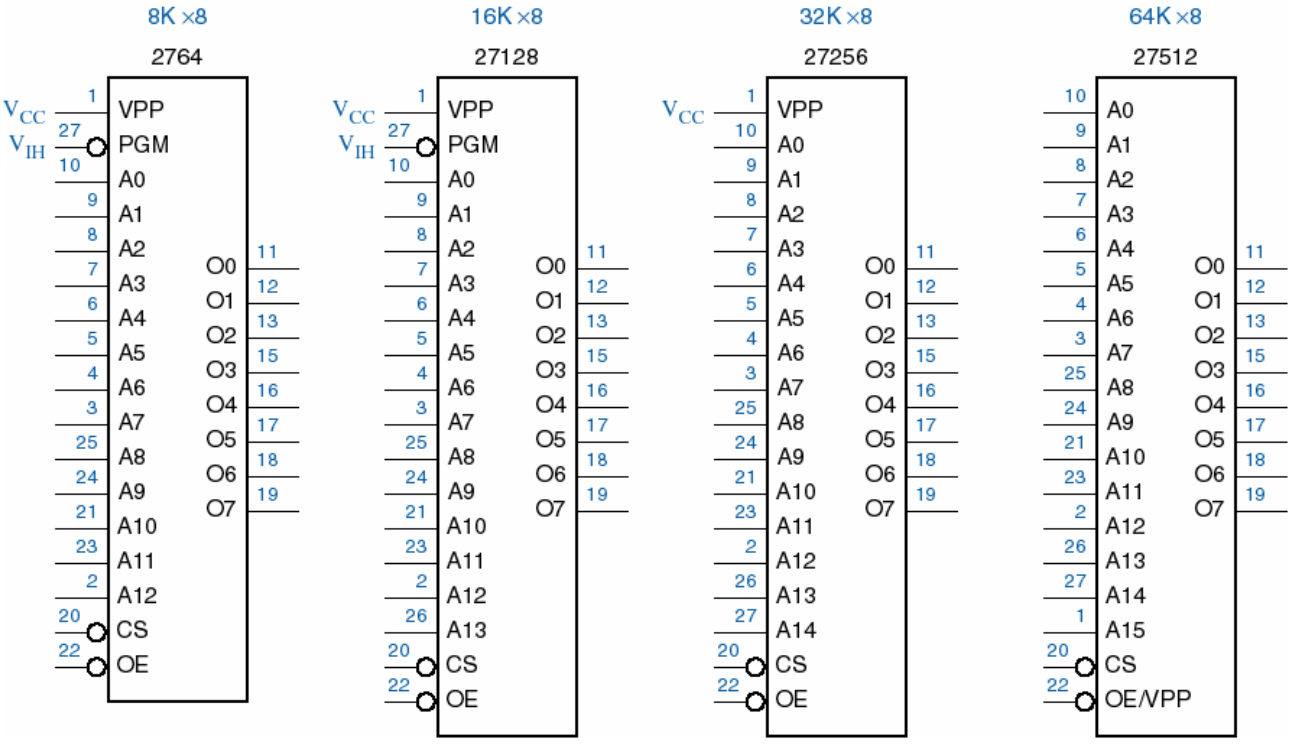



Bild 4.41: Beispiele für den Aufbau und die Pinbelegung handelsüblicher EPROM–Bausteine

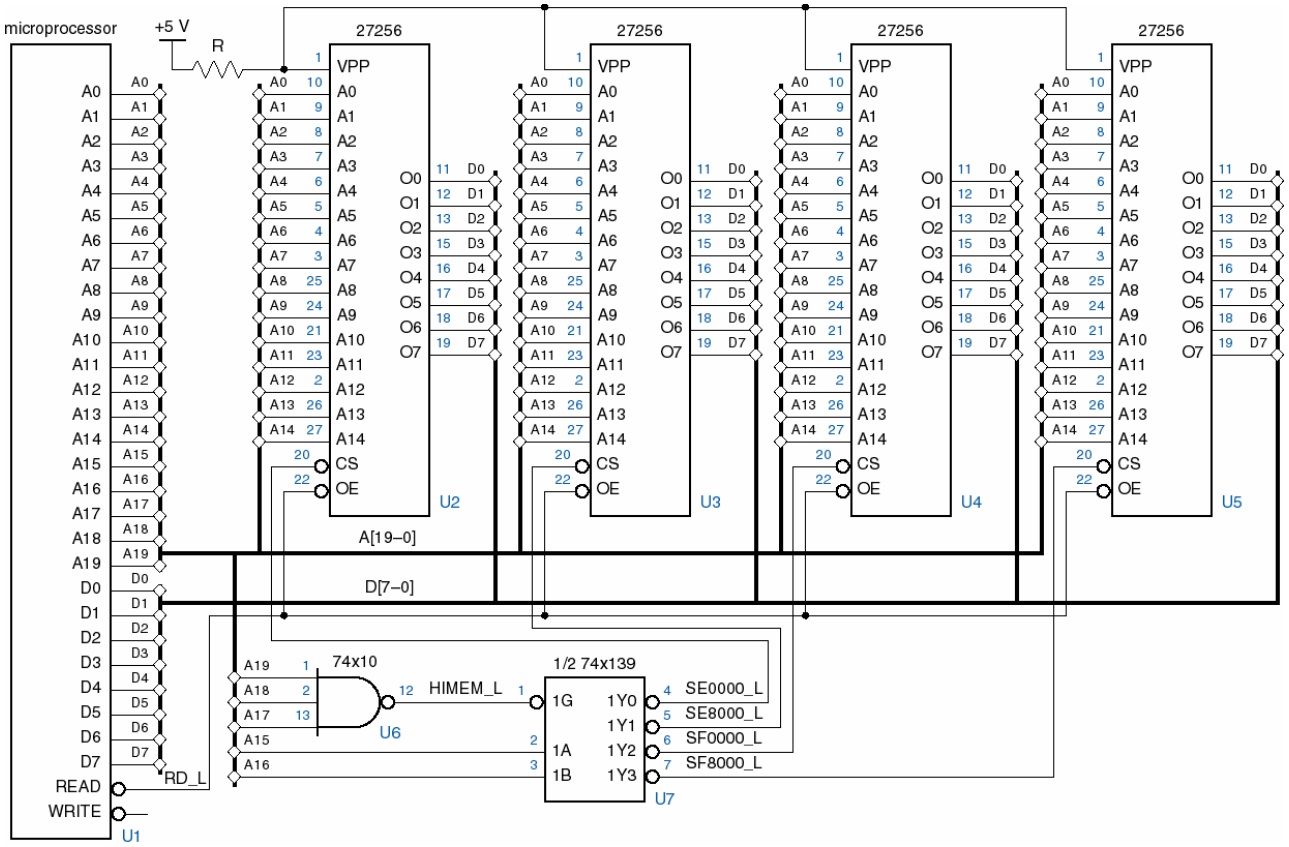



Bild 4.42: Festwertspeicherorganisation in einem Mikrorechnersystem mit EPROM–Bausteinen

Die Koordination von Speicherzugriffen zählt zu den komplexesten und störanfälligsten Aufgaben in Mikrorechnersystemen. Grundsätzlich wird zwischen Programm- und Datenspeicherzugriffen unterschieden. Während aus einem Programmspeicher stets nur gelesen wird, muß ein Datenspeicher sowohl das Lesen als auch das Schreiben zulassen. Bei jeder Art eines Speicherzugriffs ist zunächst das Anlegen einer Adresse erforderlich, die der Mikrorechner generell über seinen unidirektionalen Adreßbus (A0...A19 in Bild 4.42) zur Verfügung stellt. Das ist von zeitlichen Ablauf her gesehen immer der erste Vorgang. Gleichzeitig oder auch etwas später geht bei einem lesenden Zugriff – den wir hier zunächst anhand von Bild 4.42 untersuchen - der Datenbus (D0...D7) auf Mikrorechnerseite in den hochohmigen Zustand (Tristate). Jetzt erfolgt die Aktivierung der Ausgangstreiber des Speichers über CS (Chip Select) und OE (Output Enable)[12]. Diese Signale kommen entweder direkt vom Steuerbus des Mikrorechners (z.B. READ=OE in Bild 4.42) oder werden mittels eine Adreßbereichsdecoders erzeugt. Mit seinem 20bit breiten Adreßbus kann der Mikroprozessor im obigen Beispiel 100000H= $1.048.576_d$ Speicherstellen zu je 8bit adressieren. Das ist genau das 32-fache dessen was einer der dargestellten EPROM-Speicherbausteine vom Typ 27256 fassen kann – hier sind aber nur 4 davon vorgesehen. Jeder davon faßt gemäß seiner Adreßbusbreite $2^{15}$= 8000H Byte= 32.768Byte oder 262144bit. Die Bezeichnung ..256 kommt von der allgemein üblichen, aber etwas schlampigen Schreibweise, die 1024 bit als 1k Byte deklariert. 32k Byte sind in diese Schreibweise also tatsächlich 32·1024=32.768 Byte, was eben 262144 bit entspricht.

Im Beispiel wird also nicht der gesamte Speicherbereich genutzt, sondern nur bis $2^{17}$=20.000H, was einer Adreßbusbreite von A0...A16 entspricht. Unter dieser Voraussetzung arbeitet der 4:1-Multipexer U7 vom Typ 74x136 in Bild 4.42 als Adreßbereichsdekoder für 4 Bereiche einer Größe von jeweils 8000H. Die oberen drei Adreßbits A17, A18 und A19 sind also stets fest auf ’1’ und

[12] die **o**-Symbole an Eingängen bedeuten, das ein ’0’-Pegel eine Aktivierung bewirkt (engl.: Active Low)

liefern so über das 3-fach-NAND U6 den erforderlichen ‚'0'-Pegel für den Enable-Eingang 1G des Multiplexers, während die Adreßbits A15 und A16 die vier Bereiche der Größe 8000H selektieren.

Der erste dieser Bereiche beginnt demnach bei $7 \cdot 2^{17}$=E0000H (wegen A17, A18 und A19='1'), d.h. mit A15='0' und A16='0' wird der Ausgang 1Y0 von U7 '0' und selektiert damit das EPROM U2. Für die drei übrigen U3, U4 und U5 muß das Chip Select (CS) unbedingt '1' bleiben, da es ansonsten zu eine Buskonflikt käme, wobei mehrerer treibende Bausteine gleichzeitig versuchen würden die Zustände am Datenbus aktiv zu beschreiben. Die Schaltung in Bild 4.42 schließ dies definitiv aus. Wie man leicht erkennt wird für A15,A16='10' der Bereich E8000H... EFFFFH und schließlich für A15,A16='11' der Bereich F8000H...FFFFFH, der im Baustein U5 liegt, über 1Y3 angesprochen.

Im weiteren werden etwas komplexere Verfahren des Speicherzugriffs untersucht, wie sie bei Standard-Mikrocontrollern häufig vorkommen. Da Ein-/Ausgangsleitungen immer eine knappe Ressource darstellen wird hier Daten/Adreßmultiplex angewandt, um Leitungen einzusparen. Ein solcher Multiplexvorgang erschwert das Verständnis der Abläufe ein wenig und wird daher ausführlich behandelt.

## 4.4.2 Timing-Diagramme für den Speicherzugriff

### 4.4.2.1 Programmspeicher

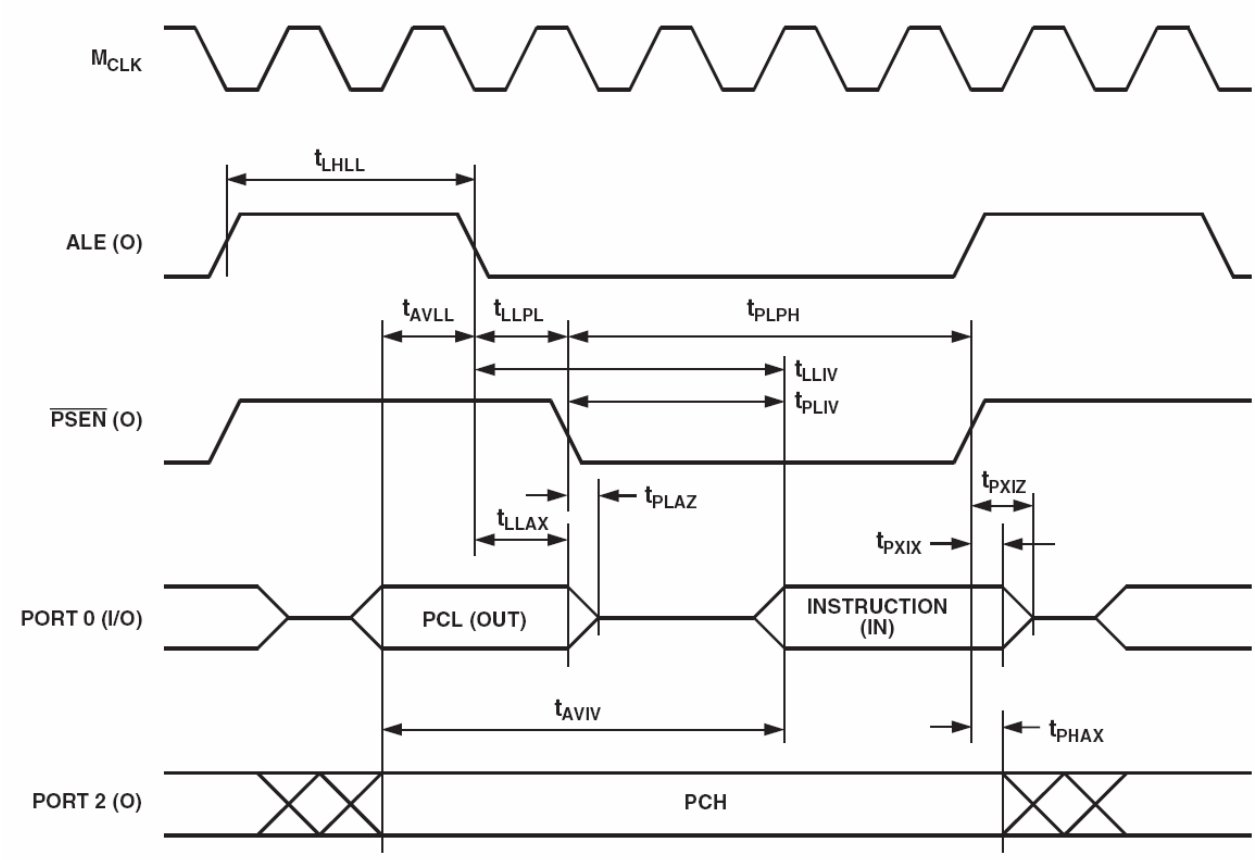



Bild 4.43: Lesevorgang aus dem Programmspeicher bei 8051-basierten Mikrocontrollern mit Daten/Adreßmultiplex an Port 0 (Datenbus) und Ausgabe der höherwertigen Adreßbits an Port 2.

Wir beginnen mit Zugriffen auf den Programmspeicher, die bekanntlich stets nur lesend sind. Es liegt ein 8bit-Mikrocontroller vor, der einen 16bit-Adreßbus verwendet, der jedoch nicht in Form von 16 parallelen Adreßleitungen vorliegt, sondern alle Ports sind nur 8bit breit. Es wird daher eine

Aufteilung des Adreßbusses in 8 niederwertige (PCL[13]) und 8 höherwertige (PCH[14]) Bits vorgenommen, die über verschiedene Ports nicht gleichzeitig, sondern in einer bestimmten zeitliche Abfolge ausgegeben werden. Aufgrund dieser sequentiellen Ausgabe wird eine Speicherung erforderlich, um die gesamte Adresse beim Zugriff zur Verfügung zu haben.

Wie man in Bild 4.43 sieht, werden die unteren 8bit der Adresse (PCL) für eine bestimmte Zeit an Port 0 (der auch den bidirektionalen Datenbus darstellt) ausgegeben, während PCH konstant über die gesamte Dauer des Zugriffs an Port 2 verfügbar ist. Wie man des weiteren sieht, steht ein Hilfssignal mit der Bezeichnung ALE (Address Latch Enable) zur Verfügung, dessen Rückflanke ziemlich genau in der Mitte des Zeitbereichs liegt, für den PCL am Port 0 verfügbar ist. Wie wir etwas später anhand von Bild 4.49 noch genauer sehen, kann diese Rückflanke für die Zwischenspeicherung der 8 niederwertigen Adreßbits in einem 8-bit-Latch verwendet werden. Ab dieser Rückflanke steht der 16bit Adreßbus vollständig und parallel an den Programmspeicher-eingängen zur Verfügung, so dass dessen Ausgangstreiber aktiviert werden können, um den Datenbus mit dem Inhalt der adressierten Speicherstelle zu belegen. Dazu geht der Bus (Port 0) in Tristate und das Steuersignal $\overline{\text{PSEN}}$ (**P**rogramm **S**torage **EN**able) gelangt an die Ausgangstreiber (Buffer) des Programmspeichers. Nach einer gewissen Zeit, die im Speicherbaustein für die Adreßdekodierung, d.h. das Ansprechen der selektierten Speicherstelle und das Einschalten des Ausgangbuffers benötigt wird, erscheinen am Bus (Port 0) die in den Mikrorechner einzulesenden Daten - hier mit INSTURUCTION bezeichnet. Das Einlesen erfolgt exakt mit der Vorderflanke von $\overline{\text{PSEN}}$. Danach kann unmittelbae eine neuer Lesezyklus gestartet werden.

Beim Ansprechen externer Datenspeicher darf $\overline{\text{PSEN}}$ nicht aktiviert werden, und es muß zwischen Lesen und Schreiben unterschieden werden. Zu diesem Zweck stehen zwei weitere Steuersignale $\overline{\text{RD}}$ (für read ≡ Lesen) und $\overline{\text{WR}}$ (für write ≡ Schreiben zur Verfügung. Beide sind „active low“, so daß sich ein ganz ähnliches Timing wie beim Programmspeicherzugriff ergibt. Die genauen Voränge werden anhand von Bild 4.44 und Bild 4.45 erläutert.

Als ’kleine’ Erhöhung der Komplexität haben wir es hier mit einem zweifachen Adreßmultiplex zu tun, aus dem schließlich ein 24-bit-Adresse hervorgeht. Das schon beschreiben Prinzip bleibt dabei gleich, d.h. es findet jetzt parallel ein Multiplex an Port 0 und Port 2 statt. Mit der Rückflanke von ALE müssen jetzt die Adreßbits A0...A7 und A16...A23 jeweils in ein 8-bit-Latch übernommen werden. Kurz danach stehen an Port 2 die restlichen Adreßbits A8...A15 an, so daß die komplette 24-bit-Adresse nun verfügbar ist. Die weiteren Abläufe ergeben sich analog zu den bereits für den Programmspeicherzugriff erläuterten.

[13] diese Abkürzung steht für ’Program Counter Low’ Byte

[14] entsprechend ’Program Counter High’ Byte

#### 4.4.2.2 Datenspeicher lesen

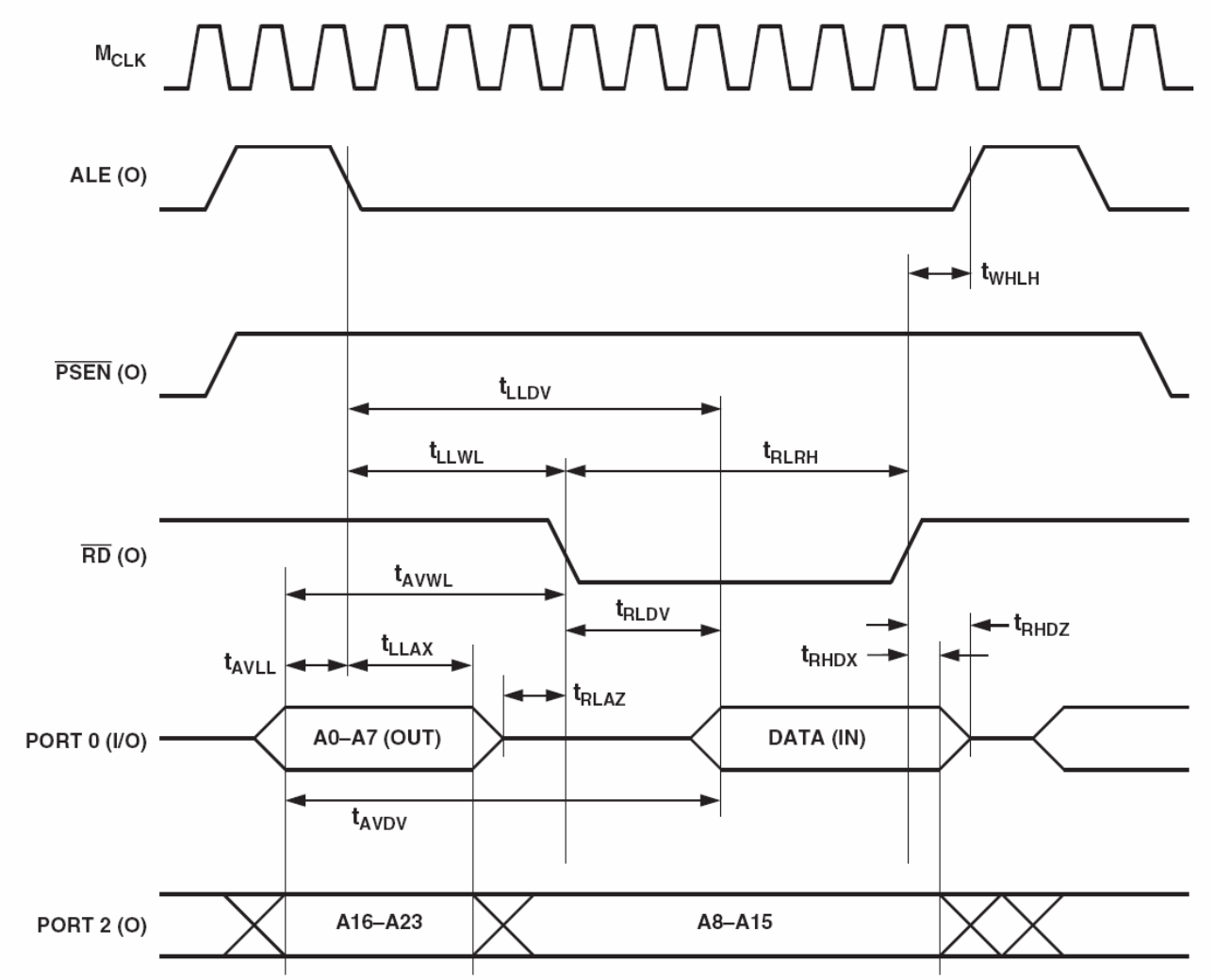



Bild 4.44: Impulsdiagramm für das Lesen aus einem Datenspeicher bei 8051-basierten Mikrorechnern mit auf 24 bit erweitertem Adreßbus

Wie man sieht, übernimmt nun $\overline{RD}$ die Rolle von $\overline{PSEN}$ und aktiviert mit seiner Rückflanke die Ausgangstreiber des Datenspeichers, der dann die Pegel am Bus (Port 0) bestimmt. Das Einlesen der Daten vom Speicher geschieht mit der Vorderflanke von $\overline{RD}$.

#### 4.4.2.3 Datenspeicher beschreiben

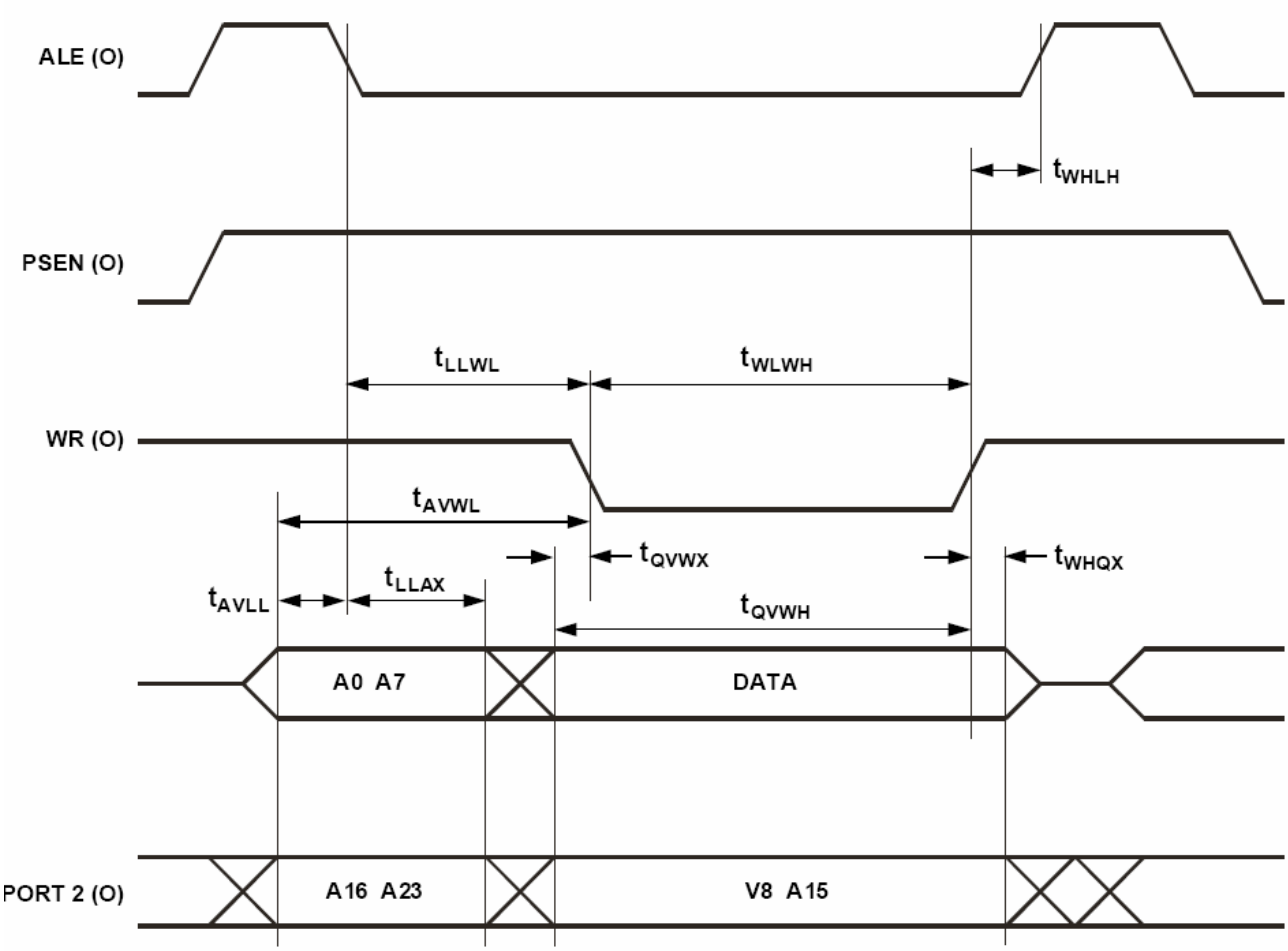



Bild 4.45: Impulsdiagramm für das Beschreiben eines Datenspeichers bei 8051-basierten Mikrorechnern mit auf 24 bit erweitertem Adreßbus

Das Schreiben in einen externen Datenspeicher erfolgt mit Hilfe des Steuersignals $\overline{WR}$ . Der zeitliche Ablauf ist in Bezug auf das Ansprechen des Speichers identisch mit dem Lesevorgang. $\overline{PSEN}$ muß auch hier natürlich inaktiv, d.h. auf ’1’-Pegel bleiben.

Ungefähr mit der Rückflanke von $\overline{WR}$ werden aber jetzt die zu schreibenden Daten am Bus (Port 0) vom Mikrocontroller zur Verfügung gestellt und können nach einer gewissen Zeit, die der Datenspeicher zum Selektieren der zu beschreibenden Stelle benötigt, mit der Vorderflanke von $\overline{WR}$ übernommen werden.

Die beiden folgenden Grafiken Bild 4.46 und Bild 4.47 veranschaulichen zusammenfassend die Entstehung von kompletten Mikrorechnersystemen, wobei über Bussysteme externe oder interne Daten- und Programmspeicher angeschlossen sein können. Im nächsten Abschnitt werden „reale“ Mikrorechner näher behandelt, die heutigen „Industriestandards“ entsprechen. Aus Gründen der Übersicht werden ausschließlich 8bit-Systeme betrachtet, die jedoch je nach Ausführung heute sehr umfangreiche und komplexe Peripheriekomponenten beinhalten können.

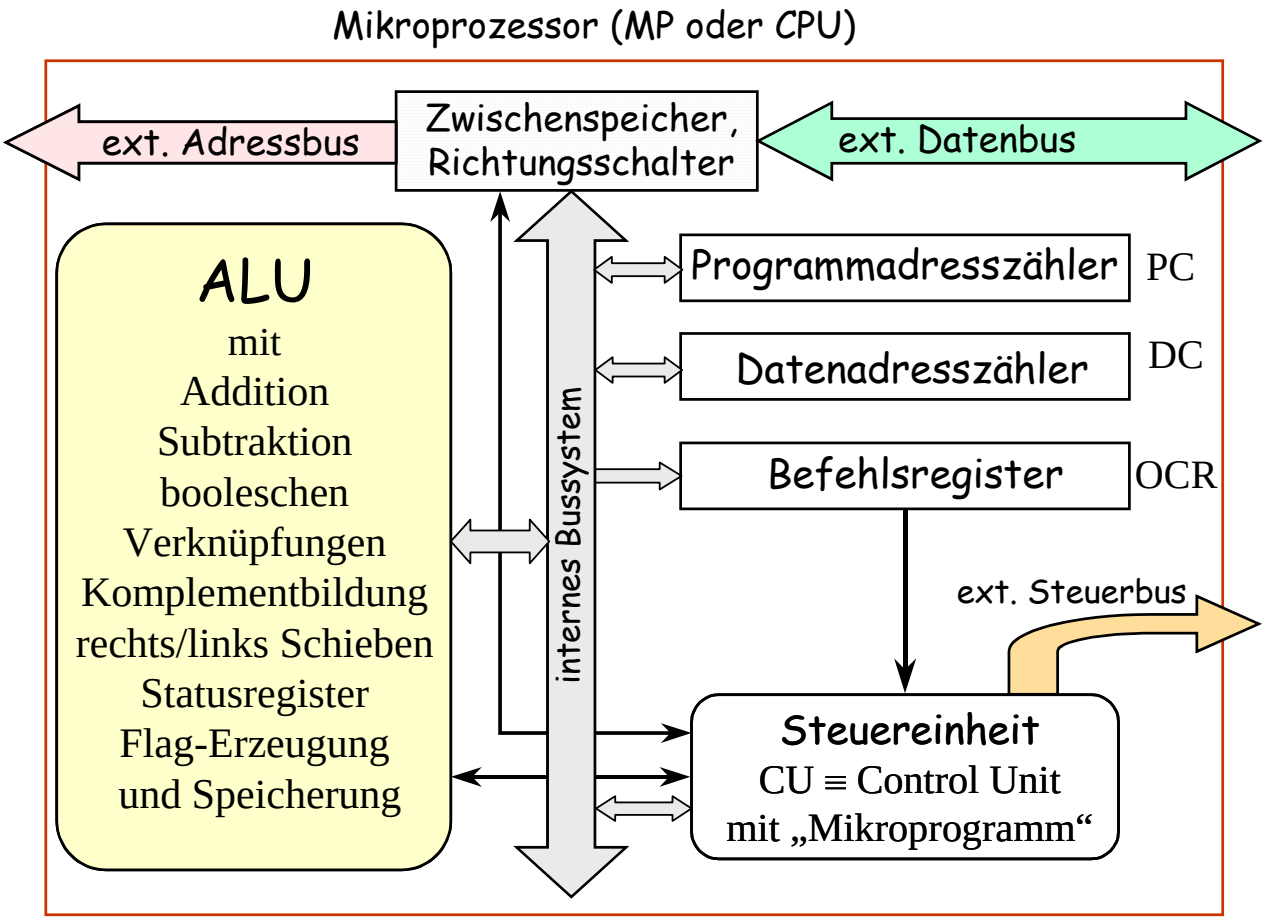



Bild 4.46: Typische Architektur eines Mikroprozessors

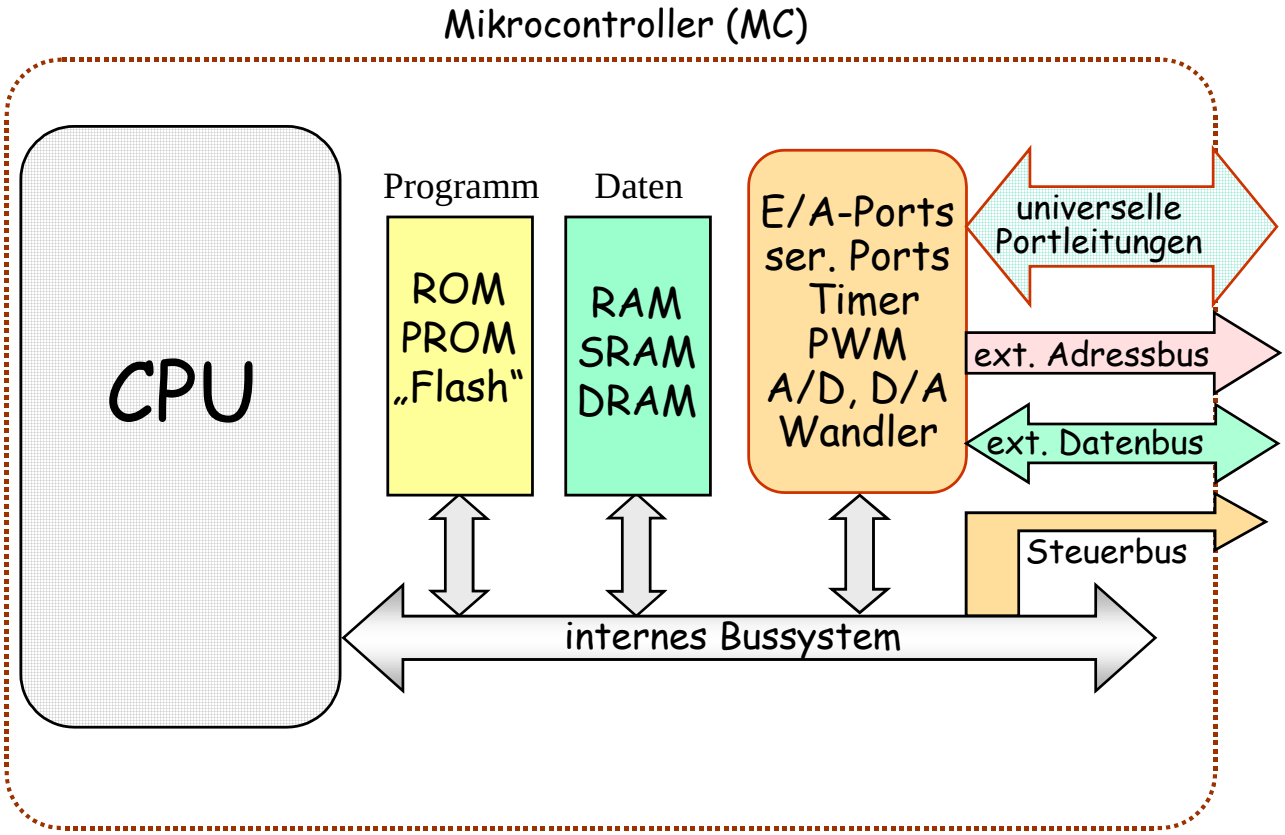



Bild 4.47: Der Mikrocontroller mit Speichern als vollständiger Mikrorechner auf einem „Chip“

## 4.5 Standard-Mikrorechner

### 4.5.1 Beispiele mit 8051/52-Mikrocontrollerkern

- 8-Bit CPU
- 4K-Byte ROM oder EPROM beim 8751
- 128/256 Byte RAM
- 21 Spezialfunktionsregister
- 32 Ein/Ausgabeleitungen
- 64K Adreßraum für externe Programmspeicher
- 64K Adreßraum für externe Datenspeicher
- Zwei 16-Bit Zähler/Zeitgeber
- Interruptsystem mit 5 Quellen und 2 Prioritätsebenen
- Serielle Schnittstelle für Vollduplexbetrieb
- Einzelbit Adressiermöglichkeiten
- Multiplikations- und Divisionsbefehle
- Integrierter Oszillator

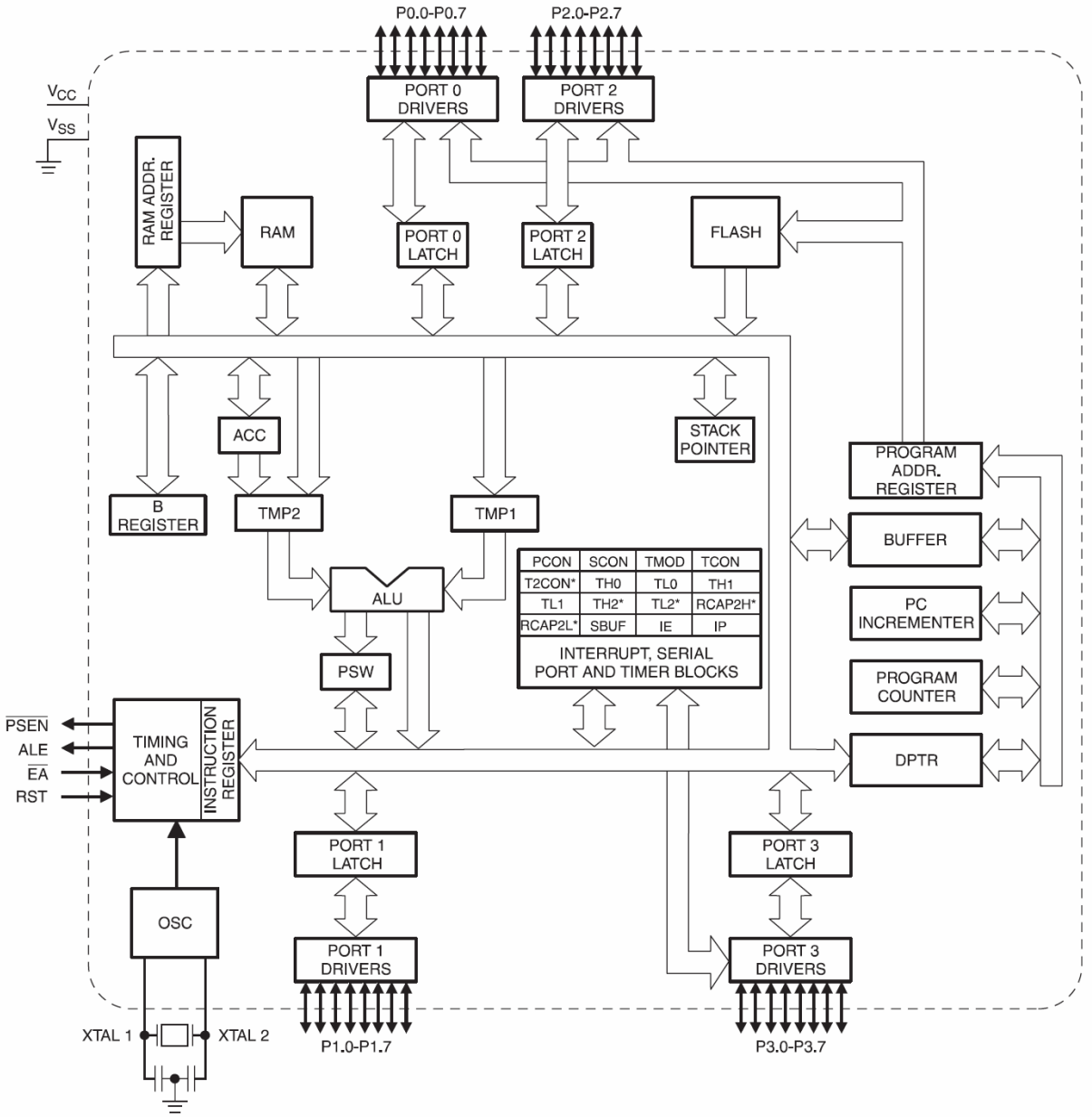



Bild 4.48: Blockschaltbild des 8051/52-Rechnerkerns

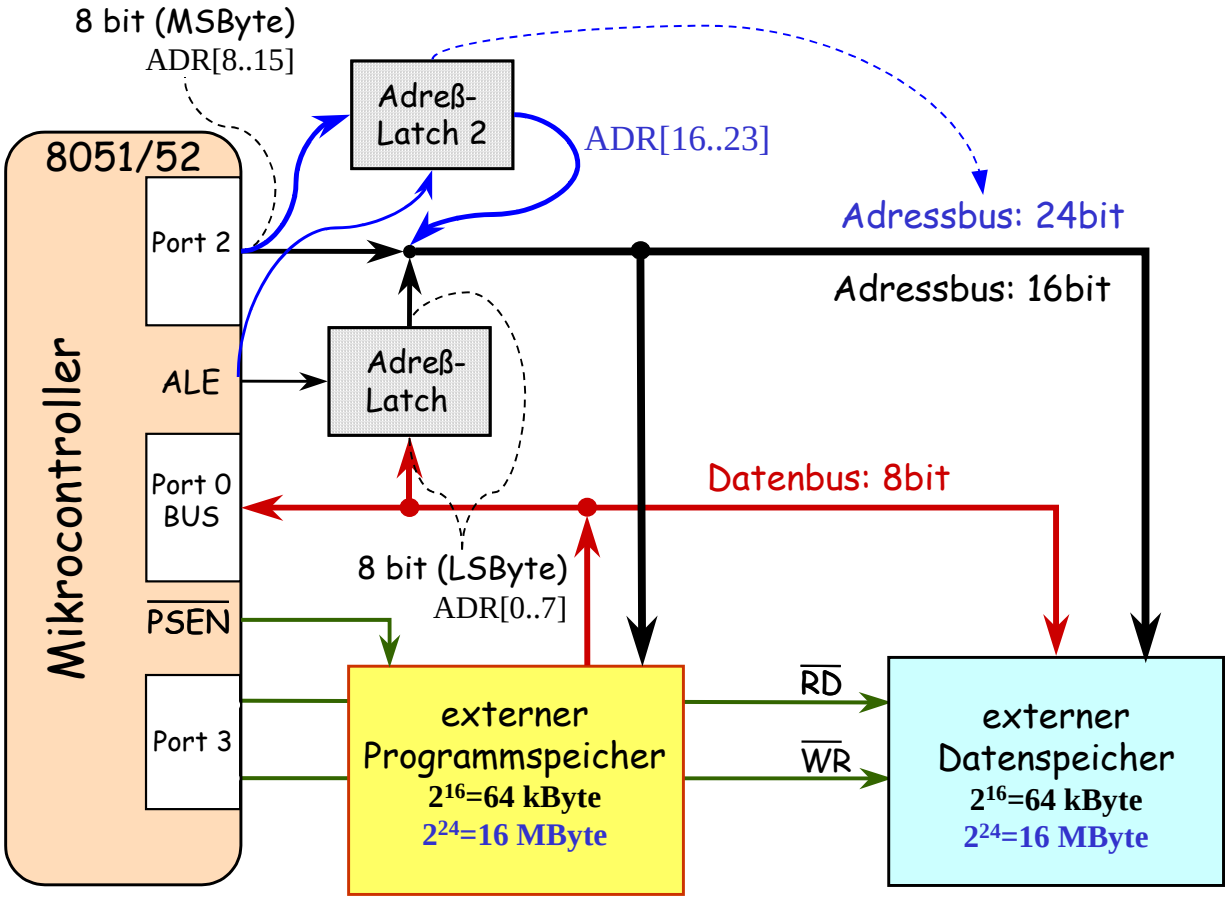



Bild 4.49: Externe Speicherweiterungen bei 8051-Mikrocontrollern

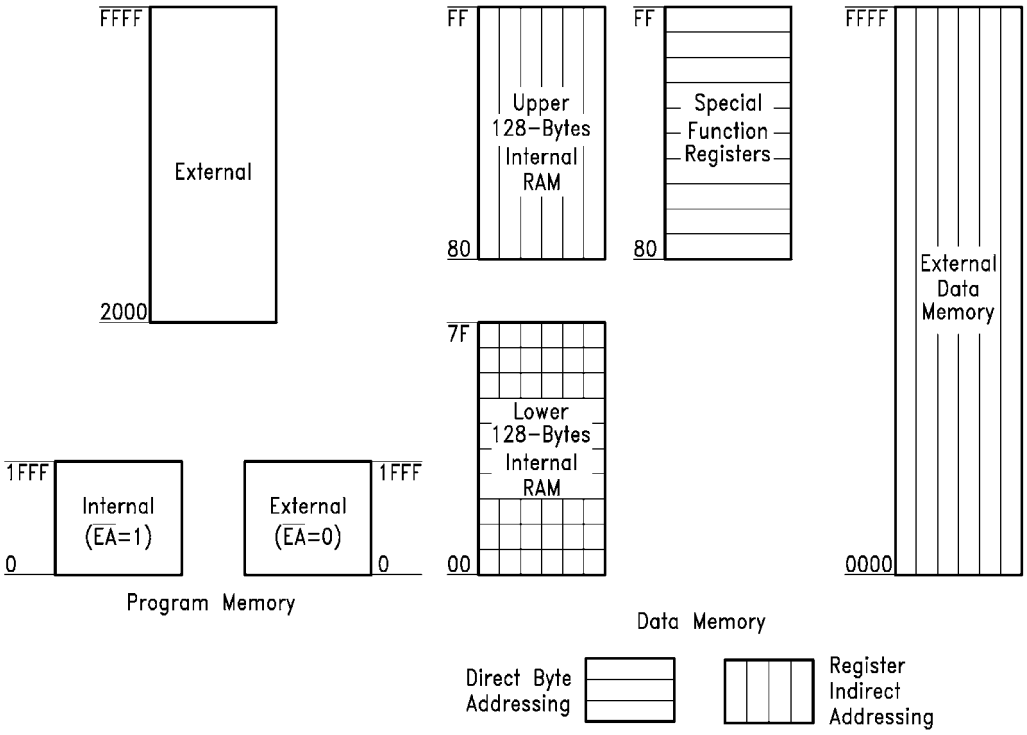



Bild 4.50: Das „Speichermodell“ des 8051

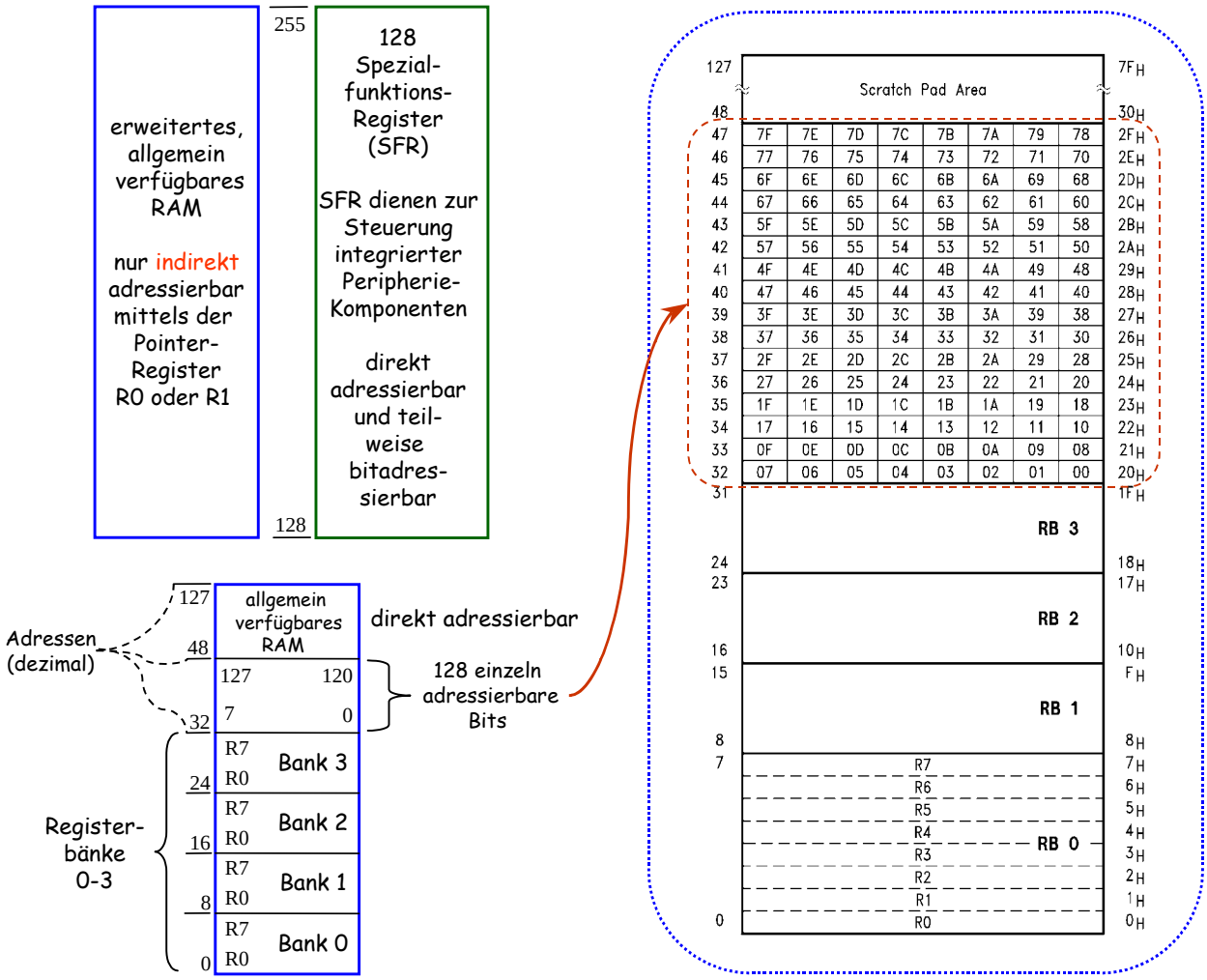



Bild 4.51: Details der Aufteilung der integrierten Datenspeicherbereiche

### 4.5.1.1 Die Spezialfunktionsregister der 8051-Einchip-Mikrocomputerfamilie

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| F8 | | | | | | | | FF |
| F0 | B | | | | | | | F7 |
| E8 | | | | | | | | EF |
| E0 | ACC | | | | | | | E7 |
| D8 | | | | | | | | DF |
| D0 | PSW | | | | | | | D7 |
| C8 | (T2CON) | (T2MOD) | (RCAP2L) | (RCAP2H) | (TL2) | (TH2) | | CF |
| C0 | | | | | | | | C7 |
| B8 | IP | | | | | | | BF |
| B0 | P3 | | | | | | | B7 |
| A8 | IE | | | | | | | AF |
| A0 | P2 | | | | | | | A7 |
| 98 | SCON | SBUF | | | | | | 9F |
| 90 | P1 | | | | | | | 97 |
| 88 | TCON | TMOD | TL0 | TL1 | TH0 | TH1 | | 8F |
| 80 | P0 | SP | DPL | DPH | | | | PCON 87 |

Bild 4.52: Spezialfunktionsregistersatz des ATMEL 89LS8252 (8052-Kern)

| Adresse dezimal | MSB | Bitadressen hexadezimal | | | | | LSB | Register-symbole und „Bitnamen“ |
|---|---|---|---|---|---|---|---|---|
| 240 | F7 | F6 | F5 | F4 | F3 | F2 | F1 | F0 | B |
| | ACC.7 | ACC.6 | ACC.5 | ACC.4 | ACC.3 | ACC.2 | ACC.1 | ACC.0 | ⇐ Bitnamen |
| 224 | E7 | E6 | E5 | E4 | E3 | E2 | E1 | E0 | ACC |
| | CY | AC | F0 | RS1 | RS0 | OV | | P | ⇐ Bitnamen |
| 208 | D7 | D6 | D5 | D4 | D3 | D2 | — | D0 | PSW |
| | | | | PS | PT1 | PX1 | PT0 | PX0 | ⇐ Bitnamen |
| 184 | — | — | — | BC | BB | BA | B9 | B8 | IP |
| 176 | B7 | B6 | B5 | B4 | B3 | B2 | B1 | B0 | P3 |
| | EA | | | ES | ET1 | EX1 | ET0 | EX0 | ⇐ Bitnamen |
| 168 | AF | — | — | AC | AB | AA | A9 | A8 | IE |
| 160 | A7 | A6 | A5 | A4 | A3 | A2 | A1 | A0 | P2 |
| | SM0 | SM1 | SM2 | REN | TB8 | RB8 | TI | RI | ⇐ Bitnamen |
| 152 | 9F | 9E | 9D | 9C | 9B | 9A | 99 | 98 | SCON |
| 144 | 97 | 96 | 95 | 94 | 93 | 92 | 91 | 90 | P1 |
| | TF1 | TR1 | TF0 | TR0 | IE1 | IT1 | IE0 | IT0 | ⇐ Bitnamen |
| 136 | 8F | 8E | 8D | 8C | 8B | 8A | 89 | 88 | TCON |
| 128 | 87 | 86 | 85 | 84 | 83 | 82 | 81 | 80 | P0 |

Tabelle 4.7: Bitadressierbare Speicherstellen im SFR-Bereich beim Standard 8051-Mikrocontroller

| | | | |
|---|---|---|---|
| ACC | Akkumulator | 00H | nach Reset |
| B | Register “B“ | 00H | nach Reset |
| PSW | Programmstatuswort | 00H | nach Reset |
| SP | Stackpointer | 07H | nach Reset |
| DPTR | 16bit-Datapointer (DPH, DPL) | 0000H | nach Reset |
| P0 | Port 0 = BUS | FFH | nach Reset |
| P1 | Port 1 | FFH | nach Reset |
| P2 | Port 2 (MS-Byte des PC) | FFH | nach Reset |
| P3 | Port 3 | FFH | nach Reset |
| TL0 | Timer 0 Low-Byte | 00H | nach Reset |
| TH0 | Timer 0 High-Byte | 00H | nach Reset |
| TL1 | Timer 1 Low-Byte | 00H | nach Reset |
| TH1 | Timer 1 High-Byte | 00H | nach Reset |
| SBUF | Sende/Empfangspuffer | unbestimmt nach Reset | |

Tabelle 4.8: Einige vom 8051-Kern belegte Spezialfunktionsregister

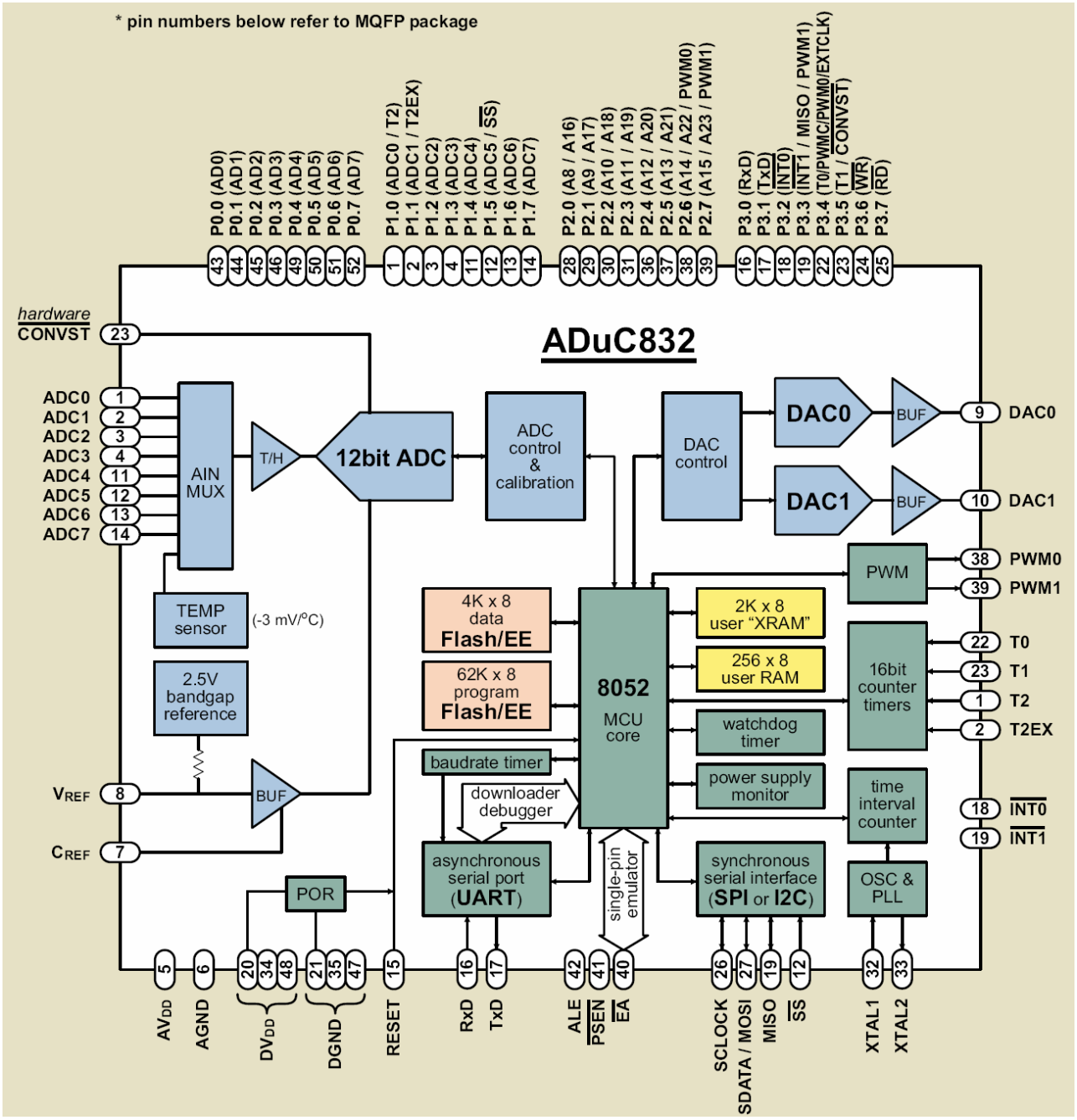



Bild 4.53: Blockschaltung des „Mikroconverters“ ADuC 832 (identisch mit ADuC 842)

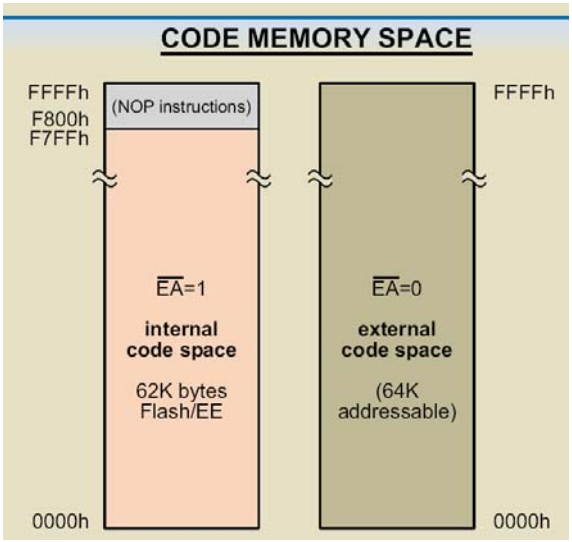



Bild 4.54: Interne und externe Speicherbereiche des ADuC 832/842 für Befehlscode

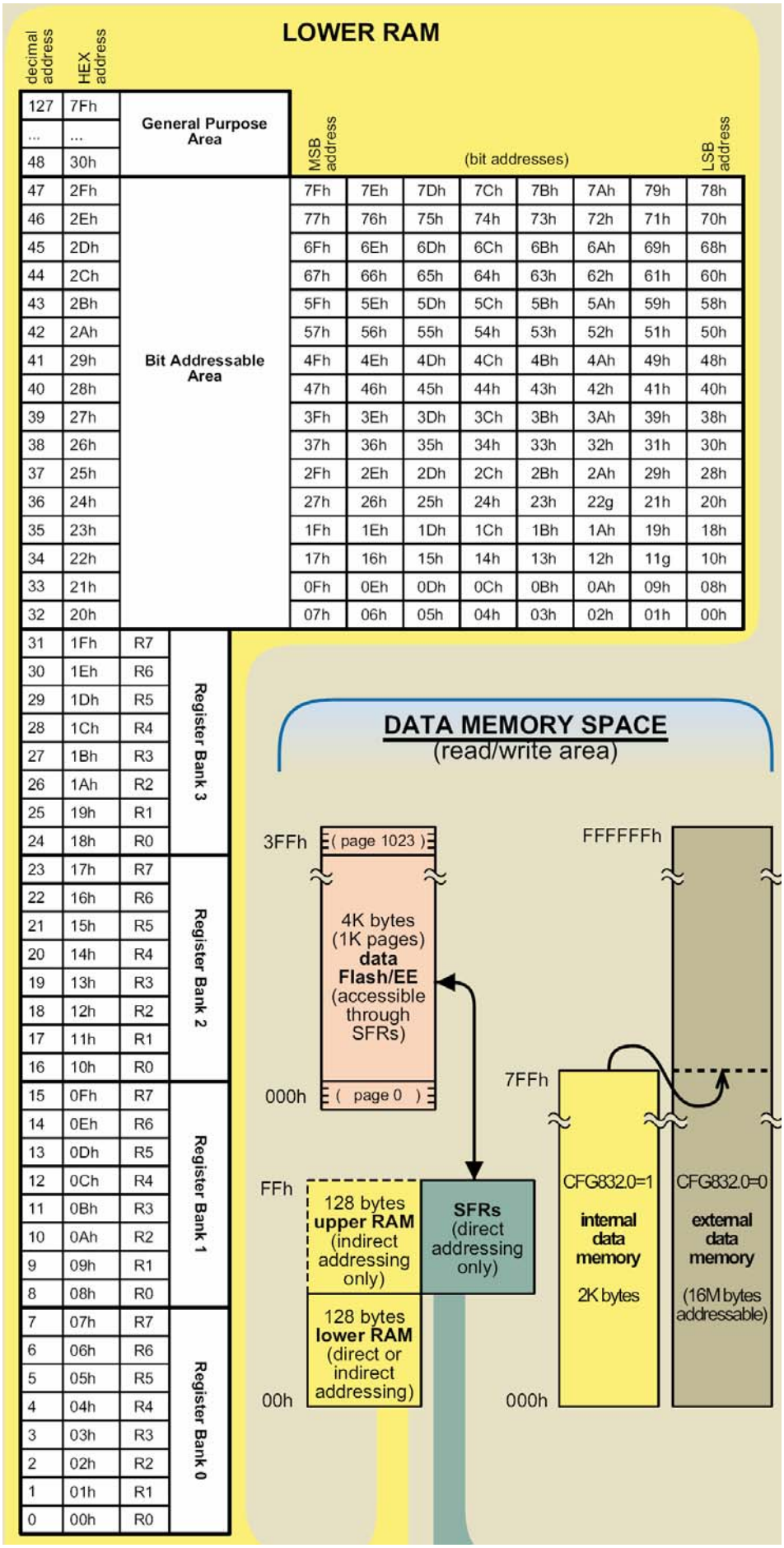



Bild 4.55: Datenspeicherorganisation im ADuC 832/842

### 4.5.2 Mikrocontroller-Befehlssatz-Übersicht am Beispiel „8051"

Befehls-Aufbau:

| **Befehl** | **Operand(en)** | |
|---|---|---|
| Mnemonische Abkürzung | Ziel | Quelle |
| MOV | @R0, | A |
| ANL | A, | R7 |

⇑

Das Ziel kann auch gleichzeitig Quelle des ersten Operanden sein

Es gibt insgesamt 111 Befehle:

- 49 1-Byte-Befehle
- 45 2-Byte-Befehle
- 17 3-Byte-Befehle
  - 64 Befehle benötigen 1 Maschinen-Zyklus
  - 45 Befehle benötigen 2 Maschinen-Zyklen
  - 2 Befehle (MUL, DIV) benötigen 4 Maschinen-Zyklen

# Gruppierung nach Funktionen

## Datentransfer:

| | |
|---|---|
| MOV | Bit, Byte |
| MOVX | externer Daten-Byte-Transfer (Adresse 16- oder 8bit)<br>Pointer: DPTR oder R0, R1 |
| MOVC | bringt 1Byte aus dem Programmspeicher in den Akku<br>Adresse: PC+@A oder DPTR+@A (indexed Offset) |
| PUSH | erhöht Stackpointer (SP) und bringt den Quellenoperanden an den vom Stackpointer adressierten Speicherplatz |
| POP | bringt ein Byte, das vom aktuellen Stackpointer adressiert wird, zum Ziel (z.B. ACC) und dekrementiert dann den SP |
| XCH | Daten-Bytes tauschen |
| XCHD | Halbbytes (d.h. nur die Bits 0...3) tauschen |

## 16 bit-Transfers für Adressen unter Verwendung des „Datenpointers“ (DPTR):

MOV DPTR, #data

⇑ 16bit-Wort (Adresse)

## Beispiele für Befehle **ohne** Operanden:

| | |
|---|---|
| NOP | ;es wird nichts ausgeführt, aber 1 Maschinenzyklus läuft ab |
| RET | ;Rücksprung aus einem Unterprogramm |
| RETI | ; Rücksprung aus einer Interrupt-Bedienungsroutine |

## Beispiele für Befehle mit **einem** Operanden:

| | | |
|---|---|---|
| SETB | C | Carry oder irgendein direkt adressierbares Bit setzen |
| CLR | C | Carry oder irgendein direkt adressierbares Bit löschen |
| CPL | A | Komplement des Akkuinhaltes bilden |
| CPL | C | Carry oder ein direkt adressierbares Bit komplementieren |
| RL, RR | A | ;Akkuinhalt um eine Bitposition links bzw. rechts schieben (ein Bit geht jeweils verloren) |
| RLC, RRC | A | ;Akkuinhalt um eine Bitposition links bzw. rechts über das Carrybit hinweg schieben (das Bit das jeweils den Akku verläßt kommt ins Carry) |
| SWAP | A | ;Halbbytes im Akku tauschen |

## Beispiele für Zweioperanden-Befehle für logische Operationen

```
ANL     A, #3FH
ORL     C, P1.3       ;analog für Byte-Operanden
XRL     A, R4         ;geht nicht für Bitoperanden
```

## Beispiele für arithmetische Zweioperanden-Befehle

**Addition**

```
ADD     A, R4
ADDC    A, #data      ;unter Einbeziehung von Carry
INC     A
DA      A             ;Dezimalkorrektur für das Rechnen mit BCD-Zahlen
```

**Subtraktion**

```
SUBB    A, R7         ;Subtraktion mit "Borgen", d.h.
                      (A)–(C)–(R7) → (A)
DEC     A
```

**Multiplikation**

```
MUL     AB            ;ohne Vorzeichen A·B
                      ;2 Byte-Ergebnis: LS-Byte =>A; MS-Byte =>B
                      ;wenn B > 0 ⇒ OV= „1“, sonst OV= „0“
                      ;C immer „0“, AC bleibt unbeeinflußt
```

**Division**

```
DIV     AB            ;ohne Vorzeichen A:B das ganzzahlige Ergebnis steht in
                      A, der Rest in B
                      ;Division durch 0 setzt OV-Flag (Ergebnis undefiniert)
                      sonst OV= „0“, C= „0“, AC unbeeinflußt
```

Flag-Bits

| | | |
|---|---|---|
| Carry | C | ;allgemeines Überlauf-Bit |
| Auxiliary-Carry | AC | ;nützlich beim Rechnen mit BCD-Zahlen |
| Overflow | OV | ;nützlich beim Rechnen mit Zweierkomplementzahlen |
| Parity | P | ;ergänzt die Modulo-2-Summe der Akku-Bits zu Null bei gerader und Eins bei ungerader Parität |

## Sprung- und Verzweigungsbefehle

unbedingt

```
JMP     @A+DPTR       ;Akku-Inhalt + DPTR => PC (Zieladresse)
SJMP                  ;“short jump” innerhalb 256Bytes: PC ± 128
AJMP                  ;11-bit-Adresse (2-Byte-Befehl) – Ziel innerhalb von 2K
LJMP                  ;“long jump” 16bit-Adresse (3-Byte-Befehl)
```

| | | |
|---|---|---|
| ACALL | | ;Unterprogrammaufruf – Ziel innerhalb von 2K |
| LCALL | | ;Unterprogrammaufruf – Ziel innerhalb eines 64K-Adreßraumes |
| RET | | ;Rückkehr aus einem Unterprogramm |
| RETI | | ;Rückkehr von einer Interruptbedienung. Hier darf **nicht RET** verwendet werden, weil die Interrupt-Hardware dann nicht zurückgesetzt wird und der gerade bediente Interrupt nie beendet würde! |

bedingte Sprünge und Verzweigungen

| | | |
|---|---|---|
| JZ, JNZ | A, Ziel | ;springe zum Ziel ,wenn A=0 bzw. ≠0 ist |
| JC, JNC | Ziel | ;springe zum Ziel ,wenn C=„1“ bzw. „0“ ist |
| JB, JNB | BIT, Ziel | ;springe zum Ziel ,wenn BIT=„1“ bzw. „0“ ist |
| JBC | BIT, Ziel | ;springe zum Ziel ,wenn BIT=„1“ ist und setzte dann BIT=“0“ |

**Vergleichsbefehle, verbunden mit Sprüngen**

| | | |
|---|---|---|
| CJNE | A, R7; Ziel | ;Compare and Jump if Not Equal<br>vergleiche A und R7 und springe zum Ziel wenn A≠R7 C= „1“ wenn A > R7, sonst ist C= „0“ |
| DJNZ | R3, Ziel | ;Decrement and Jump if Not Zero<br>vermindere R3 um Eins und springe zum Ziel, wenn R3≠0 ist |

## Arithmetic Operations

| | | bytes | OSC periods |
|---|---|---|---|
| ADD A,source | add source to A | 1,2 | 12 |
| ADD A,#data | | 2 | 12 |
| ADDC A,source | add with carry | 1,2 | 12 |
| ADDC A,#data | | 2 | 12 |
| SUBB A,source | subtract from A with borrow | 1,2 | 12 |
| SUBB A,#data | | 2 | 12 |
| INC A | increment | 1 | 12 |
| INC source | | 1,2 | 12 |
| INC DPTR * | | 1 | 24 |
| DEC A | decrement | 1 | 12 |
| DEC source | | 1,2 | 12 |
| MUL AB | multiply A by B | 1 | 48 |
| DIV AB | divide A by B | 1 | 48 |
| DA A | decimal adjust | 1 | 12 |

## Data Transfer Operations

| | | bytes | OSC periods |
|---|---|---|---|
| MOV A,source | move source to destination | 1,2 | 12 |
| MOV A,#data | | 2 | 12 |
| MOV dest,A | | 1,2 | 12 |
| MOV dest,source | | 1,2,3 | 24 |
| MOV dest,#data | | 2,3 | 12,24 |
| MOV DPTR,#data16 | | 3 | 24 |
| MOVC A,@A+DPTR | move from code memory | 1 | 24 |
| MOVC A,@A+PC | | 1 | 24 |
| MOVX A,@Ri | move to/from data memory | 1 | 24 |
| MOVX A,@DPTR | | 1 | 24 |
| MOVX @Ri,A | | 1 | 24 |
| MOVX @DPTR,A | | 1 | 24 |
| PUSH direct | push onto stack | 2 | 24 |
| POP direct | pop from stack | 2 | 24 |
| XCH A,source | exchange bytes | 1,2 | 12 |
| XCHD A,@Ri | exchg low digits | 1 | 12 |

## Program Branching

| | | bytes | OSC periods |
|---|---|---|---|
| ACALL addr11 | call subroutine | 2 | 24 |
| LCALL addr16 | | 3 | 24 |
| RET | return from sub. | 1 | 24 |
| RETI | return from int. | 1 | 24 |
| AJMP addr11 | jump | 2 | 24 |
| LJMP addr16 | | 3 | 24 |
| SJMP rel | | 2 | 24 |
| JMP @A+DPTR | | 1 | 24 |
| JZ rel | jump if A = 0 | 2 | 24 |
| JNZ rel | jump if A not 0 | 2 | 24 |
| CJNE A,direct,rel | compare and jump if not equal | 3 | 24 |
| CJNE A,#data,rel | | 3 | 24 |
| CJNE Rn,#data,rel | | 3 | 24 |
| CJNE @Ri,#data,rel | | 2 | 24 |
| DJNZ Rn,rel | decrement and jump if not zero | 2 | 24 |
| DJNZ direct, rel | | 3 | 24 |
| NOP | no operation | 1 | 12 |

## Legend

| | |
|---|---|
| Rn | register addressing using R0-R7 |
| direct | 8bit internal address (00h-FFh) |
| @Ri | indirect addressing using R0 or R1 |
| source | any of [Rn, direct, @Ri] |
| dest | any of [Rn, direct, @Ri] |
| #data | 8bit constant included in instruction |
| #data16 | 16bit constant included in instruction |
| bit | 8bit direct address of bit |
| rel | signed 8bit offset |
| addr11 | 11bit address in current 2K page |
| addr16 | 16bit address |

* INC DPTR increments the 24bit value DPP/DPH/DPL

## Logical Operations

| | | bytes | OSC periods |
|---|---|---|---|
| ANL A,source | logical AND | 1,2 | 12 |
| ANL A,#data | | 2 | 12 |
| ANL direct,A | | 2 | 12 |
| ANL direct,#data | | 3 | 24 |
| ORL A,source | logical OR | 1,2 | 12 |
| ORL A,#data | | 2 | 12 |
| ORL direct,A | | 2 | 12 |
| ORL direct,#data | | 3 | 24 |
| XRL A,source | logical XOR | 1,2 | 12 |
| XRL A,#data | | 2 | 12 |
| XRL direct,A | | 2 | 12 |
| XRL direct,#data | | 3 | 24 |
| CLR A | clear A to zero | 1 | 12 |
| CPL A | complement A | 1 | 12 |
| RL A | rotate A left | 1 | 12 |
| RLC A | ...through C | 1 | 12 |
| RR A | rotate A right | 1 | 12 |
| RRC A | ...through C | 1 | 12 |
| SWAP A | swap nibbles | 1 | 12 |

## Boolean Variable Manipulation

| | | bytes | OSC periods |
|---|---|---|---|
| CLR C | clear bit to zero | 1 | 12 |
| CLR bit | | 2 | 12 |
| SETB C | set bit to one | 1 | 12 |
| SETB bit | | 2 | 12 |
| CPL C | complement bit | 1 | 12 |
| CPL bit | | 2 | 12 |
| ANL C,bit | AND bit with C | 2 | 24 |
| ANL C,/bit | ...NOTbit with C | 2 | 24 |
| ORL C,bit | OR bit with C | 2 | 24 |
| ORL C,/bit | ...NOTbit with C | 2 | 24 |
| MOV C,bit | move bit to bit | 2 | 12 |
| MOV bit,C | | 2 | 24 |
| JC rel | jump if C set | 2 | 24 |
| JNC rel | jmp if C not set | 2 | 24 |
| JB bit,rel | jump if bit set | 3 | 24 |
| JNB bit,rel | jmp if bit not set | 3 | 24 |
| JBC bit, rel | jmp&clear if set | 3 | 24 |

Bild 4.56: Zusammenfassende Übersicht des 8051-Befehlssatzes

Für Mikroconverter vom Typ ADuC832 (pinkompatibel mit ADuC842) steht PC-basierte Entwicklungssoftware auf der Webseite „Mikrorechnertechnik“ unter MC_TOOLs zur Verfügung.

#### 4.5.2.1 Ein erstes einfaches Programmbeispiel

```
$Title (Projekt: Datenausgabe an Port 2    April 2006)
$Debug                                  ;Symbole im Objektcode
$Xref                                   ;Querverweisliste am Ende
$Registerbank (0)                       ;benutzte Registerbank
NAME           P2_DATENAUSGABE
;*****************************************************************
;Dieses Programm schaltet Pins am Port 2 nach verschiedenen
;Mustern um
;Zielhardware: 80C51 kompatibler Mikrocontroller, auch ADuC8xx
;Es wird eine Taktfrequenz von 12MHz zugrunde gelegt, bei der
;ein Maschinenzyklus exakt 1µs dauert
;*****************************************************************
;Beim Start sind alle Pins von Port 2 auf '1'
;d.h. es steht 'FFh' im P2-Latch

Start:
MOV     P2,#00  ;alle Pins auf Null setzen
SETB    P2.0    ;Bit 0 auf '1' setzen
SETB    P2.1    ;Bit 1 auf '1' setzen
SETB    P2.2    ;Bit 2 auf '1' setzen
SETB    P2.3    ;Bit 3 auf '1' setzen
SETB    P2.4    ;Bit 4 auf '1' setzen
SETB    P2.5    ;Bit 5 auf '1' setzen
SETB    P2.6    ;Bit 6 auf '1' setzen
SETB    P2.7    ;Bit 7 auf '1' setzen

;Ziffern 0...9 an eine 7-Segementanzeige ausgeben
;           xabcdefg
MOV     P2,#01111110b   ;0 = abcdef  an Siebensegmentanzeige
MOV     P2,#00110000b   ;1 = bc      an Siebensegmentanzeige
MOV     P2,#01101101b   ;2 = abdeg   an Siebensegmentanzeige
MOV     P2,#01111001b   ;3 = abcdg   an Siebensegmentanzeige
MOV     P2,#00110011b   ;4 = bcfg    an Siebensegmentanzeige
MOV     P2,#01011011b   ;5 = acdfg   an Siebensegmentanzeige
MOV     P2,#01011111b   ;6 = acdefg  an Siebensegmentanzeige
MOV     P2,#01110000b   ;7 = abc     an Siebensegmentanzeige
MOV     P2,#01111111b   ;8 = abcdefg an Siebensegmentanzeige
MOV     P2,#01111011b   ;9 = abcdfg  an Siebensegmentanzeige
```

```
MOV     P2,#01110111b   ;A = abcefg  an Siebensegmentanzeige
MOV     P2,#00011111b   ;B = cdefg   an Siebensegmentanzeige
MOV     P2,#01001110b   ;C = adef    an Siebensegmentanzeige
MOV     P2,#00111101b   ;D = bcdeg   an Siebensegmentanzeige
MOV     P2,#01001111b   ;E = adefg   an Siebensegmentanzeige
MOV     P2,#01000111b   ;F = aefg    an Siebensegmentanzeige
JMP     Start             ;wieder von vorne anfangen
END
```

Batch-Datei a.bat:

```
ass51 %1.a51          ;assemblieren
link %1.obj to %1     ;zusammenbinden „linken“
o_h %1                ;in HEX-Format wandeln
hexbin %1.hex         ;Binärformat erzeugen
del %1                ;nicht mehr benötigte
del %1.m51            ;Dateien löschen
del %1.obj
```

Aufruf: a P2_DAT

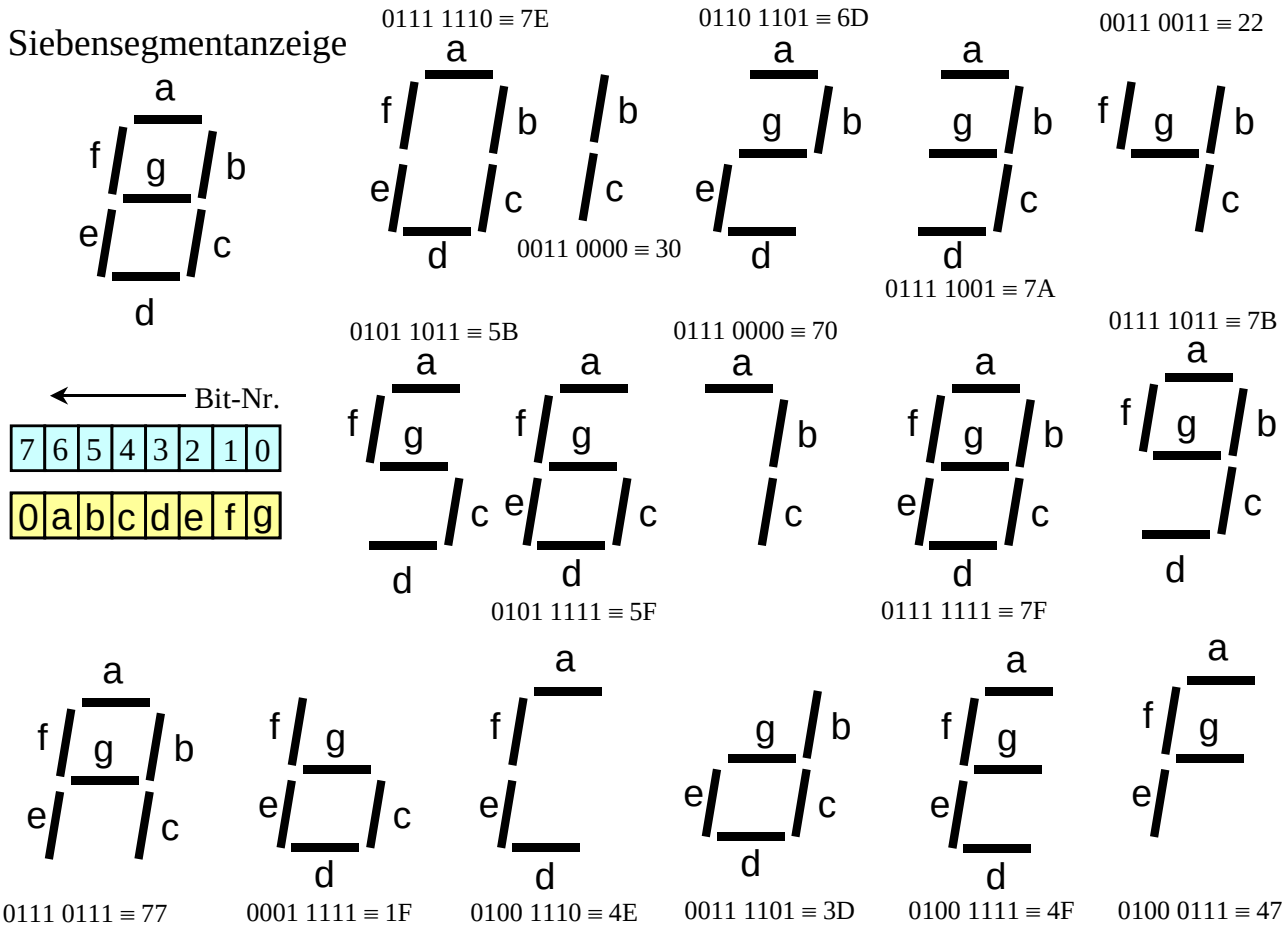



Bild 4.57: Die Siebensegmentanzeige mit der zugehörigen Zifferncodierung

Datei im Intel–HEX–Format

```
:10000000 75A000D2A0D2A1D2A2D2A3D2A4D2A5D24E
:10001000 A6D2A775A07E75A03075A06D75A0797564
:10002000 A03375A05B75A05F75A07075A07F75A0EB
:10003000 7B75A07775A01F75A04E75A03D75A04F6C
:05004000 75A04780BB24
:0000000  1FF
```

Datei im Binärformat

- enthält nur noch die „reinen“ OP-Codes der Mikrocontrollerbefehle -

```
75 A0 00 D2 A0 D2 A1 D2 A2 D2 A3 D2 A4 D2 A5 D2
A6 D2 A7 75 A0 7E 75 A0 30 75 A0 6D 75 A0 79 75
A0 33 75 A0 5B 75 A0 5F 75 A0 70 75 A0 7F 75 A0
7B 75 A0 77 75 A0 1F 75 A0 4E 75 A0 3D 75 A0 4F
75 A0 47 80 BB
```

## 4.6 Aufbereitung von Information für die digitale Verarbeitung

### 4.6.1 Entropie H einer Nachrichtenquelle

Die Entropie ist der mittlere Informationsgehalt einer Nachrichtenquelle. Sie wird empirisch definiert, so daß eine binäre Quelle mit einem Vorrat von zwei Zeichen genau $H_B$=1bit hat, wenn die beiden Zeichen mit gleicher Wahrscheinlichkeit, d.h. 0,5 auftreten.

Allgemein wird zunächst eine Quelle mit einem Zeichenvorrat $x_i$ betrachtet, wobei i = 1...*N* gilt, d.h. die Quelle liefert *N* verschiedene Zeichen

Die Auftrittswahrscheinlichkeit eines Zeichens $x_i$ sei $p(x_i)$

- Die Erfahrung lehrt, daß ein Zeichen einen hohen Informationswert hat, wenn es selten auftritt, d.h. wenn $p(x_i)$ klein, aber noch von Null verschieden ist.

Daraus leitet sich in anschaulicher Weise die Definition der Entropie

$$H = \sum_{i=1}^{N} p(x_i) \cdot \mathrm{ld}\left(\frac{1}{p(x_i)}\right) \quad \text{in bit} \tag{4.21}$$

ab, die dem mittleren Informationsgehalt der Nachrichtenquelle entspricht. Aus (4.21) ergibt sich in einfacher Weise für den wichtigen Sonderfall einer binären Quelle mit den beiden Zeichen $x_1$=0 und $x_2$=1, unter der Voraussetzung, daß $p(x_i)$=1/2 ist (beide Zeichen gleichwahrscheinlich):

$$H_B = \sum_{i=1}^{2} p(x_i) \cdot \mathrm{ld}\left(\frac{1}{p(x_i)}\right) = \frac{1}{2}\mathrm{ld}(2) + \frac{1}{2}\mathrm{ld}(2) = 1\,\mathrm{bit} \tag{4.22}$$

(4.22) stellt somit die Definition der kleinsten Informationseinheit dar.

Weitere Beispiele:

Die 10 Ziffern: 0...9: Wenn hier $p(x_i)$=1/10 ist, dann hat man

$$H_Z = \sum_{i=1}^{10} \frac{1}{10} \cdot \mathrm{ld}(10) = \mathrm{ld}(10) = 3{,}32\,\mathrm{bit} \tag{4.23}$$

Das Alphabet: A...Z (ohne Umlaute, d.h. N=26): Unter der Annahme, daß $p(x_i)$=1/26 erhält man

$$H_A = \mathrm{ld}(26) = 4{,}7\ \mathrm{bit} \tag{4.24}$$

Die Auftrittshäufigkeit der Buchstaben des Alphabets in einem beliebigen Text ist nicht gleich; sie ist auch erheblich von der Sprache abhängig. So gilt z. B. für einen deutschen Text:

$$H_{DT} = 4{,}11\ \mathrm{bit} < H_A, \text{ denn } p(x_i) \neq 1/26$$

Die Entropie einer Quelle wird immer dann maximal, wenn die Auftrittswahrscheinlichkeit ihrer Zeichen gleich ist. Das sei im folgenden am Beispiel der binären Quelle gezeigt:

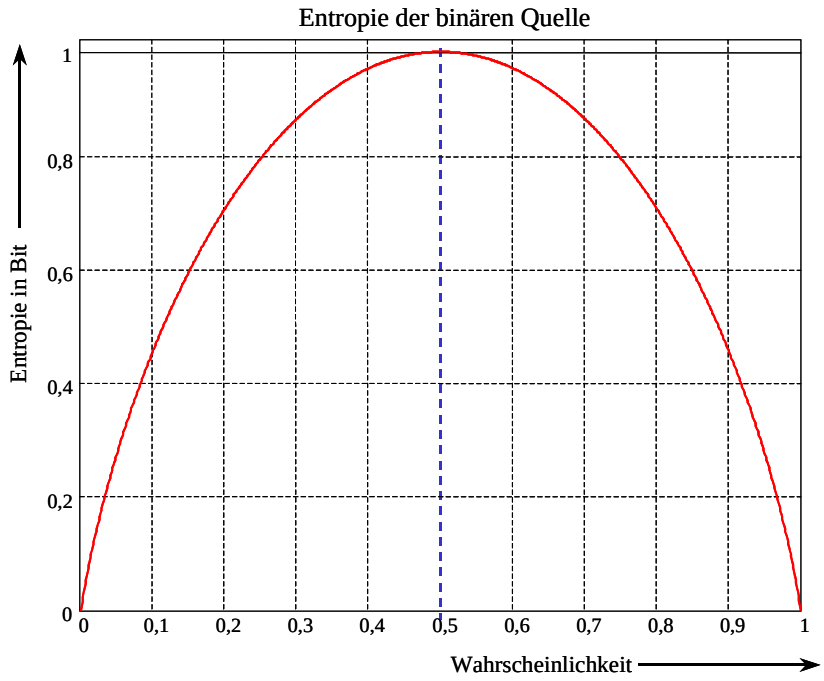



Bild 4.58: Entropie als Funktion der Wahrscheinlichkeit

Man erkennt leicht, daß das Maximum für $p(x_i)=1/2$ auftritt. Das läßt sich auch in einfacher Weise analytisch aus (4.22) ermitteln, wenn man dort z.B. $p(x_i)=\xi$ setzt, die Gleichung

$$\frac{-\xi}{\ln 2}\cdot\ln(\xi)+\frac{-(1-\xi)}{\ln 2}\cdot\ln(1-\xi)$$

nach $\xi$ differenziert und das Ergebnis Null setzt. Man erhält $\xi=1/2$.

## 4.6.2 Informationsgehalt analoger, zeit- und wertkontinuierlicher Signale

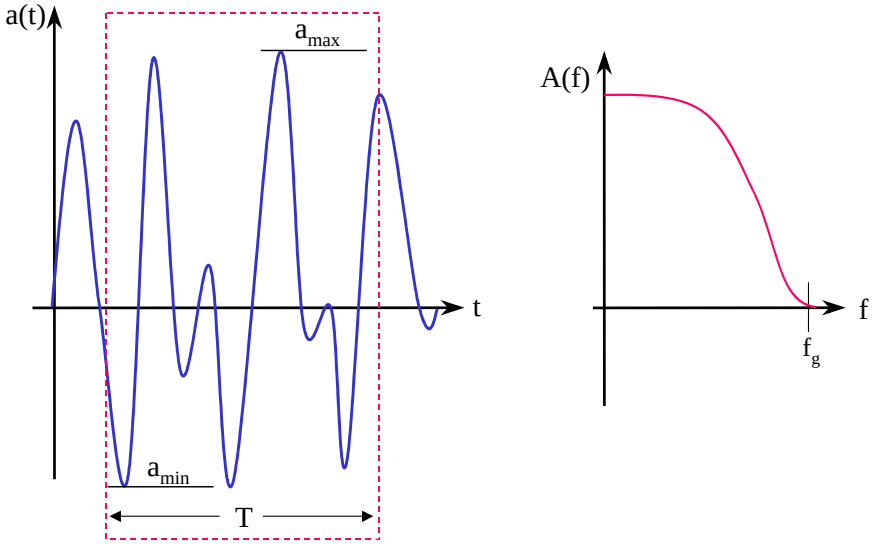



Bild 4.59: Zeit- und Frequenzdarstellung eines Analogsignals

$a_{max}$ und $a_{min}$ haben physikalische Grenzen: $a_{max}$ bei der Versorgungsspannung und $a_{min}$ beim Hintergrundrauschen.

Das bedeutet, daß in einem technischen System nur eine endliche Anzahl Z von Amplitudenstufen in einem Bereich $|a_{max}\text{-}a_{min}|$ unterscheidbar ist.

Unter der Annahme, daß jede der Amplitudenstufen mit gleicher Wahrscheinlichkeit auftritt, kann in einfacher Weise die Entropie der Signalquelle berechnet werden:

$$H_{AS}=\mathrm{ld}(Z) \qquad (4.25)$$

Um den Informationsgehalt eines Signalausschnitts der Dauer $T$ zu bestimmen, müssen wir zunächst wissen, wie viele „Proben“ bzw. Abtastwerte dem Signal innerhalb der Dauer $T$ entnommen werden müssen. Die Antwort liefert das sogenannte Shannon’sche Abtasttheorem – *s. weiterführendes Ergänzungsskript*, das besagt, daß die Abtastwerte mindestens mit dem Doppelten der oberen Grenzfrequenz $f_g$ zu nehmen sind. Dann ist gewährleistet, daß eine fehlerfreie Rekonstruktion des Analogsignals aus den Abtastwerten erfolgen kann. Innerhalb der Dauer $T$ fallen somit $2\cdot f_g\cdot T$ Abtastwerte an, so daß man

$$H_{\mathrm{AST}} = 2 \cdot f_{\mathrm{g}} \cdot T \cdot \mathrm{ld}(Z) \qquad [\mathrm{bit}] \tag{4.26}$$

für Informationsgehalt des Signalausschnitts erhält.

Das folgende Beispiel eines Ausschnitts aus einem in „HiFi-Qualität“ verfügbaren Musikstück verdeutlicht die Anwendung von (4.26):

Dauer: $T = 1\mathrm{s}$

obere Grenzfrequenz: $f_{\mathrm{g}} = 15\ \mathrm{kHz}$

Zahl der Amplitudenstufen: $Z = 65536 \equiv 2^{16}$

$$H_{\mathrm{AST}} = 1\mathrm{s} \cdot 2 \cdot 15.000\,\mathrm{s}^{-1} \cdot \mathrm{ld}(2^{16})\ \mathrm{bit} = 30.000 \cdot 16\,\mathrm{bit} = 480.000\,\mathrm{bit} \tag{4.27}$$

Die erforderliche Datenrate zum digitalen Übertragen wäre also 480 kbit/s.

### 4.6.3 Übersicht von Grundverfahren zur A/D- und D/A-Wandlung

Die Qualität digitaler Signalverarbeitungs- und Kommunikationssysteme wird wesentlich durch die Genauigkeit des eingangsseitigen Digitalsignals bestimmt. Die schwerwiegendsten Fehler passieren in der Regel bei der Analog/Digitalwandlung. Sie sind meist irreversibel, d.h. auch durch noch so aufwendige Signalverarbeitung nicht mehr korrigierbar.

Für die A/D-Wandlung sind zahlreiche Verfahren mit einer Reihe von Varianten bekannt. An dieser Stelle werden nur die wichtigsten einführend und grundlegend behandelt. Eine detaillierte Analyse muß späteren Vorlesungen vorbehalten bleiben, weil tiefergehende Kenntnisse der Systemtheorie und der Schaltungstechnik dazu benötigt werden.

**Die wichtigsten A/D-Wandler-Prinzipien:**

- Integrierende Verfahren (Zählmethode über alle Amplitudenstufen)
- Sukzessive Approximation (schrittweise Annäherung in Zweierpotenzstufen)
  - wird im folgenden exemplarisch behandelt
- Flash-Wandlung (Direktwandlung in einem Schritt, mit sehr vielen Komparatoren)
- Pipelining-Prinzip (vereint Geschwindigkeit und hohe Auflösung)
  - wird im folgenden exemplarisch behandelt
- Sigma-Delta-Prinzip (sehr hohe Auflösung bei geringem Aufwand auf Analogseite)

Wie wir sehen werden, beinhalten A/D-Wandler häufig einen D/A-Wandler in einem Rückführungszweig, so daß die Genauigkeit der A/D-Wandlung wesentlich durch die Qualität dieses Bausteins mitbestimmt wird. Beim Flash- und Pipelining-Wandler ist die D/A-Komponente nicht auf den ersten Blick erkennbar, weil sie „verteilt“ realisiert ist.

**D/A-Wandler sind in ihrer Funktion leichter zu verstehen als A/D-Wandler**

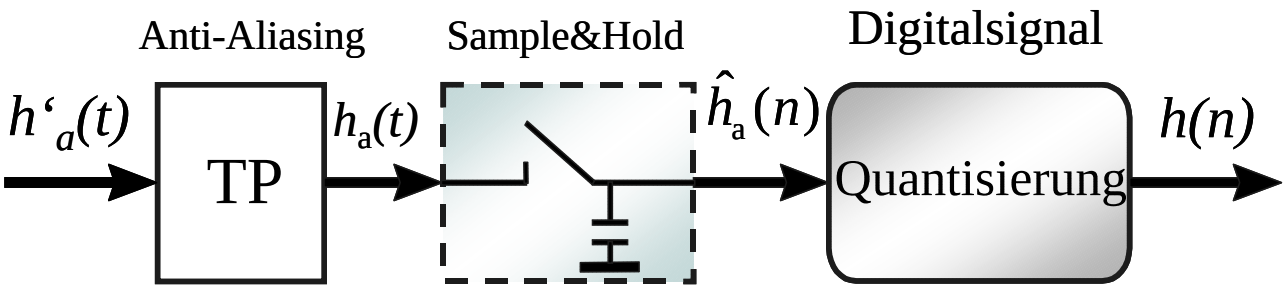
**Hier werden exemplarisch**

- R/2R-Wandler, deren Kernstück ein einfaches Widerstandsnetzwerk ist
- Pulsweitenmodulatoren, bei denen ein Analogwert durch das Tastverhältnis einer Rechteckfolge bestimmt wird

**betrachtet, da sie in enger Beziehung zur Mikrorechnertechnik stehen.**

#### 4.6.3.1 A/D-Wandelverfahren

Die typischen Schritte einer A/D-Wandlung sind im folgenden Bild zusammengefaßt – vgl. *weiterführendes Ergänzungsskript* für eine ausführlichere Beschreibung. Für das Verständnis der folgenden Betrachtungen genügt es, sich darüber im Klaren zu sein, daß ein Analogsignal vor der Wandlung stets einer Bandbegrenzung unterzogen werden muß, damit eine obere Grenzfrequenz $f_g$ definiert werden kann. Diese Aufgabe erfüllt der „Anti-Aliasing-Tiefpaß ". Das „Abtasttheorem" - s. *weiterführendes Ergänzungsskript* – gibt vor, daß das Sample&Hold-Glied mit mindestens der Rate $2 \cdot f_g$ die Abtastung vorzunehmen hat. Im letzten Schritt (Quantisierung) werden dann die zeitdiskreten aber noch wertkontinuierlichen Abtastwerte diskreten Amplitudenstufen zugeordnet. Dieser Vorgang ist grundsätzlich fehlerbehaftet. Die Berechnung des sogenannten Quantisierungsfehlers und des daraus ableitbaren „Störabstandes", der ein wichtiges Qualitätsmaß eines A/D-Wandlers darstellt, sind ebenfalls im *weiterführenden Ergänzungsskript* zu finden. An dieser Stelle genügt das Ergebnis in Form einer einfachen „Daumenregel":




$$\frac{a}{\mathrm{dB}} = 10\lg\left(\frac{S_s}{N_q}\right) \approx 2 + 6 \cdot Z \qquad (4.28)$$

Mit dieser Regel lassen sich A/D-Wandler verschiedener Auflösung (ausgedrückt durch ihre **Bitzahl *Z***) sehr einfach hinsichtlich ihres Störabstandes zum Quantisierungsrauschen miteinander vergleichen. So beträgt der Störabstand für einen 8bit-Wandler z.B. rund 50 dB, während man mit 16 bit 98 dB erreichen kann. Für die Praxis sind diese Werte wichtig, weil man auf ihrer Grundlage technische Anforderungen festlegen kann bzw. muß. Für die digitale Audiosignalverarbeitung sind z.B. mindestens Auflösungen von 16 bit nötig, da das menschliche Gehör einen Dynamikumfang in der Größenordnung von 95 dB hat.

##### 4.6.3.1.1 Sukzessive Approximation (schrittweise Annäherung in Zweierpotenzstufen)

Bei diesem Wandelverfahren wird mit verhältnismäßig geringem Aufwand ein schneller Ablauf des Wandelvorganges erzielt. In einem „sukzessiven Approximationsregister" werden die einzelnen Bits des digitalen Ausgangsignals gesetzt oder rückgesetzt. Der Ablauf dieser Setz- und Rücksetzvorgänge wird über einen endlichen Automaten (FSM ≡ Finite State Machine) gesteuert. Die

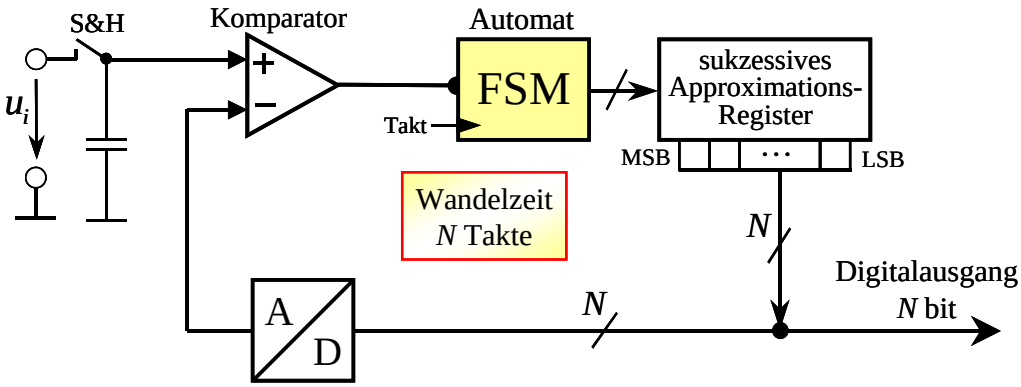



Bild 4.60: Prinzip des SAR-Wandelverfahrens

Funktionsweise und der Aufbau solcher Automaten wird später behandelt.

Eingangsseitig finden wir in Bild 4.60 nach dem Sample&Hold-Glied (S&H) einen Komparator. Solange $u_i$ größer ist als der über den D/A-Wandler rückgeführte Wert, läuft der im folgenden näher erläuterte Wandelvorgang weiter, d.h. mit jedem Taktimpuls für die FSM erhöht sich der im SAR gespeicherte Wert und damit wächst der Analogwert am – Eingang des Komparators. Dieser signalisiert somit der FSM, ob der rückgeführte D/A-gewandelte Wert des Digitalsignals das Eingangssignal $u_i$ übersteigt oder nicht.

Zu Beginn einer Wandlung sind alle Bits des SAR auf Null, so daß der Komparator der FSM mit einem „1"-Pegel signalisiert, daß $u_i$ größer ist als der zugehörige aktuelle Digitalwert. Die FSM setzt daraufhin das MSB[15] auf „1". Bleibt der Komparatorausgang danach auf „1", wird das nächste, niederwertigere Bit des SAR von der FSM gesetzt. Wenn $u_i$ den vollen Wertebereich des Wandlers ausschöpft, ist der weitere Ablauf sehr einfach: Die FSM setzt Bit für Bit im SAR bis zum LSB[16] und die Wandlung ist beendet. Kippt jedoch zwischendurch der Komparator, bedeutet dies, daß der rückgeführte Wert zu groß ist. Die FSM nimmt dann das gerade gesetzte Bit wieder zurück und setzt das nächste niederwertigere. Auch hierbei ist die Wandlung mit Erreichen des LSB beendet. Der beschriebene Ablauf kann auch leicht für den Extremfall, wenn $u_i$ gerade nur so groß wie das LSB ist, nachvollzogen werden.

Da in allen Fällen in jedem Taktschritt ein Bit des Ausgangssignals „abgehakt" wird, ist stets nach *N* Takten ein Wandelvorgang abgeschlossen. Für eine Auflösung von 12 bit werden demnach exakt 12 Taktzyklen benötigt. Trotz der verfahrensbedingten Schnelligkeit hat die Wandelrate auch beim SAR-Verfahren Grenzen, d.h. die maximale Taktfrequenz für die FSM wird ungefähr durch den Kehrwert der Summe der Verzögerungszeiten von FSM, SA-Register, D/A-Wandler und Komparator vorgegeben. SAR-Wandler ermöglichen heute Wandelraten im Bereich von 500 kHz bis über 20 MHz bei Auflösungen von 12...16 bit. Sie decken das größte Anwendungsspektrum auf allen Gebieten der Technik ab.

#### 4.6.3.1.2 Einführung in die Arbeitsweise von Pipelining

Pipelining ist eine für die gesamte Mikrorechnertechnik bedeutsame Strategie. Es wird uns z.B. bei der Befehlsabarbeitung (Instruktionspipelining) oder beim Pipeline-Multiplizierer wieder begegnen. Dem Pipelining liegen folgende Prinzipien zugrunde:

- Pipelining läuft stets nach Maßgabe eines Taktes in mehreren Schritten ab
- In jedem dieser Schritte wird eine (Teil-) Funktion autonom durchgeführt
- Für die autonomen Funktionen sind voneinander unabhängige Komponenten nötig
- Die unabhängigen Komponenten verarbeiten parallel verschiedene Daten
- Mit jedem Taktschritt wird *nach Füllung* der Pipeline ein Ergebnis geliefert

[15] Most Significant Bit
[16] Least Significant Bit

Eine *n*-stufige Pipeline benötigt somit ***n*** autonome Funktionseinheiten, die möglichst gleich lange Durchlaufzeiten für Signale aufweisen sollten. Nach *n* Taktschritten ist die Pipeline gefüllt und liefert mit jedem Takt ein komplettes Ergebnis, obwohl zu dessen Erzeugung immer genau *n* Takte benötigt werden. Das ist möglich, da aufgrund der parallel vorhandenen Funktionseinheiten stets ***n*** verschiedene aufeinanderfolgende Operanden gleichzeitig in der Bearbeitung sind. Der Aufwand für Pipelining steckt also in der Parallelisierung, wobei die parallel ablaufenden Teiloperationen jedoch einfach gestaltet werden können. In Bild 4.61 ist die Anwendung des Pipelining-Prinzips zu A/D-Wandlung dargestellt, womit ein Kompromiß zwischen der Geschwindigkeit einer Direktwandlung (Flash) und der schrittweisen Erzeugung des digitalen Ausgangssignals erzielt wird. Pipelining-A/D-Wandler ermöglichen Wandelraten im Bereich 50...150 MHz bei Auflösungen von 10...14 bit.

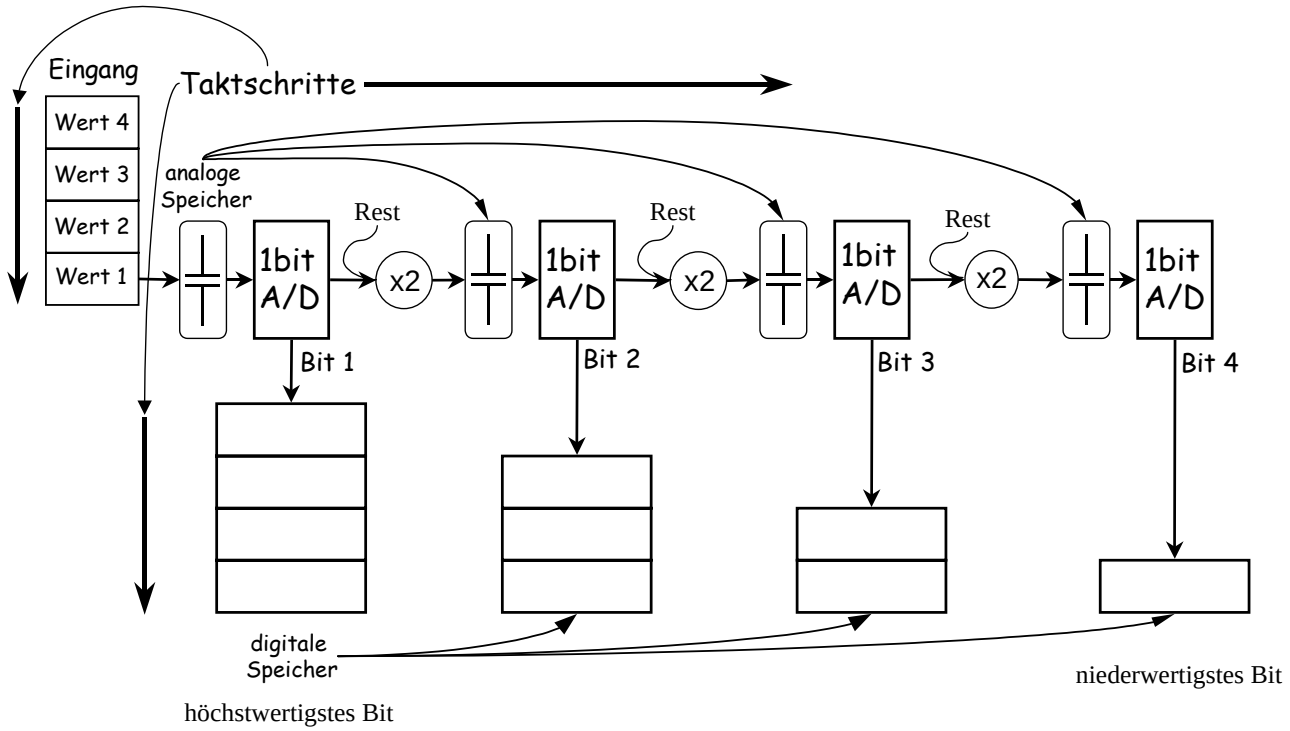



Bild 4.61: Pipelining-Prinzip zur A/D-Wandlung am Beispiel eines 4bit-Wandlers

Aus Bild 4.61 ist ersichtlich, daß die 4bit-Wandlung auf genau vier Arbeitsschritte aufgeteilt wird, wobei in jeder Stufe jeweils ein Bit „bearbeitet“ wird. Die Abtastwerte des zu wandelnden Analogsignals gelangen nach Maßgabe eines Taktes der Reihe nach auf den ersten (am weitesten links stehenden) Analogspeicher, der in Form einer Sample&Hold-Schaltung realisiert ist. In den mit 1bit-A/D bezeichneten Blöcken wird vom anstehenden Analogsignal eine Referenzspannung $U_{ref}$ abgezogen, die der halben maximalen Eingangsspannung $\mathbf{(1/2 \cdot U_{imax})}$ entspricht. Wenn die Differenz (Rest) positiv ist, wird das zugeordnete Bit auf ’1’ gesetzt und der Rest wird an den folgenden Verdoppler (x2) weitergegeben, so daß der jeweils verdoppelte Rest im nachfolgenden Analogspeicher festgehalten wird. Ist dagegen die Differenz negativ, wird das Bit auf ’0’ gesetzt und das analoge Eingangssignal ohne Abzug an den folgenden Verdoppler gegeben. Die folgende Bildserie erläutert, wie die Pipeline schrittweise gefüllt wird und die Wandelergebnisse entstehen.

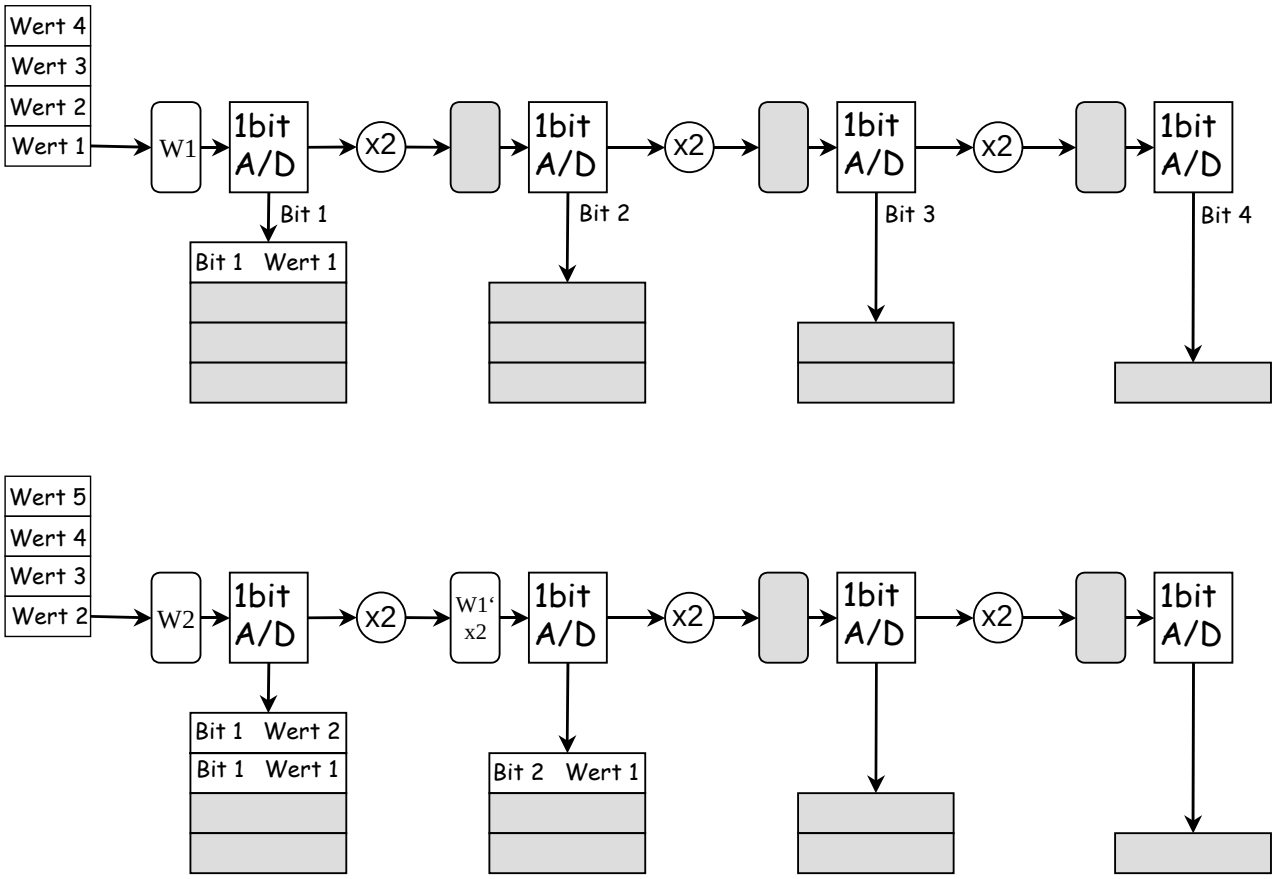



Bild 4.62: Die beiden ersten Schritte zur Füllung der Pipeline (graue Felder bezeichnen noch undefinierte Speicherinhalte, d.h. hier erfolgte noch keine Wertzuweisung)

In Bild 4.62 (oben) sieht man, wie der erste Abtastwert (W1) des zu wandelnden Analogsignals bearbeitet wird. Hieran ist zunächst nur die erste Stufe beteiligt. Wie oben beschrieben, wird dabei das höchstwertige Bit (Bit 1) von Wert 1 generiert. In Bild 4.62 (unten) erkennt man, wie nun der zweite Abtastwert (W2) in der ersten Stufe bearbeitet wird, während aus dem verdoppelten Rest des ersten (W1’x2) in der zweiten Stufe in gleicher Weise wie oben geschildert das zweithöchste Bit (Bit 2) von Wert 1 erzeugt wird.

In Bild 4.63 (oben) ist gezeigt, wie ein weiterer Eingangssignalabtastwert (W3) in der ersten Stufe bearbeitet wird, während die Reste der Vorgänger (W1 und W2) in Stufe 2 nach Verdopplung (W2’x2) bzw. in Stufe 3 nach Vervierfachung (W1’’x4) die entsprechenden niederwertigeren Bits liefern.

Mit dem Einlesen von Wert 4 in die erste Stufe – s. Bild 4.63 (unten) – ist die Pipeline gefüllt, d.h. es gibt keine grauen Felder mehr. Der nach drei Stufen verbliebene Rest von W1 hat nun nach Verachtfachung (W1’’’x8) die letzte Stufe erreicht, die das niederwertigste Bit liefert. Wie man sieht stehen jetzt auch in allen digitalen Speichen die zu Wert 1 gehörigen Bits in der untersten Zeile zur Verfügung.

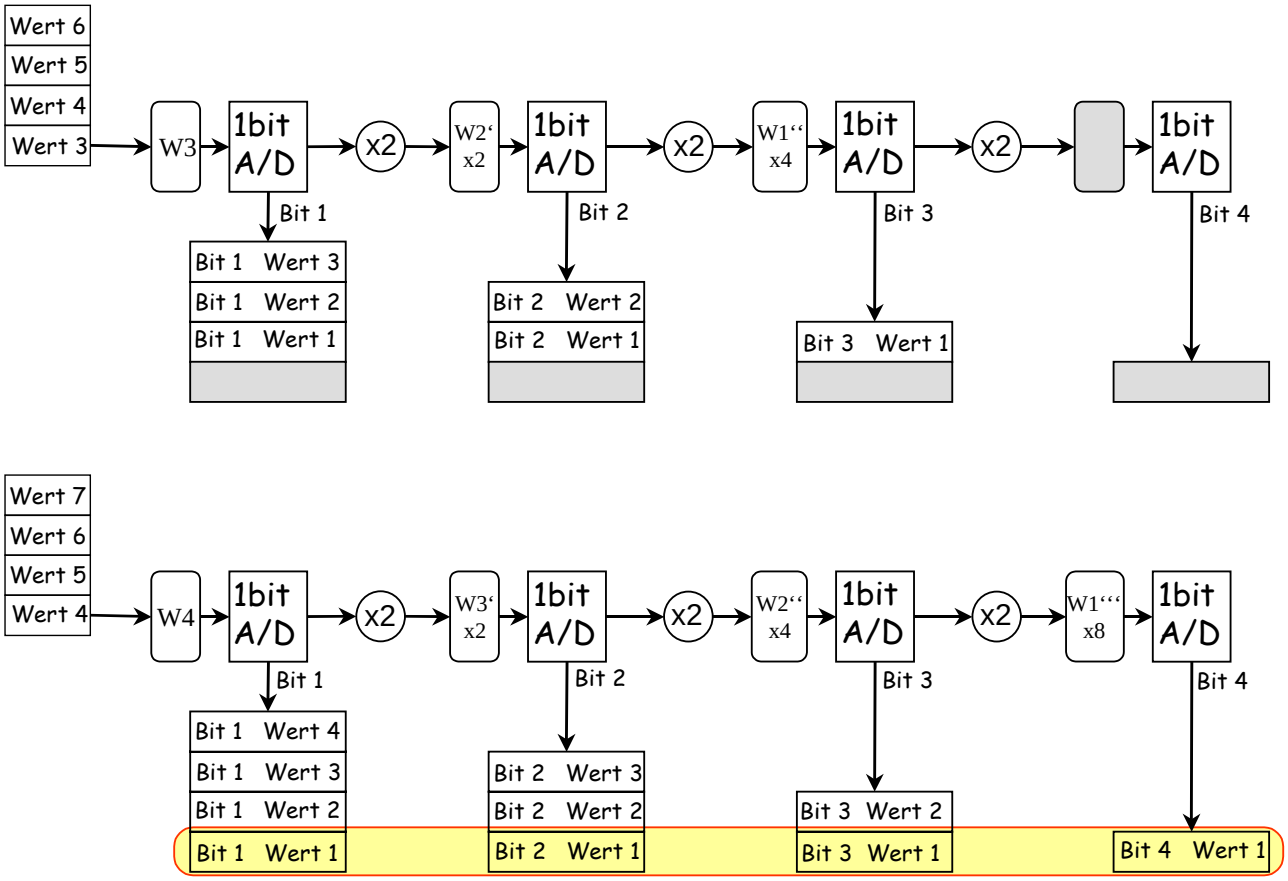



Bild 4.63: Die beiden letzten Schritte zur Füllung der Pipeline liefern das erste Wandelergebnis

Der weitere Ablauf liefert nun mit jedem Taktschritt ein neues komplettes Wandelergebnis, d.h. während Wert 5 an den Eingang gelangt, steht das digitale Ergebnis von Wert 2 zur Verfügung – usw.

Die parallele Hardware ermöglicht also im gezeigten Beispiel eine Vervierfachung der Arbeitsgeschwindigkeit, obwohl jeder Wert nach wie vor vier Schritte zur Bearbeitung durchläuft. Der entscheidende „Trick“ beim Pipelining ist, daß die Hardware verschiedene Werte gleichzeitig bearbeiten kann. Im *weiterführenden Ergänzungsskript* ist die technische Realisierung eines Pipeline-A/D-Wandlers und ein komplexeres Wandelbeispiel zur weiteren Vertiefung dargestellt.

#### 4.6.3.1.3 Sigma-Delta-Prinzip zur A/D-Wandlung

Im *weiterführenden Ergänzungsskript* wird der Vollständigkeit halber auch das Sigma-Delta-Prinzip zur A/D-Wandlung kurz vorgestellt, weil Sigma-Delta-Wandler als monolithisch integrierte Bestandteile moderner Mikrorechner zunehmend an Bedeutung gewinnen. Man findet sie heute sowohl in einfachen 8 bit–Mikrocontrollern als auch in komplexen digitalen Signalprozessoren.

Die theoretischen Grundlagen zum Verständnis der Arbeitsweise von Sigma-Delta-Wandlern fehlen an dieser Stelle noch. Deshalb wurde eine vereinfachte qualitative Beschreibung gewählt.

### 4.6.3.2 Grundlagen der D/A-Wandlung

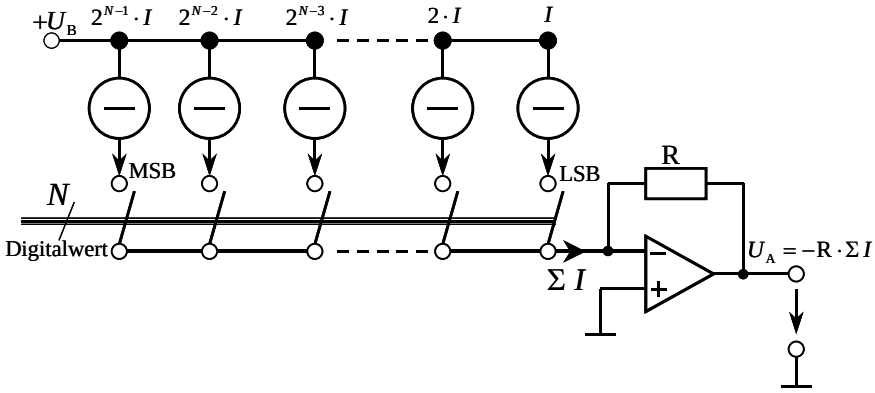



Bild 4.64: Prinzip der D/A-Wandlung mit Stromquellen

Die Funktionsweise eines D/A-Wandlers ist deutlich einfacher zu verstehen als die eines A/D-Wandlers. Die technische Realisierung kann jedoch bei hohen Anforderungen an Geschwindigkeit und Genauigkeit sehr komplex und aufwendig werden. Bild 4.64 verdeutlicht das Prinzip eines einfachen stromquellenbasierten Wandlers. Die Hauptbestandteile sind *N* Stromquellen, deren Ströme in Zweierpotenzen abgestuft sind. Nach Maßgabe eines *N* bit breiten Digitalwertes

$$D_N = \sum_{i=0}^{N-1} b_i \cdot 2^i \,, \quad \text{mit } b_i \in \{0,1\} \tag{4.29}$$

werden die eingezeichneten Schalter betätigt, d.h. geschlossen falls $b_i$=1 vorliegt und geöffnet für $b_i$=0. Alle Ströme, die über die Schalter kommen, fließen auf den – Eingang des Operationsverstärkers[17] und dessen Ausgangsspannung hat den Wert

$$-U_A = R \cdot I \cdot \left[ \sum_{i=0}^{N-1} b_i \cdot 2^i \right]. \tag{4.30}$$

#### 4.6.3.2.1 R/2R-Wandler, deren Kernstück ein einfaches Widerstandsnetzwerk ist

Bei dem in Bild 4.65 dargestellten R/2R-Wandlerprinzip, das seinen Namen vom dem Widerstandsnetzwerk hat, in dem nur zwei verschiedene Werte vorkommen, die zudem exakt im Verhältnis **1:2** stehen, wird wiederum am virtuellen Nullpunkt eines Operationsverstärkers eine Stromsumme gebildet, so daß am Ausgang eine dazu proportionale Spannung $U_A$ auftritt. Im Unterschied zu Bild 4.64 haben die durch die einzelnen Bits des Digitalwertes betätigten Schalter hier zwei Positionen, d.h. bei $b_i$=0 besteht eine Verbindung zur Masse, die den Strom aufnimmt, und bei $b_i$=1 fließt der Strom auf den virtuellen Nullpunkt des OPV.

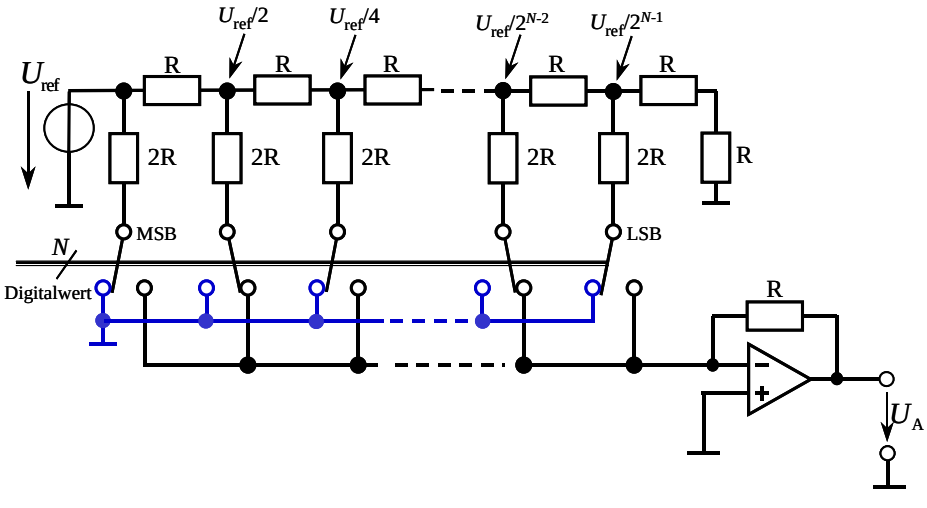



Bild 4.65: D/A-Wandler nach dem R/2R-Prinzip

Auf diese Weise sind Spannungs- und Stromverhältnisse im Netzwerk stets unabhängig von den Schalterstellungen.

[17] Arbeitsweise ist im *weiterführenden Ergänzungsskript* näher beschrieben

Wie man aus Bild 4.65 erkennt, liefert das LSB z.B. den Strom

$$I_{LSB} = \frac{U_{\text{ref}}}{2^{N-1} \cdot 2\text{R}} = \frac{U_{ref}}{2^N \cdot R}, \tag{4.31}$$

und das MSB

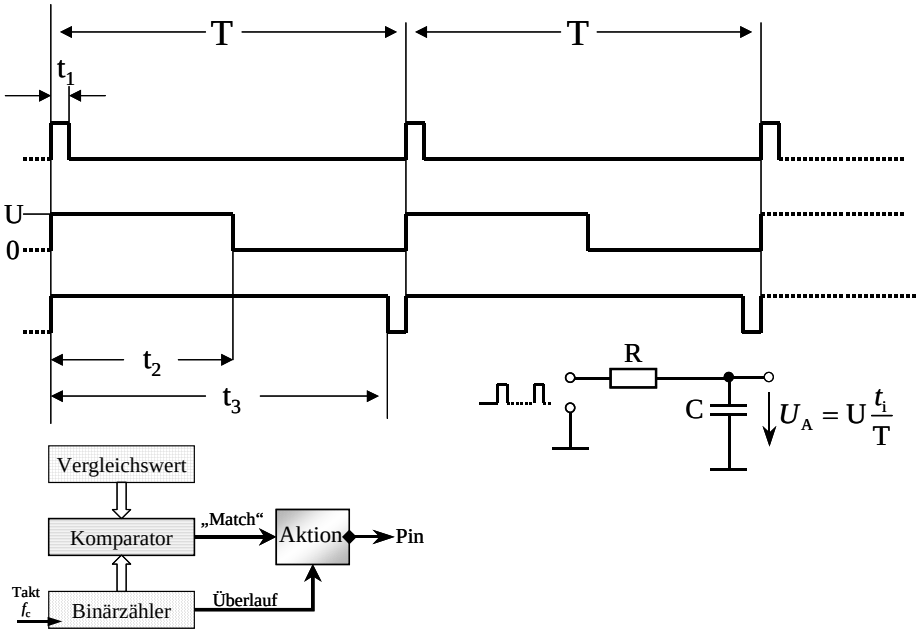
$$I_{MSB} = \frac{U_{ref}}{2R}. \tag{4.32}$$

Für einen *N* bit breiten Digitalwert nach (4.29) erhält man dann analog zu (4.30)

$$-U_{\text{A}} = R \cdot \left[ \sum_{i=0}^{N-1} \frac{U_{\text{ref}}}{2^N \cdot R} b_{\text{i}} \cdot 2^i \right] = \sum_{i=0}^{N-1} \frac{U_{\text{ref}}}{2^N} b_{\text{i}} \cdot 2^i \,. \tag{4.33}$$

#### 4.6.3.2.2 Pulsweitenmodulation zur D/A-Wandlung

Die Pulsweitenmodulation (PWM) ist wohl das einfachste Verfahren zur Digital/Analogwandlung. Der entscheidende Nachteil ist jedoch, daß man keine hohen Wandelraten erzielen kann. Die Anwendung der PWM ist deshalb auf Gebiete beschränkt, wo es nicht so sehr auf Geschwindigkeit, sondern auf kostengünstige Lösungen ankommt, die aber durchaus sehr hohe Auflösungen bereitstellen können.




Ein PWM liefert ein Rechtecksignal mit der Periodendauer T, bei dem das Tastverhältnis $t_i/T$ in feinen Stufen veränderbar ist. Die drei in Bild 4.66 dargestellten Muster verdeutlichen das Grundprinzip. Das Rechtecksignal mit der Amplitude U wird in der Regel auf ein einfaches RC-Tiefpaßfilter – s. unten rechts in Bild 4.66 – gegeben.

Bild 4.66: Prinzip der Pulsweitenmodulation

Am Kondensator baut sich dann eine Spannung $U_A$ auf, die linear mit dem Tastverhältnis $t_i/T$ wächst. Damit an C eine „glatte“ Spannung auftritt, muß die Zeitkonstante R·C hinreichend groß gewählt werden, d.h. R·C >> T. Bei zu kleiner Zeitkonstante treten „Rippel“ mit der Frequenz der Rechteckspannung auf, die dem gewünschten Gleichanteil (Mittelwert) überlagert sind, und die zu Störungen in nachfolgenden Schaltungen führen können. Diese benötigte Filterung ist hauptsächlich für die Langsamkeit der D/A-Wandlung auf PWM-Basis verantwortlich.

Variable Rechteckfolgen gemäß Bild 4.66 Art lassen sich einfach und flexibel mit den integrierten Timersystemen von Mikrorechnern erzeugen. Eine vereinfachte Prinzipdarstellung der Hardware ist

unten links im Bild zu sehen. Es wird ein Binärzähler ein Vergleichswertregister und ein digitaler Komparator benötigt. Der Zähler läuft im einfachsten Fall mit einem Takt der Frequenz $f_c$ von Null bis zum Maximalwert (111...) und beginnt nach Überlauf wieder mit Null, so daß man bei $N$ Binärstellen einen Zyklus der Dauer $T=2^N \cdot 1/f_c$ erhält. Wenn mit jedem Überlauf z.B. das Setzen eines Portpins auf „1“-Pegel verknüpft ist, lassen sich Folgen nach Bild 4.66 leicht erzeugen. Dabei übernimmt der Komparator die Aufgabe, nach jedem Takt den Zählerinhalt mit dem Vergleichswertregister zu vergleichen und bei Übereinstimmung („Match“) ein Rücksetzen des Portpins auf „0“ zu veranlassen. Im Timersystem eines Mikrorechners kann das Vergleichwertregister mit jedem beliebigen Wert, den der Zähler erreichen kann, geladen werden, so daß die Impulsdauer $t_i$ in weiten Grenzen verstellt werden kann. Des weiteren ist die Taktfrequenz in gewissem Rahmen variabel, und der Binärzahler muß bei Überlauf nicht bei Null beginnen, sondern mit einem sogenannten Reload-Wert, der per Programm vorgegeben wird. So erhält man die Möglichkeit, die Dauer $T$ in weiten Grenzen zu variieren – s. die Beispiele in Bild 4.67 . Heute enthalten schon sehr einfache 8 bit-Mikrocontroller PWM-Einheiten, die mit Taktfrequenzen bis zu einigen 10 MHz bei 16 bit Zählerlänge betrieben werden können, so daß man eine Auflösung von 16 bit, allerdings nur bei Wandelraten unter 1MHz realisieren kann.

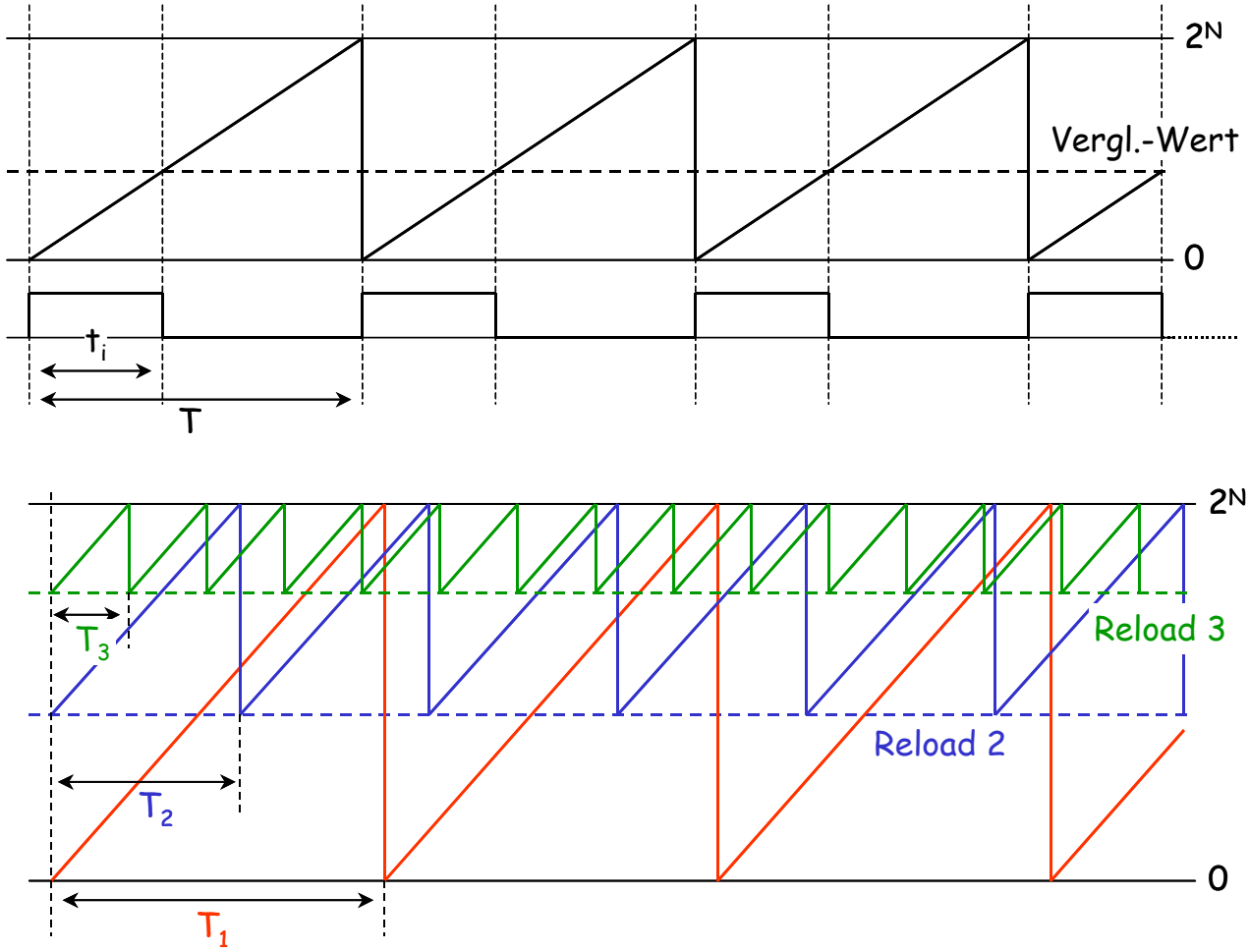



Bild 4.67: Timerfunktionen für die Pulsweitenmodulation

Die Pulsweitenmodulation zur D/A-Wandlung hat sich schon sehr früh in der Unterhaltungselektronik bei der Fernbedienung von Rundfunk und Fernsehgeräten durchgesetzt. Für Lautstärke, Helligkeit, Kontrast, Farbtiefe etc. benötigt man hinreichend feinstufig verstellbare Spannungen, die im Empfangsteil einer Fernbedienung mit PWM leicht erzeugt werden können. Änderungsbefehle vom Fernbedienungssender werden in digitaler Form übermittelt und müssen nur entsprechende Vergleichswerte – s Bild 4.66 unten links – verändern.

Bei Standard-Mikrocontrollern können zahlreiche Portpins (z.B. 8, 16 oder 32) gleichzeitig als PWM-Ausgänge konfiguriert werden. Damit ergeben sich vielfältige Möglichkeiten zur Ansteuerung von Schrittmotoren, d.h. ein ganz anderes Gebiet von PWM-Anwendungen. Darüber hinaus finden sich in der modernen Antriebstechnik und Leistungselektronik weiter, zum Teil sehr komplexes Einsatzfelder der PWM. So lassen sich z.B. aus der gleichgerichteten Netzwechselspannung mittels PWM Drehstromsysteme variabler Frequenz generieren, mit denen man sehr robuste und präzise Antriebe mit variabler Drehzahl auf der Basis von Asynchronmaschinen realisieren kann.

Die Stromversorgung elektrischer und elektronischer Geräten aller Art ist heute wohl das umfangreichste Einsatzgebiet der PWM, denn sämtliche Schaltnetzteile beruhen auf dem PWM-Prinzip.

## 4.7 Integrierte Peripheriekomponenten in Mikrorechnern und ihre Steuerung über Spezialfunktionsregister (Memory Mapping)

Anhand von Bild 4.52 sowie Tabelle 4.7 und Tabelle 4.8 wurde bereits das Konzept der Abbildung integrierter Peripheriekomponenten von Mikrorechnern im Datenspeicherbereich – das sogenannte „Memory Mapping" - einführend behandelt. Im weiteren werden einige wichtige Peripheriekomponenten und ihre Steuerung bzw. Programmierung über die zugeordneten Spezialfunktionsregister detailliert betrachtet.

### 4.7.1.1 Mikrorechner-Portstrukturen am Beispiel der 8051-Familie

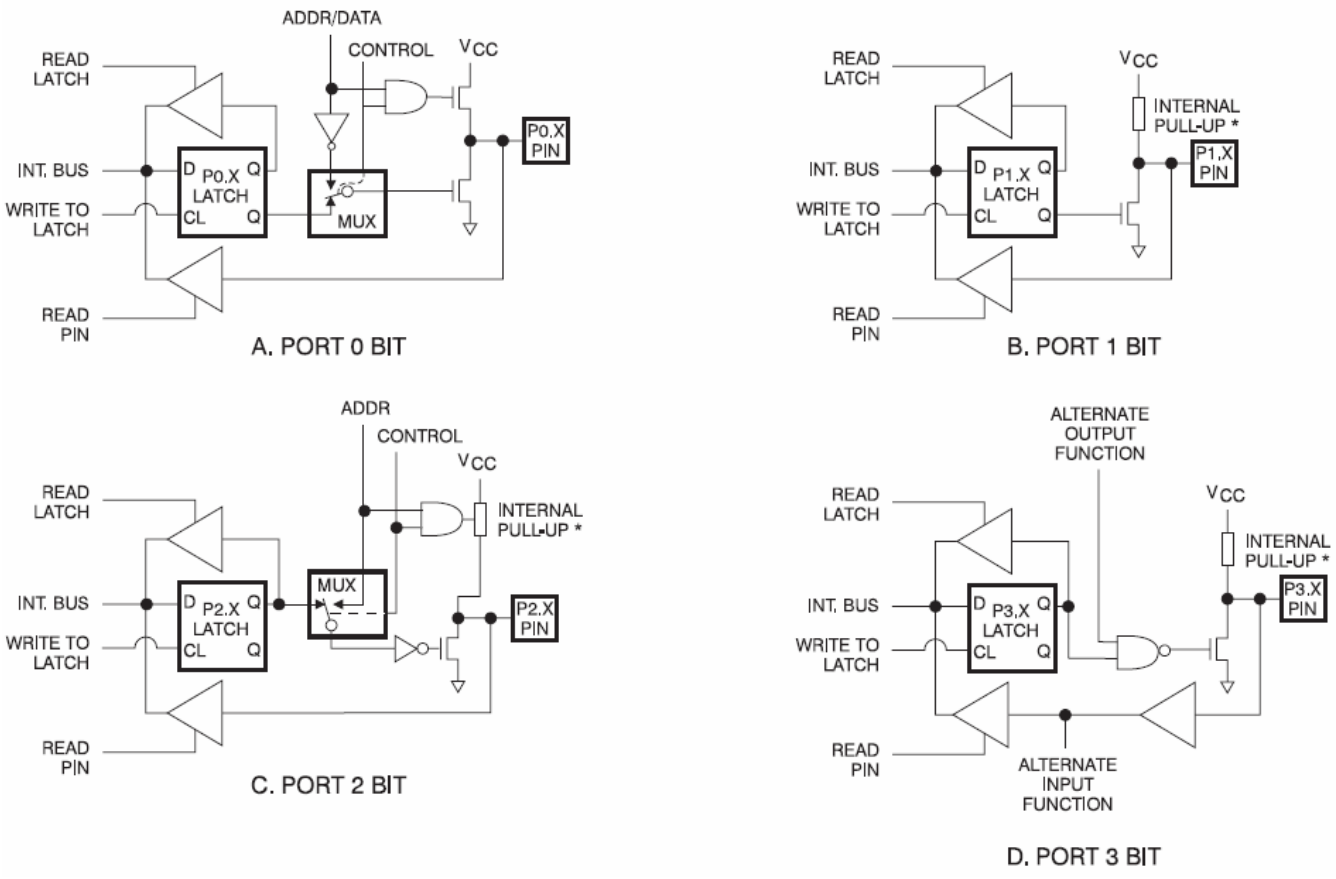



Bild 4.68: Übersicht der Portpinfunktionen bei 80C5x-Mikrocontrollern

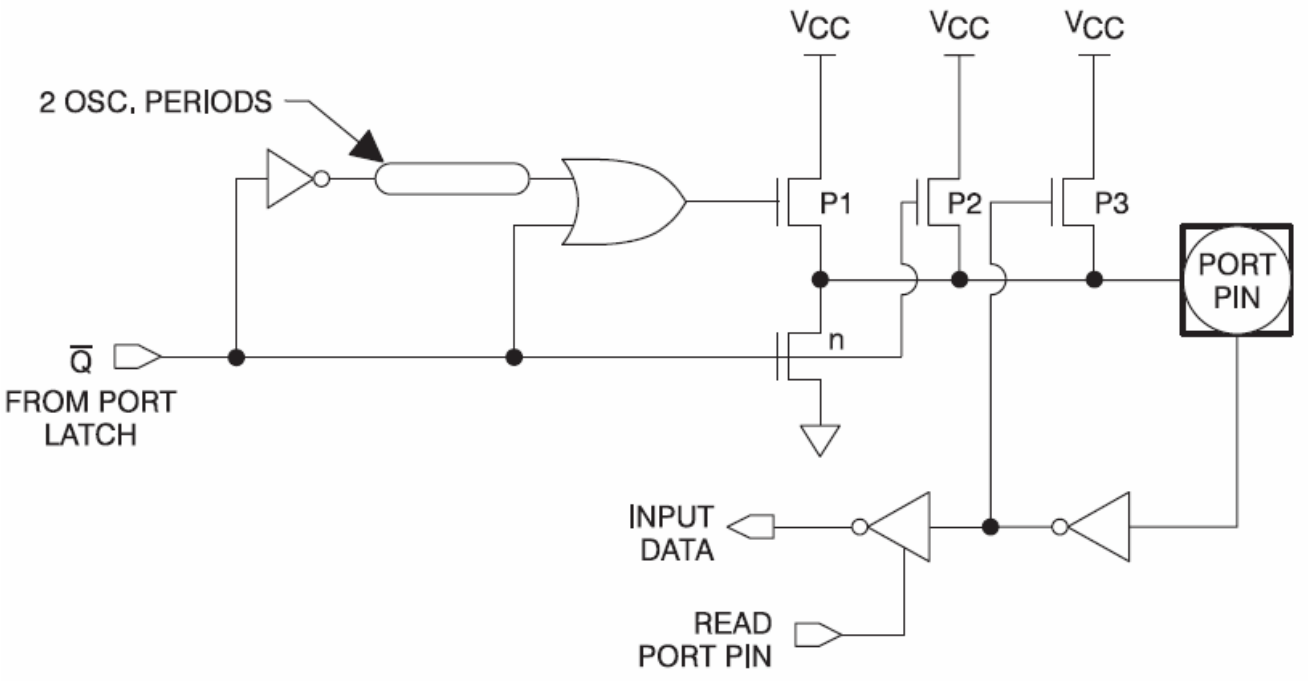



Bild 4.69: Zur Erläuterung des elektrischen Verhaltens eines Portpins (CMOS)

### 4.7.1.2 Timersysteme und Timerfunktionen

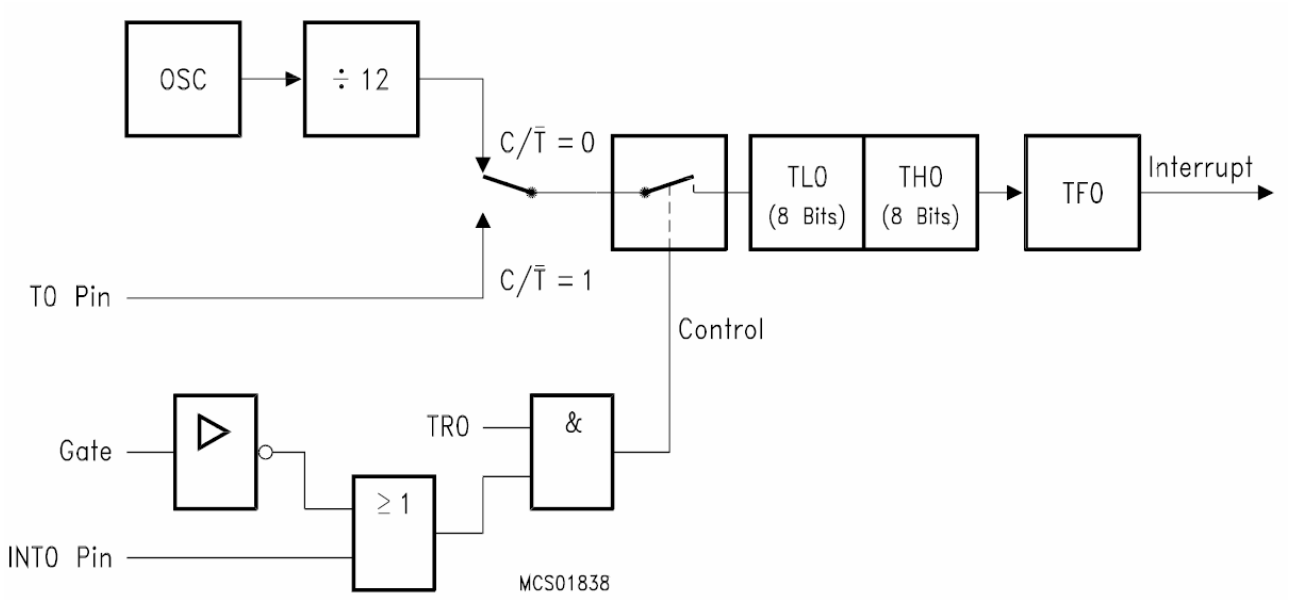



Bild 4.70: Blockschaltbild zur Timerfunktion von T0 in der Betriebsart 1

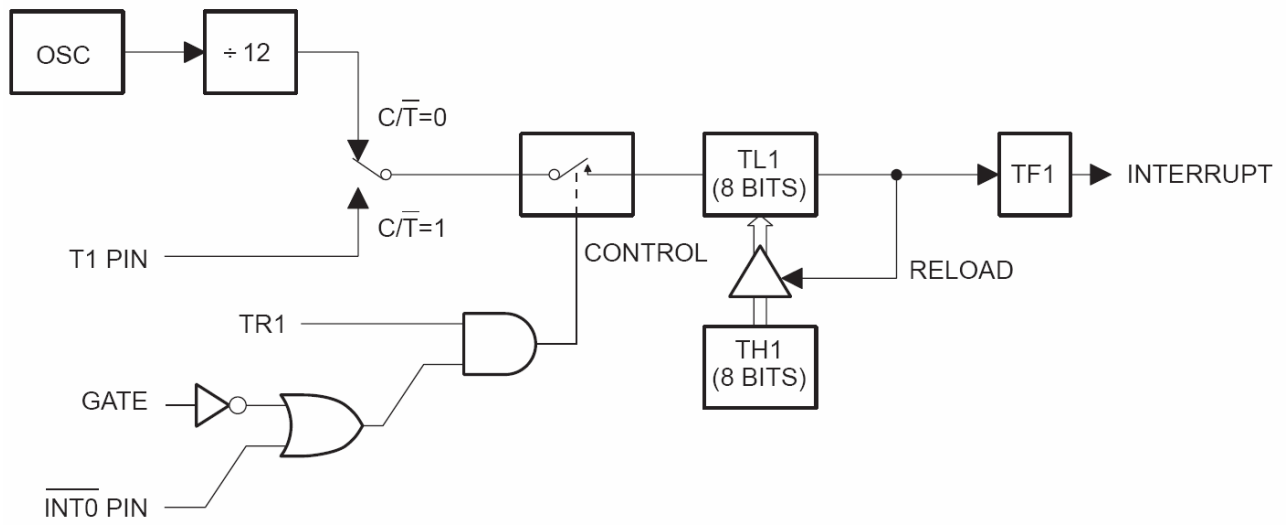



Bild 4.71: Blockschaltbild zur Funktion von Timer T0 in der Betriebsart 2

## TMOD das Timer/Counter-Betriebsarten-Register (089H ≡ 137)

Bit-Nr.

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| Gate | C/T | M1 | M0 | Gate | C/T | M1 | M0 |
| Timer 1 | | | | Timer 0 | | | |

| M1 | M0 | Betriebsart | Funktion |
|---|---|---|---|
| 0 | 0 | 0 | 13bit Zähler |
| 0 | 1 | 1 | 16bit Zähler |
| 1 | 0 | 2 | 8bit Zähler mit Auto-Reload |
| 1 | 1 | 3 | teilt Timer 0 in 2 8bit-Timer |

C/T wählt Zähler- oder Zeitgeber-Funktion aus

| C/T | Funktion |
|---|---|
| 1 | Zähler, der die Rückflanken an PIN T0 bzw. T1 zählt |
| 0 | Zeitgeber, der (quarzgenau) Maschinenzyklen zählt |

GATE erlaubt Bedienung (Start/Stopp) des Timers über eine MC-PIN

| GATE | Funktion |
|---|---|
| 1 | T0 bzw. T1 zählen nur, wenn INT0 bzw. INT1 PIN auf “1“ |
| 0 | T0 bzw. T1 werden allein durch ihre TR0, TR1 Bits bedient |

Nach Reset ist TMOD (00000000).

**TCON das Timer/Counter Control Register (088H ≡ 136)**

Bit-Nr.

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| TF1 | TR1 | TF0 | TR0 | IE1 | IT1 | IE0 | IT0 |
| Timer1/0 | | | | externe Interrupts | | | |

TF1/0 Timer-Überlauf-Flags, die beim jeweiligen Timer-Überlauf per Hardware gesetzt werden, und wieder gelöscht werden, sobald der MC die zugehörige Interrupt-Bedienungsroutine beginnt

TR1/0 Bedienung (Start/Stopp) des Timers

| TR1/0 | Funktion |
|---|---|
| 1 | T0 bzw. T1 zählen |
| 0 | T0 bzw. T1 stehen |

IE1/0 Flags der externen Interrupts an den MC-PINs INT1, INT0

| IE1/0 | Funktion |
|---|---|
| 1 | falls IT1/0=„1“ und ein 1→0 Übergang an INT1 bzw. INT0 |
| 0 | mit Beginn der entsprechenden Interrupt-Bedienungsroutine |

IT1/0 bestimmt, ob ein Interrupt an INT0/1 flanken- oder pegelgetriggert ist

| IT1/0 | Funktion |
|---|---|
| 1 | flankengetriggert, auf 1→0 Übergang an INT1 bzw. INT0 |
| | pegelgetriggert „0“ löst Interrupt aus |

Nach Reset ist TCON (00000000)

## Der Timer T2 und Beispiele für seine Betriebsarten

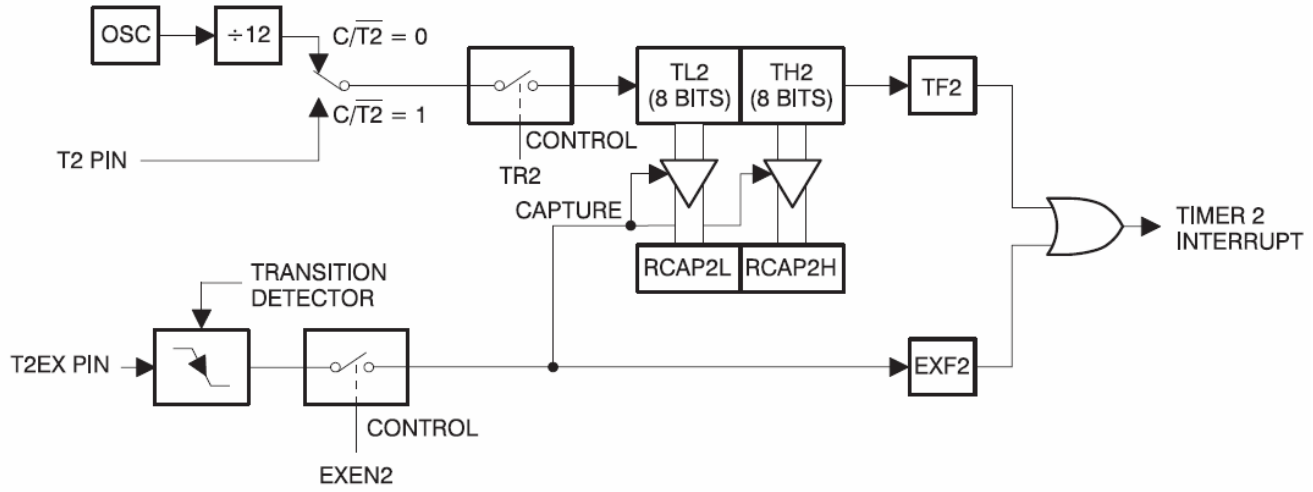



Bild 4.72: Funktions-Blockschaltbild für Timer 2 im „Capture“-Betrieb

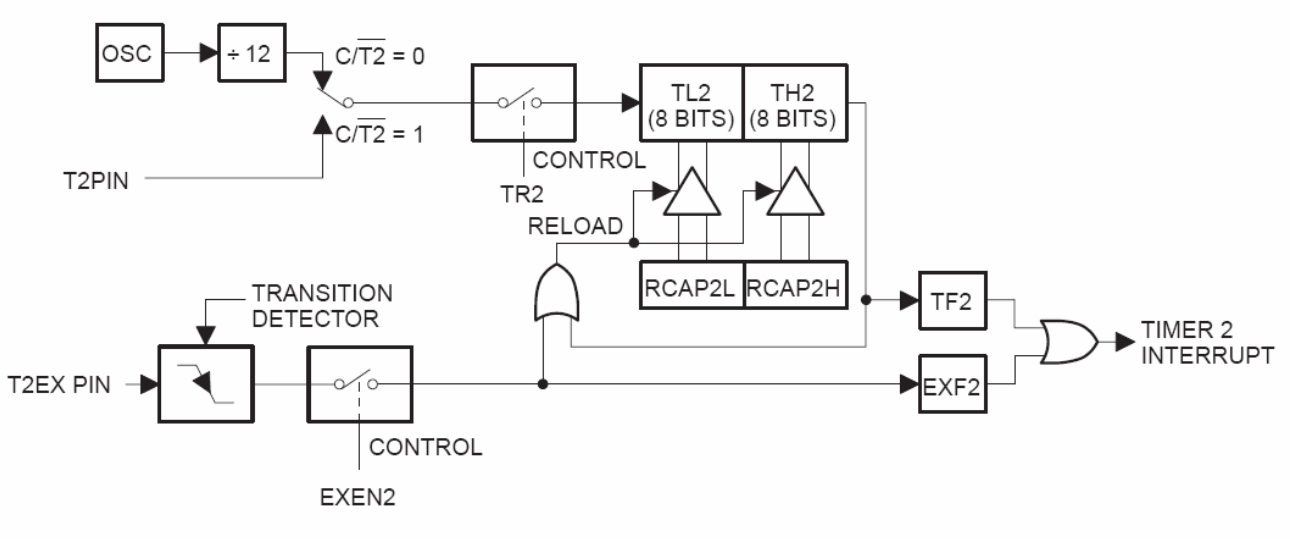



Bild 4.73: Funktions-Blockschaltbild für Timer 2 im „Auto-Reload“-Betrieb

## T2CON das Timer/Counter-2-Steuerregister (0C8H)

Bit-Nr.

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| TF2 | EXF2 | RCLK | TCLK | EXEN2 | TR2 | C/T2 | CP/RL2 |

TF2 Timer 2 Überlauf-Flag; wird bei Timer 2 Überlauf gesetzt und per Software zurückgesetzt.
TF2 kann nicht gesetzt werden, wenn entweder RCLK oder TCLK „1“ ist.

EXF2 externes Timer 2 Flag; wird gesetzt, wenn durch eine Rückflanke an Pin T2EX (Pl.5) entweder ein ***Capture*** oder ein ***Reload***-Vorgang ausgelöst wurde. (EXEN2 muß vorher gesetzt worden sein)
EXF2 muß per Software gelöscht werden.

RCLK Flag für den Empfangstakt der seriellen Schnittstelle. In den Betriebsarten (MODES) 1 und 3 der seriellen Schnittstelle werden die Timer-2-Überlauf-Impulse als Basis für den Empfangstakt genommen. Bei RCLK = „0“ werden Timer-l-Überläufe benutzt.

TCLK Flag für den Sendetakt der seriellen Schnittstelle. In den Betriebsarten (MODES) 1 und 3 der seriellen Schnittstelle werden die Timer-2-Überlauf-Impulse als Basis für den Sendetakt genommen. Bei TCLK = „0“ werden Timer-l-Überläufe benutzt.

EXEN2 Flag für die Freigabe (*Enable*) extern ausgelöster Timer-2-Funktionen. Ein ***Capture***- oder ***Reload***-Vorgang wird bei Auftreten einer Rückflanke an Port-Pin Pl.5 (T2EX) ausgeführt, falls Timer 2 nicht als Takt-Basis für die serielle Schnittstelle benutzt wird. Bei EXEN2 = „0“ werden Ereignisse an Pl.5 vom Timer 2 ignoriert.

TR2 Start/Stopp-Bit für Timer 2. TR2= „1“ startet den Timer 2.

C/T2 Auswahl: Zähler oder Zeitgeber. C/T2= „0“ ⇒ Timerfunktion
C/T2= „1“ ⇒ Zähler für externe Ereignisse (Rückflanken) an Portpin P1.7 (T2)

CP/RL2 ***Capture***- bzw. ***Reload***-Flag. Wenn CP/RL2= „1“wird beim Auftreten einer Rückflanke Port-Pin P1.5 (T2EX) ein, ***Capture***-Vorgang ausgeführt, falls EXEN2= „1“ ist.
Wenn CP/RL2= „0“ ist, erfolgt ***Auto-Reload***, entweder bei Timer-2-Überläufen oder bei Rückflanken an Port-Pin Pl.5 (T2EX), falls EXEN2=„1“ ist.
Wenn jedoch eines der Bits RCLK oder TCLK= „1“ ist, wird CP/RL2 ignoriert und Timer 2 führt bei Überlauf stets ***Auto-Reload*** durch.

| (MSB) | | | | | | | (LSB) |
|---|---|---|---|---|---|---|---|
| TF2 | EXF2 | RCLK | TCLK | EXEN2 | TR2 | C/$\overline{T2}$ | CP/$\overline{RL2}$ |

| Symbol | Position | Name and Significance |
|---|---|---|
| TF2 | T2CON.7 | Timer 2 overflow flag set by a Timer overflow and must be cleared by software. TF2 will not be set when either RCLK = 1 or TCLK = 1. |
| EXF2 | T2CON.6 | Timer 2 external flag set when either a capture or reload is caused by a negative transition on T2EX and EXEN2 = 1. When Timer 2 interrupt is enabled, EXF2 = 1 will cause the CPU to vector to the Timer 2 interrupt routine. EXF2 must be cleared by software. |
| RCLK | T2CON.5 | Receive clock flag. When set, causes the serial port to use Timer 2 overflow pulses for its receive clock in Modes 1, 3 and Timer 1 provides transmit baud rate. RCLK = 0 causes Timer 1 overflow to be used for the receive clock. |
| TCLK | T2CON.4 | Transmit clock flag. When set, causes the serial port to use Timer 2 overflow pulses for its transmit clock in Modes 1, 3 and Timer 1 provides transmit baud rate. TCLK = 0 causes Timer 1 overflows to be used for the transmit clock. |
| EXEN2 | T2CON.3 | Timer 2 external enable flag. When set, allows a capture or reload to occur as a result of a negative transition on T2EX if Timer 2 is not being used to clock the serial port. EXEN2 = 0 causes Timer 2 to ignore events at T2EX. |
| TR2 | T2CON.2 | Start/stop control for Timer 2. A logic 1 starts the timer. |
| C/$\overline{T2}$ | T2CON.1 | Timer or counter select. (Timer 2)<br>0 = Internal timer (OSC/12)<br>1 = External event counter (falling edge triggered). |
| CP/$\overline{RL2}$ | T2CON.0 | Capture/Reload flag. When set, captures will occur on negative transitions at T2EX if EXEN2 = 1. When cleared, auto-reloads will occur either with Timer 2 overflows or negative transitions at T2EX when EXEN2 = 1. When either RCLK = 1 or TCLK = 1, this bit is ignored and the timer is forced to auto-reload on Timer 2 overflow. |

Tabelle 4.9: T2CON-Beschreibung in Englisch (T2CON ist bitadressierbar und “00“ nach Reset)

## T2MOD das Timer/Counter-2-Betriebsarten-Register (0C9H)

Bit-Nr.

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| - | - | - | - | - | - | T2OE | DCEN |

T2OE Timer 2 Output-Enable

DCEN erlaubt Timer 2 als Vorwärts/Rückwärts-Zähler zu betreiben

Die Programmierung von Timer 2 erfolgt über 6 Spezialfunktionsregister:

| Bezeichnung | Adresse (hex) | Funktion |
|---|---|---|
| T2CON | C8 | Steuerung |
| T2MOD | C9 | Betriebsarteinstellung |
| RCAP2L | CA | Capture/Reload-Puffer, Low-Byte |
| RCAP2H | CB | Capture/Reload-Puffer, High-Byte |
| TL2 | CC | Timer/Counter 2, Low-Byte |
| TH2 | CD | Timer/Counter 2, High-Byte |

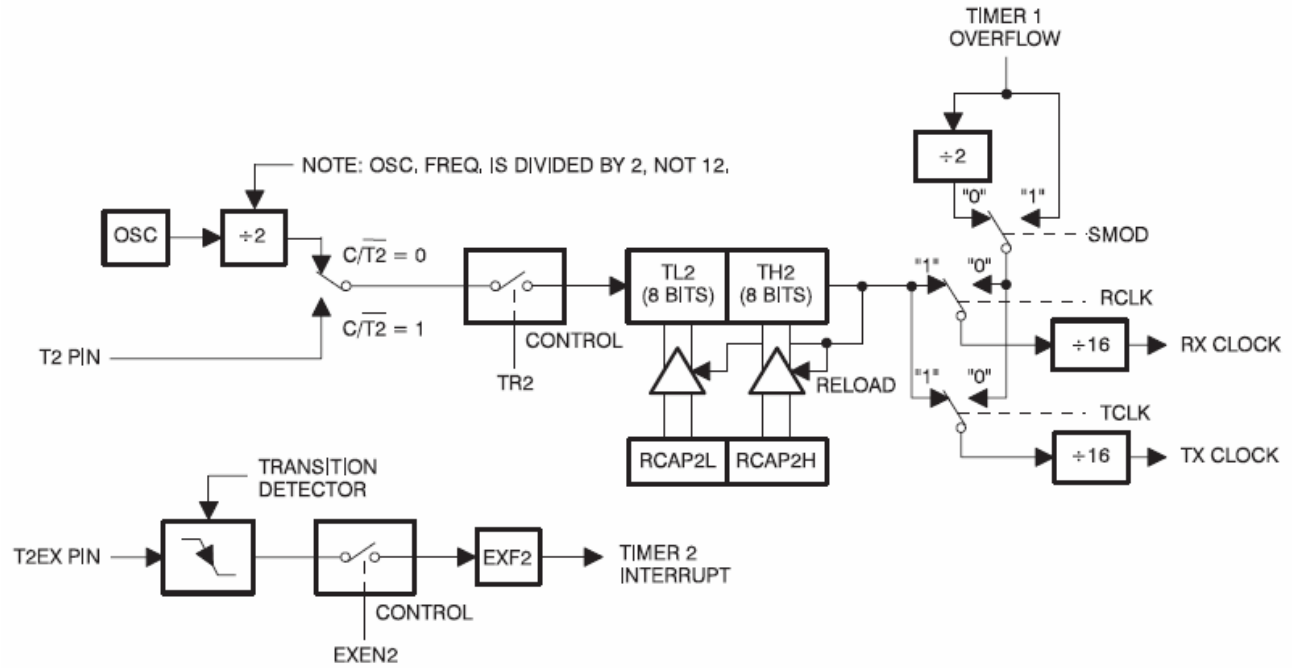



Bild 4.74: Anwendungsbeispiel von Timer 2 als Baudratengenerator für die serielle Schnittsstelle

### 4.7.1.3 Das 8051-Interruptsystem

**IP das Interrupt-Prioritäts-Register (0B8H ≡ 184)**

Bit-Nr. "1" ≡ höhere Priorität

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| – | – | – | PS | PT1 | PX1 | PT0 | PX0 |

PS : Serielle Schnittsstelle
PT1: Timer 1
PX1: Externer Interrupt 1
PT0: Timer 0
PX0: Externer Interrupt 0

Nach Reset ist IP (xxx00000)

## IE das Interrupt-Enable-Register (0A8H ≡ 168)

Bit-Nr. „1“ ≡ freigeben, „0“ ≡ sperren

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| EA | – | – | ES | ET1 | EX1 | ET0 | EX0 |

EA : Alle Interrupts

ES : Interrupts der seriellen Schnittstelle

ET1: Timer 1 Interrupt

EX1: Externer Interrupt 1

ET0: Timer 0 Interrupt

EX0: Externer Interrupt 0

Nach Reset ist IE (0xx00000)

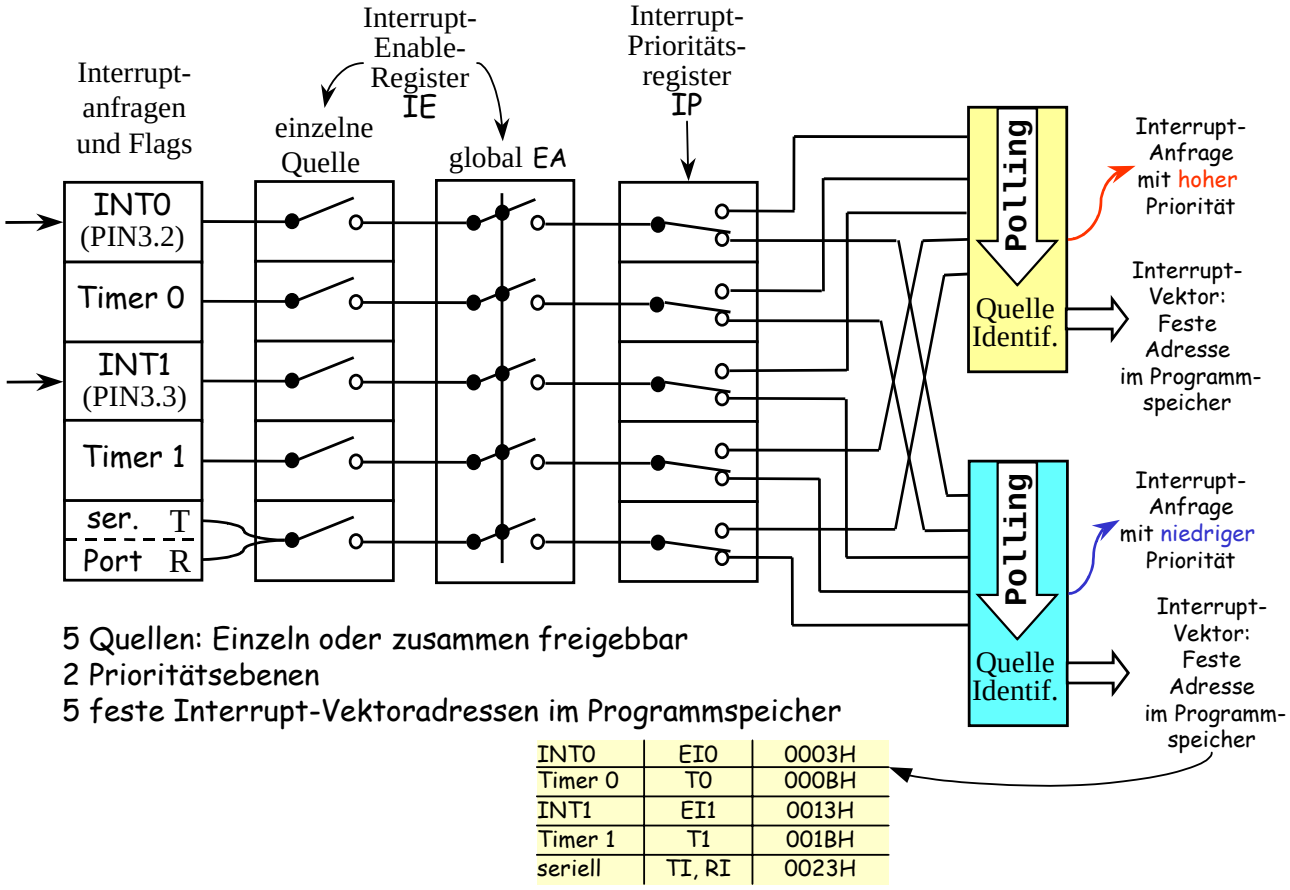



Bild 4.75: Übersicht über das 8051-Interruptsystem

## 4.7.2 Einrichtungen zum Datentransfer: parallele und serielle Schnittstellen

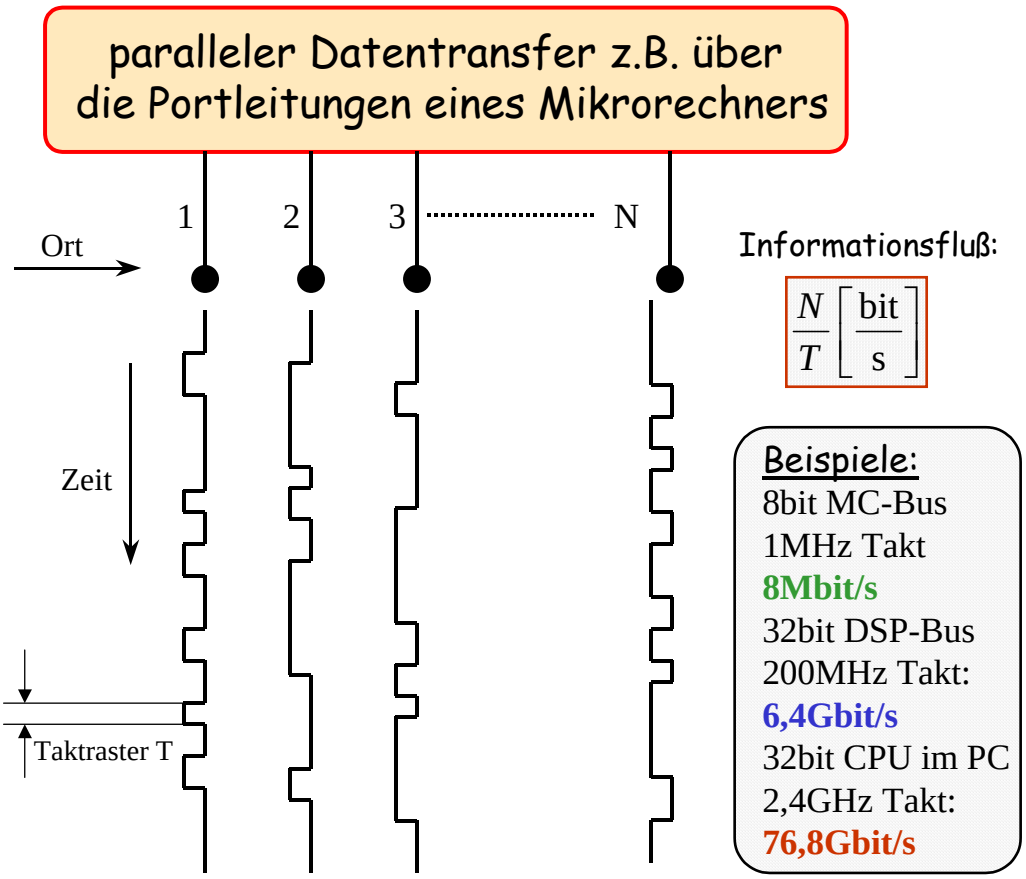



Bild 4.76: Betrachtungen zum Informationsfluß am Parallelport eines Mikrorechners

### 4.7.2.1 Von der parallelen zur zur seriellen Datenübertragung

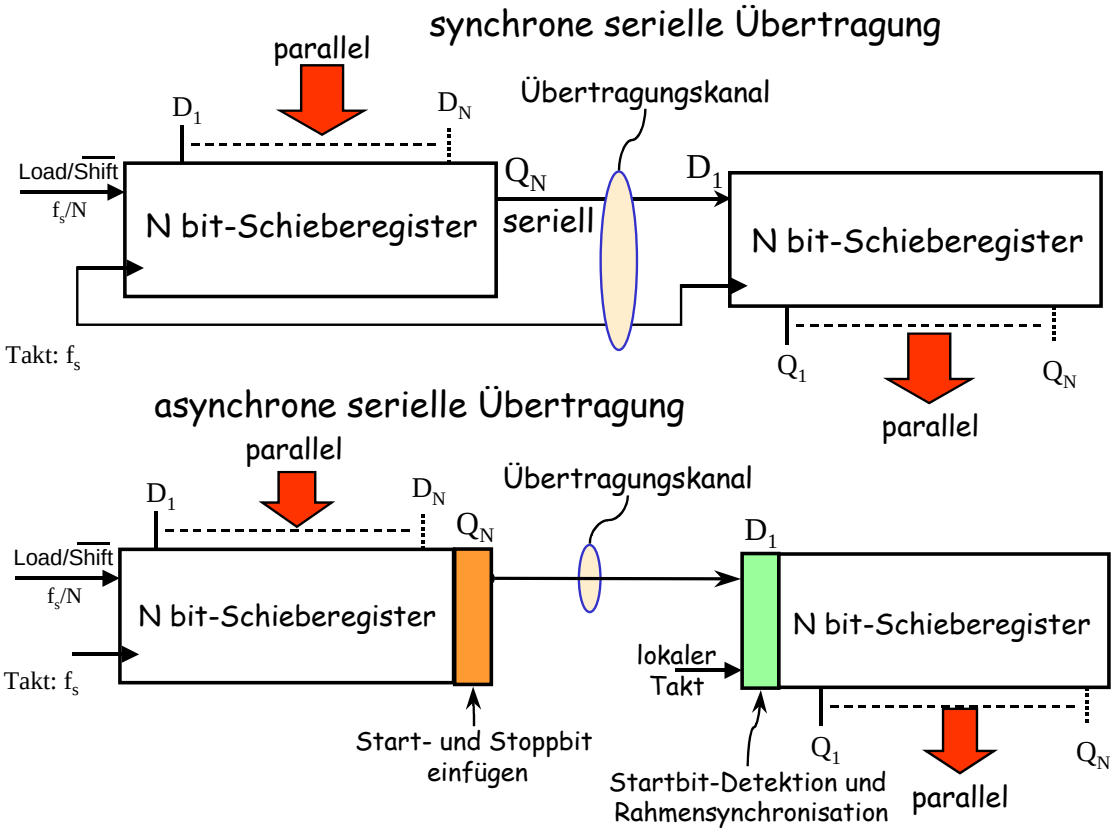



Bild 4.77: Synchrone und asynchrone serielle Datenübertragung

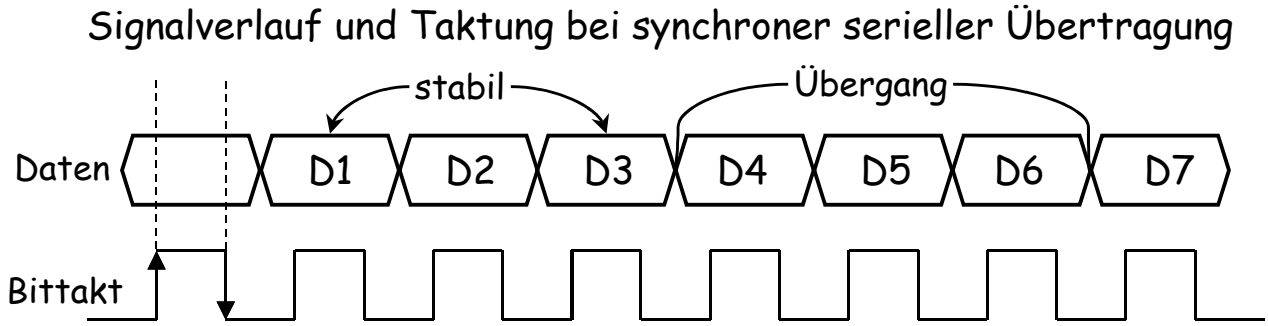



Bild 4.78: Synchroner Datentransfer mit einem übertragenen, gemeinsamen Bittakt

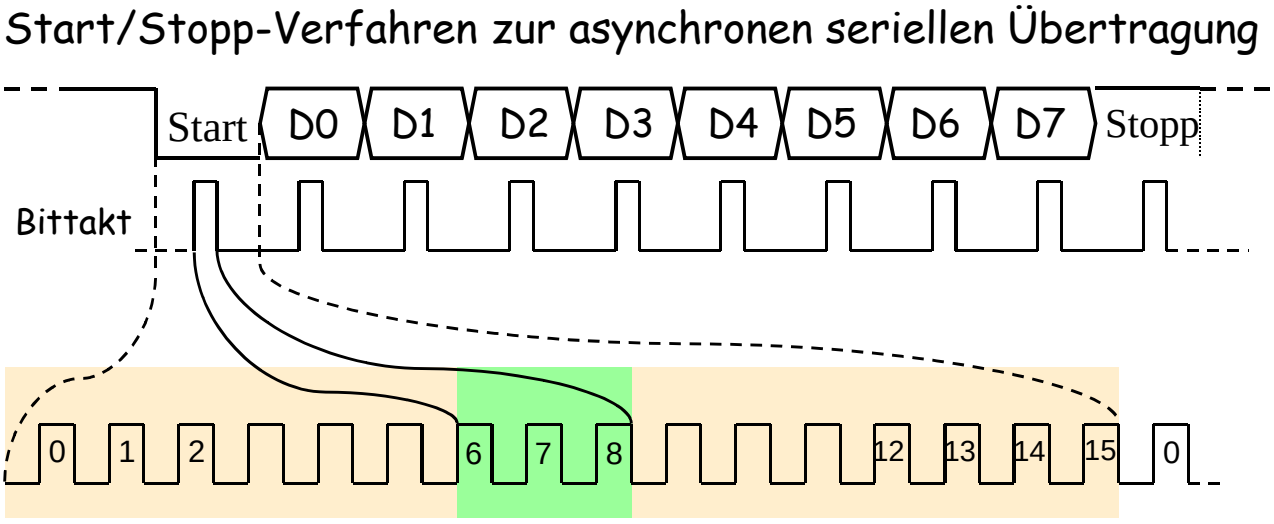



Bild 4.79: Grundlagen der Rahmensynchronisation zur asynchronen Übertragung

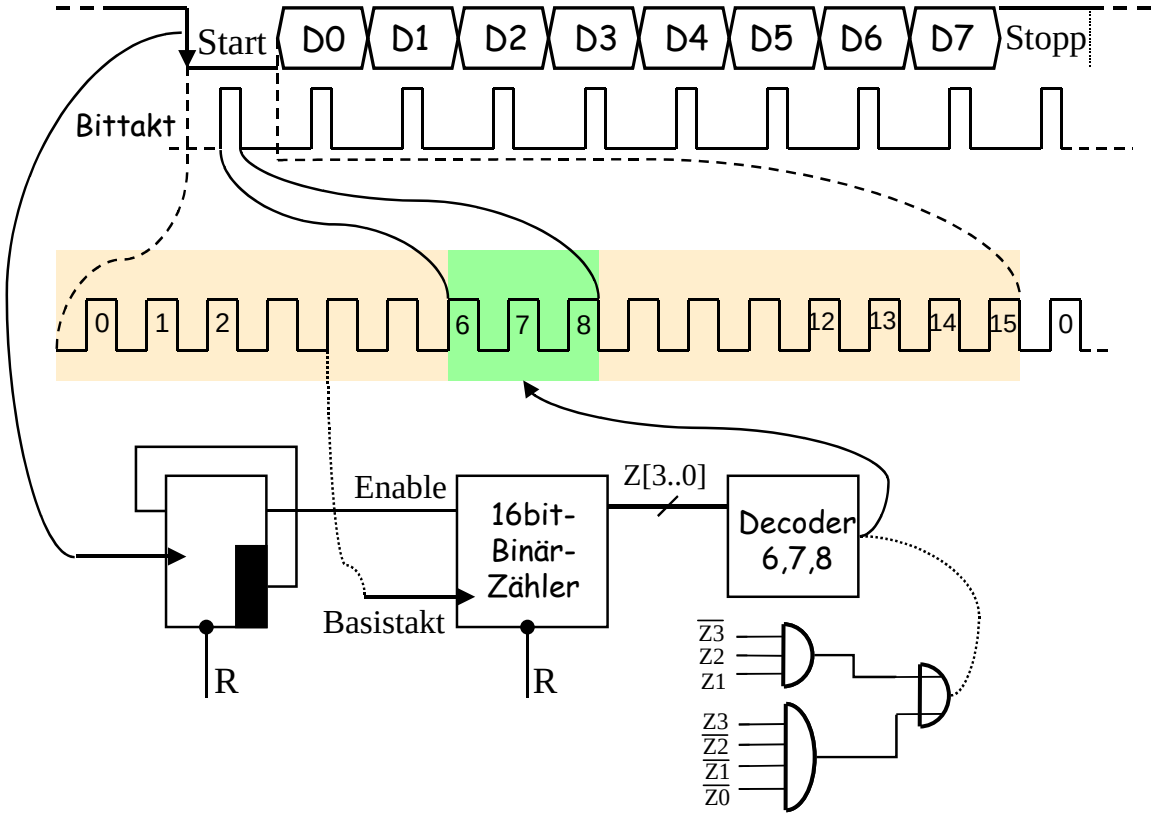



Bild 4.80: Bittakterzeugung aus einem schnellen Basistakt und „störsichere“ Bitabtastung

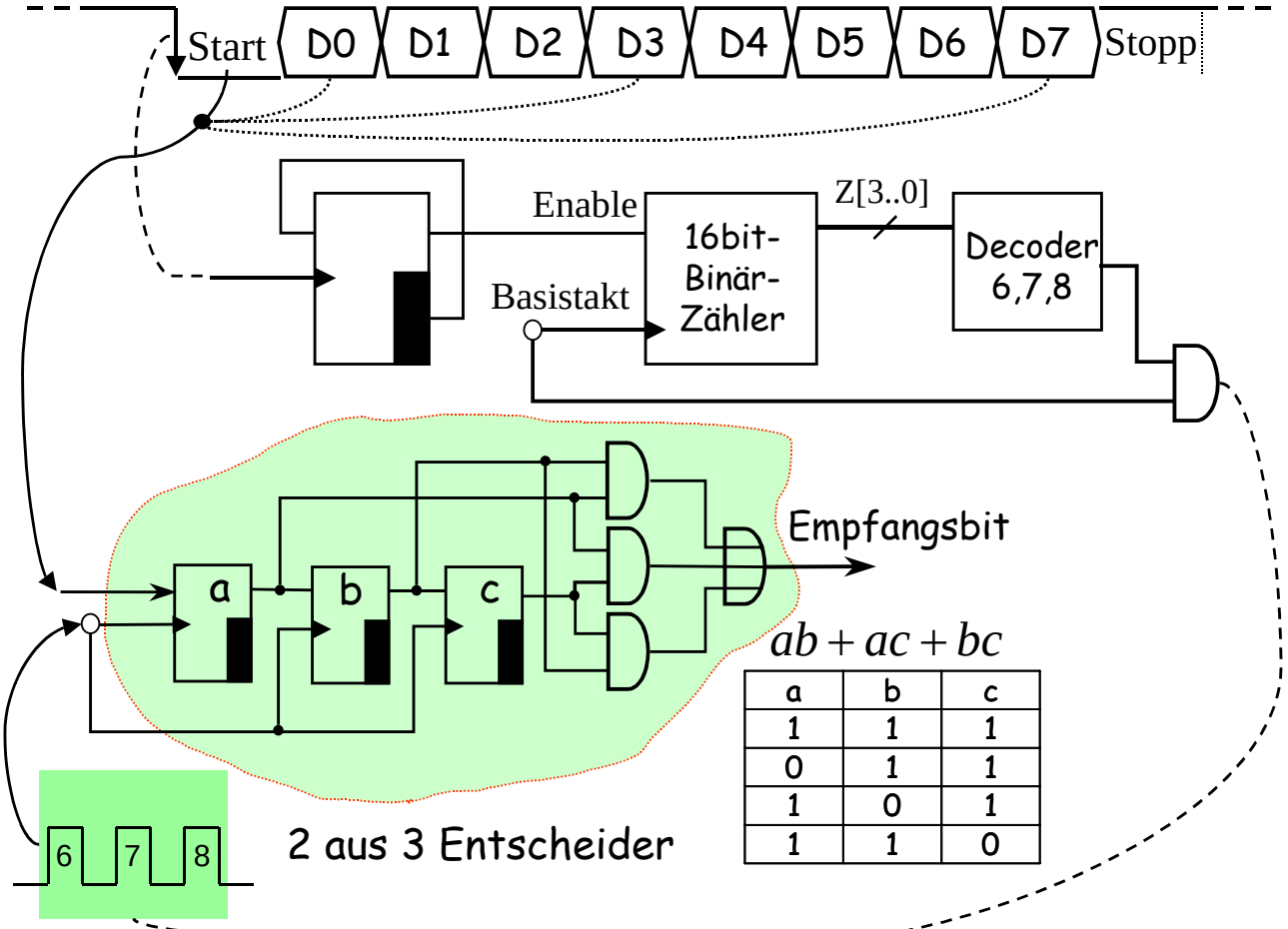



Bild 4.81: Details der Hardware zur sicheren asynchronen Bitdetektion

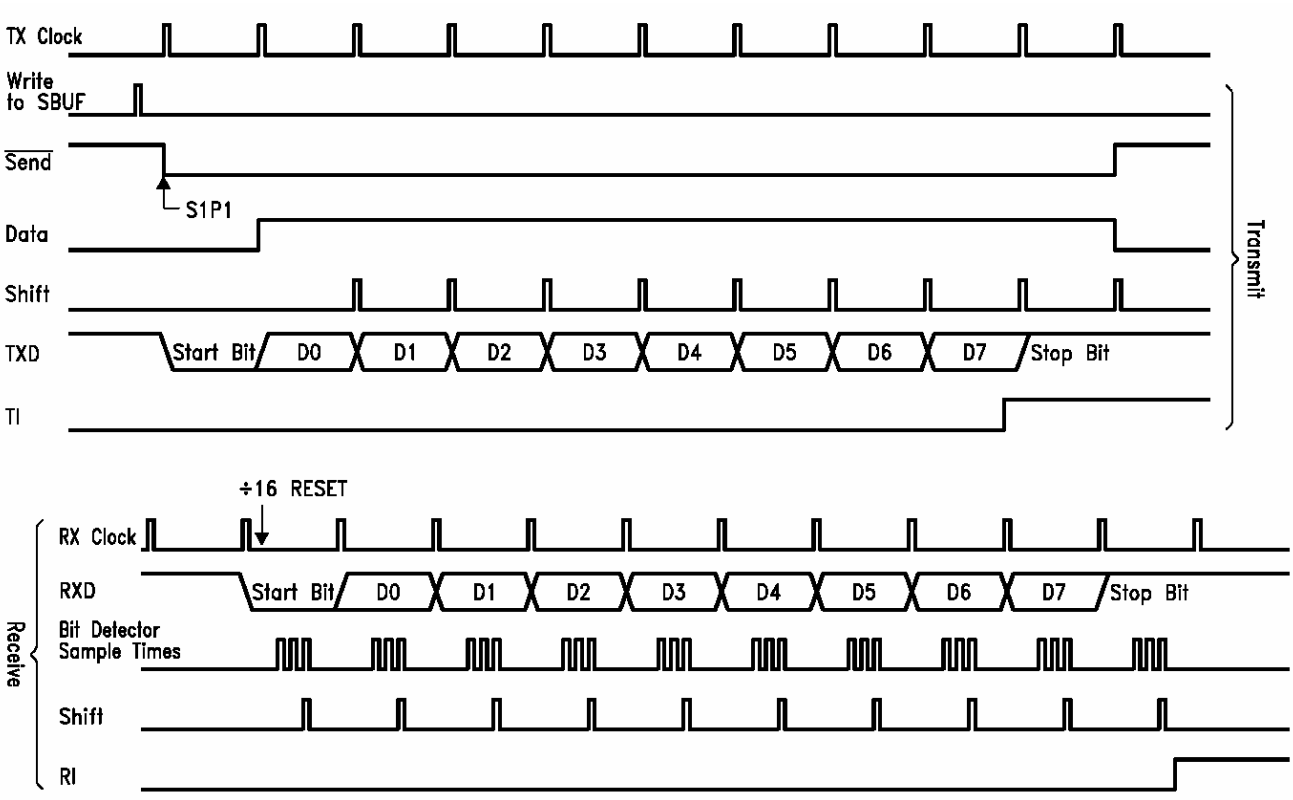



Bild 4.82: Vollständige Timing-Schemata für Senden und Empfangen

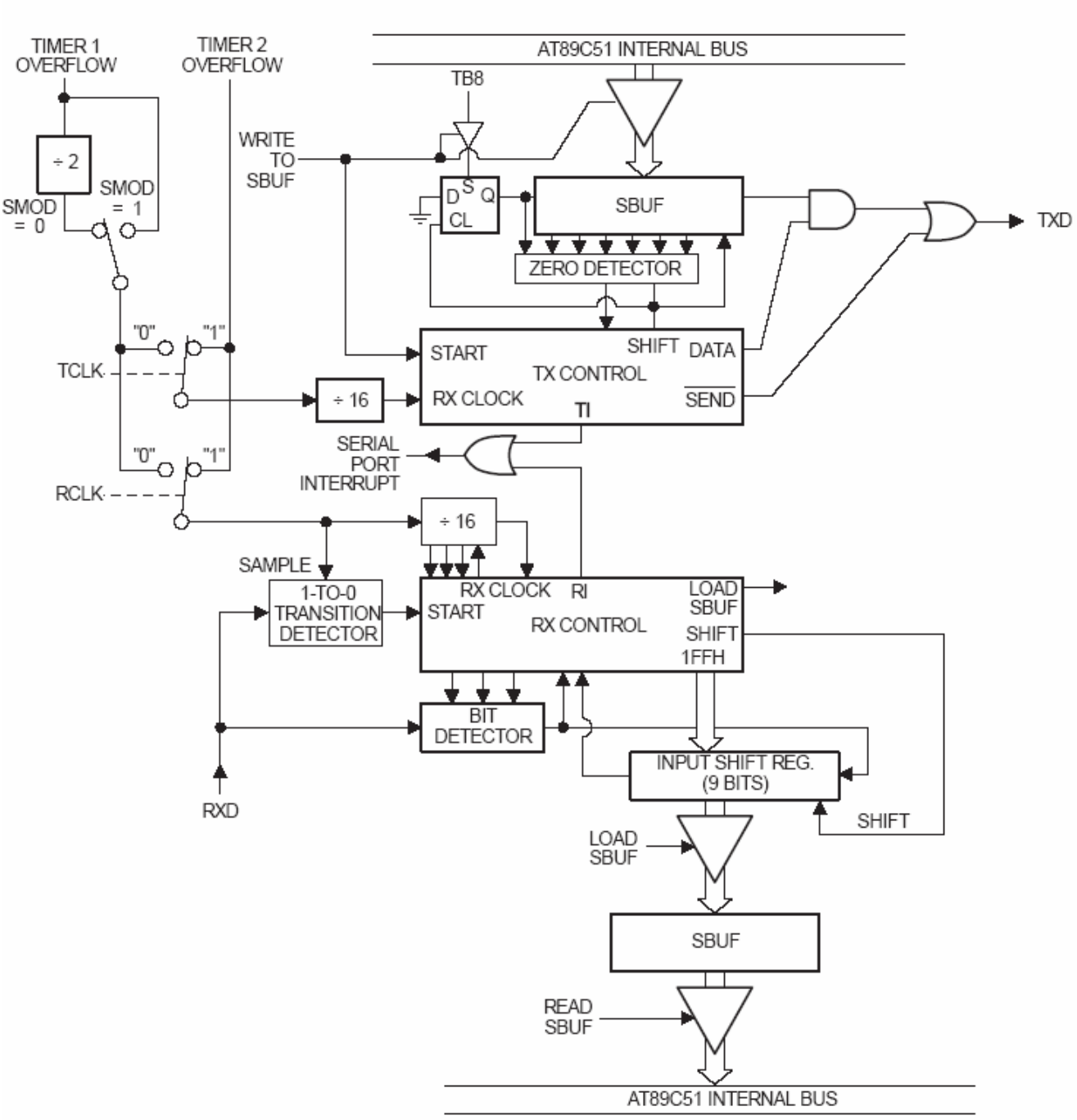



Bild 4.83: Funktionsblockschaltbild einer seriellen Schnittstelle für Vollduplexbetrieb

#### 4.7.2.1.1 Details zur seriellen Datenübertragung im 8051

- Vollduplexbetrieb mit Pufferregister (1Byte)

- SBUF (SFR) ⇒ **Schreiben** lädt das Senderegister
  ⇒ **Lesen** liest das Empfangsregister
  (gleicher Name SBUF, aber zwei separate Register)

## Es sind 4 Betriebsarten (MODES) möglich

**synchrone** Betriebsart

**MODE 0**: - Daten Ein- und Ausgabe über RxD ≡ P3.0-Pin
- TxD gibt den Bittakt aus
- Halbduplex-Betrieb ( abwechselnd Senden und Empfangen)
- feste Baudrate: 1/12 $f_{osz}$
- 8bit lange Datenblöcke können abwechselnd gesendet oder empfangen werden **1. Bit: LSB**; **9. Bit** immer ≡ **1**

**asynchrone** Betriebsart

**MODE 1**: - Senden über TxD
- Empfangen über RxD
- 10bit-Rahmen: 1 Startbit "L", 8 Datenbits (1. Bit ≡ LSB),
  1 Stoppbit "H" gelangt in **RB8** im SFR **SCON**
- Vollduplexbetrieb mit variabler Baudrate (Basis: Timer 1)

**asynchrone** Betriebsart

**MODE 2**: - Senden über TxD
- Empfangen über RxD
- 11bit-Rahmen: 1 Startbit „0“, 8 Datenbits (1. Bit ≡ LSB),
  1 programmierbares 9. Bit (TB8 in SCON)
  ⇒ gelangt beim Empfänger in RB8 im SFR SCON
  1 Stoppbit „1“
- Vollduplexbetrieb mit 2 Baudraten: $f_{osz}/32$ oder $f_{osz}/64$

Bemerkung: TB8 wird als 9. Bit gesendet; RB8 wird als 9. Bit empfangen.
(Speicherung in SCON)

**MODE 3:** - entspricht MODE 2 mit variabler Baudrate (Basis: Timer1)

| | $9F_H$ | $9E_H$ | $9D_H$ | $9C_H$ | $9B_H$ | $9A_H$ | $99_H$ | $98_H$ | |
|---|---|---|---|---|---|---|---|---|---|
| $98_H$ | SM0 | SM1 | SM20 | REN0 | TB80 | RB80 | TI0 | RI0 | S0CON |

| Bit | | Symbol | |
|---|---|---|---|
| SM0 | SM1 | | |
| 0 | 0 | Serial mode 0: | Shift register mode, fixed baud rate |
| 0 | 1 | Serial mode 1: | 8-bit UART, variable baud rate |
| 1 | 0 | Serial mode 2: | 9-bit UART, fixed baud rate |
| 1 | 1 | Serial mode 3: | 9-bit UART, variable baud rate |
| SM20 | | Enables the multiprocessor communication feature in modes 2 and 3. In mode 2 or 3 and SM20 being set to 1, RI0 will not be activated if the received 9th data bit (RB80) is 0. In mode 1 and SM20 = 1, RI0 will not be activated if a valid stop bit has not been received. In mode 0, SM20 should be 0. | |
| REN0 | | Receiver enable. Enables serial reception. Set by software to enable reception. Cleared by software to disable reception. | |
| TB80 | | Transmitter bit 8. Is the 9th data bit that will be transmitted in modes 2 and 3. Set or cleared by software as desired. | |
| RB80 | | Receiver bit 8. In modes 2 and 3 it is the 9th bit that was received. In mode 1, if SM20 = 0, RB80 is the stop bit that was received. In mode 0, RB80 is not used. | |
| TI0 | | Transmitter interrupt. Is the transmit interrupt flag. Set by hardware at the end of the 8th bit time in mode 0, or at the beginning of the stop bit in the other modes, in any serial transmission. Must by cleared by software. | |
| RI0 | | Receiver interrupt. Is the receive interrupt flag. Set by hardware at the end of the 8th bit time in mode 0, or during the stop bit time in the other modes, in any serial reception. Must be cleared by software. | |

Bild 4.84: Das Steuerregister *SCON* der seriellen Schnittstelle (englische Beschreibung)

**SCON das Steuerregister der seriellen Schnittstelle(98H)**

Bit-Nr. (00000000 nach Reset)

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| SM0 | SM1 | SM2 | REN | TB8 | RB8 | TI | RI |

| SM0 | SM1 | Betriebsart | | Baudrate |
|---|---|---|---|---|
| 0 | 0 | M0 | Schieberegister | $f_{osz}/12$ |
| 0 | 1 | M1 | 8bit UART[18] | variabel |
| 1 | 0 | M2 | 9bit UART | $f_{osz}$ /32 oder 64 |
| 1 | 1 | M3 | 9bit UART | variabel |

[18] Universal Asynchronous Receiver / Transmitter

SM2 ermöglicht die Kommunikation zwischen mehreren Mikrocontrollern in den Betriebsarten 2 und 3. Wenn SM2= "1" ist, wird RI nicht gesetzt, wenn das 9. Empfangsbit (RB8) „0" ist. In Betriebsart 1 wird bei SM2=„1" RI solange **nicht** aktiviert, bis ein gültiges Stoppbit („1") empfangen wurde.

REN kann per Software gesetzt oder rückgesetzt werden und gibt den seriellen Empfang frei. REN = „0" sperrt den Empfang.

TB8 ist das zu sendende 9. Datenbit in den Betriebsarten 2 und 3. Es ist per Software setz- und löschbar.

RB8 ist das 9. empfangen Datenbit in den Betriebsarten 2 und 3. In Betriebsart 1 ist RB8 das empfangene Stoppbit und wird bei SM2= „0" überprüft, d.h. es **muß „1" sein**, damit der empfangene Datenrahmen angenommen wird.

TI Flag für Sendeinterrupt. Es wird in den Betriebsarten 1-3 am Anfang des Stoppbits gesetzt und muß per Software gelöscht werden (in der Regel innerhalb einer entsprechenden Interrupt-Bedienungsroutine.

RI Flag für Empfangsinterrupt. Es wird in den Betriebsarten 1-3 in der Mitte des Stoppbits gesetzt und muß per Software gelöscht werden (in der Regel innerhalb einer entsprechenden Interrupt-Bedienungsroutine.
Bemerkung: Bei SM2= „1" wird RI nicht gesetzt, wenn das Stoppbit nicht korrekt als „1" erkannt wurde.

## Bemerkung zur Interruptbedienung bei der seriellen Schnittstelle

TI, RI bewirken die **gleiche** Interruptanfrage (Vektoradresse 0023H). Somit weiß die CPU am Beginn der Bedienungsroutine nicht, ob TI der RI oder beide gleichzeitig vorlagen. Die beiden Flags müssen deshalb im SFR SCON nacheinander abgefragt und nach der zugehörigen Interruptbedienung **per Software** gelöscht werden.

## Das Datenregister SBUF

**Senden:** Schreiben nach SBUF lädt ein 9-Bit Schiebregister und initialisiert den Sendevorgang. Das 9. Bit ist entweder „1" (Stoppbit) oder TB8 in den Betriebsarten 2 und 3. SBUF ist beim Senden ein Nur-Schreibe-Register.

**Empfangen:** 8bit-Eingabe-Schiebregister in Betriebsart 0. In den Betriebsarten 1...3 sind 9 bit zu verarbeiten. SBUF ist jetzt ein Nur-Lese-Register; es wird geladen, wenn RI aktiviert wird. In den Betriebsarten 1...3 wird das 9. empfangene Bit in RB8 geschrieben.

Bemerkung: RB8 und SBUF bleiben unverändert, wenn wegen SM2=„1" die Empfangsdaten ignoriert werden.

## Detaillierte Beschreibung der Betriebsart 1

**M1:** **UART** ≡ **U**niversal **A**synchronous **R**eceiver/**T**ransmitter

**Senden:** Wenn ein Schreibvorgang in SBUF stattgefunden hat, beginnt das Senden mit dem Taktschritt S1P1[19], der auf ersten 16er-Teiler-Überlauf folgt –siehe Bild 4.82. Es wird somit auf den Teiler synchronisiert.

Jetzt wird das Signal $\overline{\text{SEND}}$ = „0“, d.h. das Startbit „0“ gelangt an TxD -siehe Bild 4.83.

Nach einer Bitdauer wird DATA= „1“ und das 1. Datenbit (D0) steht am Ausgang TxD.

Wenn das letzte Datenbit (D7 ≡ MSB) an TxD steht, befindet sich das 9. Bit (Stoppbit) links davon im Schieberegister SxBUF (Bild 4.83). Jetzt spricht der Null-Detektor (Zero Detector) an und es wird die letzte Schiebeoperation ausgeführt. Danach werden $\overline{\text{SEND}}$ = „1“, DATA= „0“ und TI= „1“ gesetzt und der Sendevorgang ist beendet.

**Empfangen:** wird initialisiert durch eine Rückflanke am RxD-Pin. Wenn hier eine Rückflanke auftritt, wird der Teiler durch 16 sofort zurückgesetzt und „1FFH“ in das 9 bit lange Empfangsschieberegister geschrieben –siehe Bild 4.83. Jedes Empfangsbit wird durch den Teiler in 16 gleiche Teile eingeteilt, wobei es jeweils im 7., 8. und 9. Teilerschritt (Zählerstände 6, 7 und 8) im Bit-Detektor abgetastet wird. Zur Weiterverarbeitung gelangt derjenige Wert, der bei mindestens zwei Abtastungen ermittelt wurde. Wenn dabei allerdings das Startbit nicht als „0“ erkannt wird, wird der Empfangsablauf sofort abgebrochen und es wird erneut auf eine Rückflanke an RxD gewartet.

Wenn das Startbit links im Empfangsschieberegister angekommen ist (der erste „0“–Pegel an 9. Position), wird eine letzte Schiebeoperation ausgeführt, und danach werden SBUF und RB8 geladen sowie RI= „1“ gesetzt.

**Bemerkung:** Das Laden von SBUF und RB8 und Setzen von RI erfolgen nur dann, wenn beim letzten Schiebeimpuls

RI = „0“ war

**und**

entweder SM2= „0“ oder ein gültiges Stoppbit, d.h. RB8= „1“ empfangen wurde.

Anderenfalls sind die Empfangsdaten verloren!

In jedem Fall wird nach einer neuen Rückflanke an RxD gesucht.

[19] ein Maschinenzyklus besteht aus 12 Quarztaktphasen (6 Schritten S zu je 2 Phasen P) S1P1...S6P2

**Weitere wichtige Bemerkungen:**

Aus den obigen Regeln folgt, daß bei SM2= „1“ das Stoppbit (RB8) immer „1“ sein muß, damit ein Empfangrahmen angenommen wird. Andererseits gilt bei SM2= „0“ daß der Pegel des Stoppbits **irrelevant** ist, d.h. es besteht ein erhöhtes Risiko, fehlerhafte Datenrahmen als gültig anzunehmen!

**Zu den MODES 2 und 3:**

- im Unterschied zu MODE 1 ist das 9. Bit TB8 in SCON programmierbar
- beim Empfang wird das 9. Bit in RB8 in SCON gespeichert.
- es werden stets 11 bit übertragen: 1 Startbit, 8 Datenbits, TB8 und 1 Stoppbit

## Wie generiert man Standard-Baudraten aus einem 12MHz-Takt mit Hilfe der Timer im MC8051

Im Standard 8051 ist der Timer 1 per Hardware direkt mit der seriellen Schnittstelle gekoppelt, d.h. bei jedem Überlauf wird ein Taktimpuls an den Eingang „Baud Rate Clock“ - siehe Bild 4.83 – geliefert. Durch geeignete Programmierung der Überlaufrate kann damit die passende Baudrate eingestellt werden.

Mit dem Timer 2 (z.B. im 8052 oder ADuC 832/842) hat man zusätzliche Möglichkeiten und erhöhte Flexibilität bei der Baudratengenerierung – s. Bild 4.83 und Tabelle 4.9 sowie die davor stehende Beschreibung von T2CON, dort insbesondere die Bits RCLK und TCLK. Durch gleichzeitige Verwendung von Timer 1 und Timer 2 können verschiedene Baudraten für Sende- und Empfangsrichtung eingestellt werden - Bild 4.83.

Im folgenden sind dazu einige Beispiele unter Verwendung von Timer 1 angegeben.

**50 Baud** Timer 1 in MODE 1 setzen; Interrupt freigeben

in Interruptroutine Timer 1 laden mit 64.298 = **FB 2A** H

⇒ 1238 Takte bis Überlauf +12 Maschinenzyklen für Int.-Bedienung

⇒ eine Timerperiode dauert 1250µs

somit ist die resultierende Überlaufrate $10^6/1250s^{-1}= 800s^{-1}$

SMOD[20] = „1“ ⇒ Teiler durch 16 aktiv

Baudrate: $800s^{-1}/16=$ 50 Baud **exakt!**

Im folgenden sind die nötigen Schreibvorgänge in die entsprechenden Spezialfunktionsregister für 50 Baud angegeben.

TMOD Bit-Nr.

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | Gate | C/T | M1 | M0 |
| Timer 1 | | | | Timer 0 | | | |

```
MOV     TMOD,   #10H        ;ggf. Einstellungen für Timer 0
```

[20] Steuerbit im SFR PCON

IE (Interrupt-Enable-Register) Bit-Nr.

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| **1** | – | – | **1** | **1** | EX1 | ET0 | EX0 |

```
SETB    EA
SETB    ET1             ; Timer 1 Interruptfreigabe
SETB    ES              ; auch den Interrupt für die serielle Schnittstelle freigeben

MOV     TH1,  #0FBH     ; Timer-1-Register laden
MOV     TL1,  #2AH
```

PCON (Power Control Register SFR-Adr. 87H) Bit-Nr.

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| SMOD | SERIPD | INT0PD | ALEOFF | GF1 | GF0 | PD | IDL |

```
ORL     PCON,   #80H
```

TCON Bit-Nr.

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| TF1 | **1** | TF0 | TR0 | IE1 | IT1 | IE0 | IT0 |
| Timer1/0 | | | | externe Interrupts | | | |

```
SETB    TR1     ; Timer 1 starten
```

(TF1= „**0**“ nach Reset und wird im weiteren von der Hardware bedient)

**75 Baud** wie oben, jedoch Laden des Timers mit 64.715
⇒ 821 Takte bis Überlauf +12 Maschinenzyklen für Int.-Bedienung
resultierende Überlaufrate: 1200,48$s^{-1}$
SMOD= „1“ ⇒ Teiler durch 16 aktiv
Baudrate: 1200,48$s^{-1}$/16 = 75,03 Baud
relativer Fehler: **0,4 ‰**

**110 Baud** wie oben, jedoch Laden mit 64.980
⇒ 556 Takte bis Überlauf +12 Maschinenzyklen für Int.-Bedienung
resultierende Überlaufrate: 1760,56$s^{-1}$
SMOD= „1“ ⇒ Teiler durch 16 aktiv
Baudrate: 1760,56$s^{-1}$/16 = 110,035 Baud
relativer Fehler: **0,32 ‰**

**150 Baud** Timer 1 in MODE 2 ≡ 8 bit-Auto-Reload
Interrupt sperren!
Reload-Wert 48 ⇒ 256-48 ⇒ 208 Takte bis Überlauf
Überlaufrate: 4807,69$s^{-1}$

SMOD= „0“ ⇒ Teiler durch 32 aktiv

Baudrate: 4807,69$s^{-1}$/32 = 150,24 Baud

relativer Fehler: **1,6 ‰**

TMOD Bit-Nr.

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| **0** | **0** | **1** | **0** | Gate | C/T | M1 | M0 |
| Timer 1 | | | | Timer 0 | | | |

```
MOV     TMOD,   #20H    ;ggf. Einstellungen für Timer 0
CLR     ET1             ; Interrupts von Timer 1 sperren
ANL     PCON,   #7FH    ;UND-Verknüpfung mit '01111111', d.h. SMOD= „0“
MOV     TL1,    #30H    ; Timer-1-Register laden
MOV     TH1,    #30H    ; Timer-1-Reloadwert speichern
SETB    TR1             ; Timer 1 starten
```

**300 Baud** wie 150 Baud, jedoch SMOD = „1“

Baudrate: 4807,69$s^{-1}$/16 = 300,48Baud

relativer Fehler: **1,6 ‰**

**600 Baud** wie 300 Baud, jedoch Reload-Wert 152

⇒ 104 Takte bis Überlauf

Überlaufrate: 9615,38$s^{-1}$

SMOD= „1“ ⇒ Teiler durch 16 aktiv

Baudrate: 9615,38$s^{-1}$/16 = 600,96 Baud

relativer Fehler: **1,6 ‰**

**1200 Baud** wie 600 Baud, jedoch Reload-Wert 204

⇒ 52 Takte bis Überlauf

Überlaufrate: 19.230,76$s^{-1}$

SMOD= „1“ ⇒ Teiler durch 16 aktiv

Baudrate: 19.230,76$s^{-1}$/16 = 1201,92 Baud

relativer Fehler: **1,6 ‰**

**2400 Baud** wie 1200 Baud, jedoch Reload-Wert 230

⇒ 26 Takte bis Überlauf

Überlaufrate: 38.461,538$s^{-1}$

SMOD= „1“ ⇒ Teiler durch 16 aktiv

Baudrate: 38.461,538$s^{-1}$/16 = 2.493,84 Baud

relativer Fehler: **1,6 ‰**

**4800 Baud** wie 2400 Baud, jedoch Reload-Wert 243

⇒ 13 Takte bis Überlauf

Überlaufrate: 76.923$s^{-1}$

SMOD= „1“ ⇒ Teiler durch 16 aktiv

Baudrate: 76.923$s^{-1}$/16 = 4.807,69 Baud

relativer Fehler: **1,6 $^0/_{00}$**

## Versuch für 9600 Baud:

**9600 Baud** wie 4800 Baud, jedoch Reload-Wert 250

1. Versuch ⇒ 6 Takte bis Überlauf

Überlaufrate: 166.666,66$s^{-1}$

SMOD= „1“ ⇒ Teiler durch 16 aktiv

Baudrate: 166.666,66$s^{-1}$/16 = 10.416,66 Baud

**relativer Fehler: +8,5%**

2. Versuch ⇒ 7 Takte bis Überlauf

Überlaufrate: 142.857,14$s^{-1}$

SMOD= „1“ ⇒ Teiler durch 16 aktiv

Baudrate: 142.857,14$s^{-1}$/16 = 8.928,57 Baud

**relativer Fehler: -7%**

### Fehlerbetrachtungen:

Wegen der Abtastung ungefähr in Bitmitte muß der maximale Zeitfehler stets kleiner als eine **halbe Bitdauer** sein.

Da der Fehler über einen Datenrahmen (10bit) akkumuliert, wäre das 10. Bit (Stoppbit) bei einem relativen Fehler von 7% (wie oben für Versuch 2 bei 9600 Baud ermittelt) um 70% zeitversetzt und somit auf keinen Fall mehr korrekt detektierbar. Lediglich die ersten 6 bit, inkl. Startbit, könnten richtig erkannt werden.

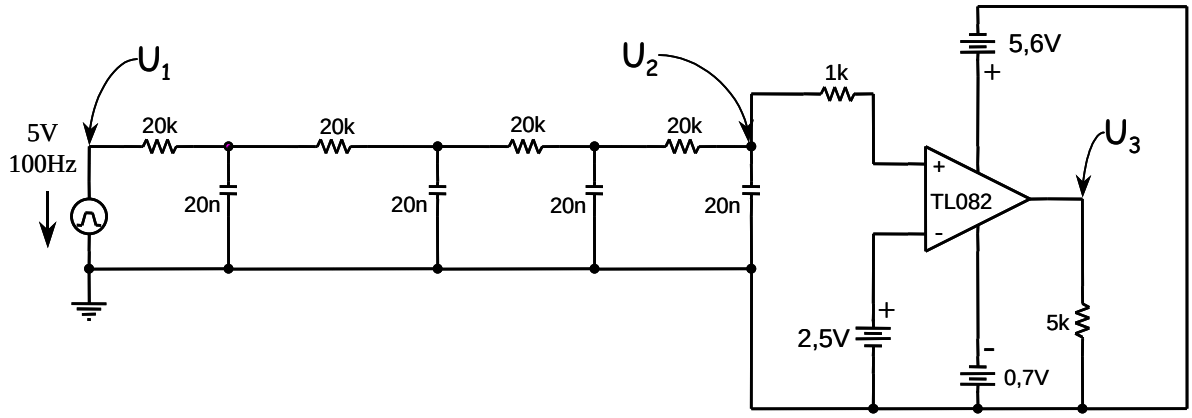
Für die übrigen oben berechneten Fälle (110-4800 Baud) bleibt der relative Fehler durch die Numerik des Timers bei maximal 1,6 $^0/_{00}$, d.h. bis zum 10. Bit kann er auf maximal 1,6% akkumulieren, was absolut unkritisch ist.

**Neben der Timernumerik müssen noch zwei weitere Fehlerursachen in Betracht gezogen werden:**

- **Genauigkeit der Frequenzerzeugung** in Sender und Empfänger. Da Sender und Empfänger außer der Datenleitung keine Verbindung haben muß die Stabilität der lokalen Takterzeugung ausreichend hoch sein. Unter der Annahme, daß auf beiden Seiten handelsübliche Schwingquarze zum Einsatz kommen, darf von einer Toleranz in der Größenordnung von $\pm$ 100pm $\hat{=}$ $10^{-4}$ oder 0,1 $^0/_{00}$ ausgegangen werden. Selbst wenn die Toleranzen entgegengesetzt sind und somit von 0,2 $^0/_{00}$ relativem Fehler ausgegangen werden muß, führt die Akkumulation bei 10 bit langen Datenrahmen lediglich auf 2 $^0/_{00}$ was in jedem Fall vernachlässigbar ist. Bei Verwendung billigerer Taktquellen - wie z.B. keramischen Resonatoren - muß im Einzelfall überprüft werden, ob die Genauigkeit genügt.

- **„Schrittverzerrungen“** durch die Eigenschaften des Übertragungskanals. Am einfachsten ist dies bei Leitungen zu übersehen. Sie können in erster Näherung durch eine Kettenschaltung von RC-Tiefpässen ersetzt werden – siehe Bild 4.85. Mit wachsender Leitungslänge werden aus den sauberen rechteckförmigen Datenbits „verschliffene“ Zeitverläufe, die sich mehr und mehr einer Sinusform annähern, wobei gleichzeitig die Amplitude absinkt. Wenn die Entscheidungsschwelle „0“ – „1“, bzw. „1“ – „0“ im Empfänger **nicht passend optimiert wird**, kommt es zu den „Schrittverzerrungen“, die letztlich darin bestehen, daß „0“- und „1“-Bits verschiedene Dauern annehmen, die stark vom Taktraster abweichen können.

Bild 4.85: Schaltung zur PSPICE-Simulation von Schrittverzerrungen bei der seriellen Datenübertragung



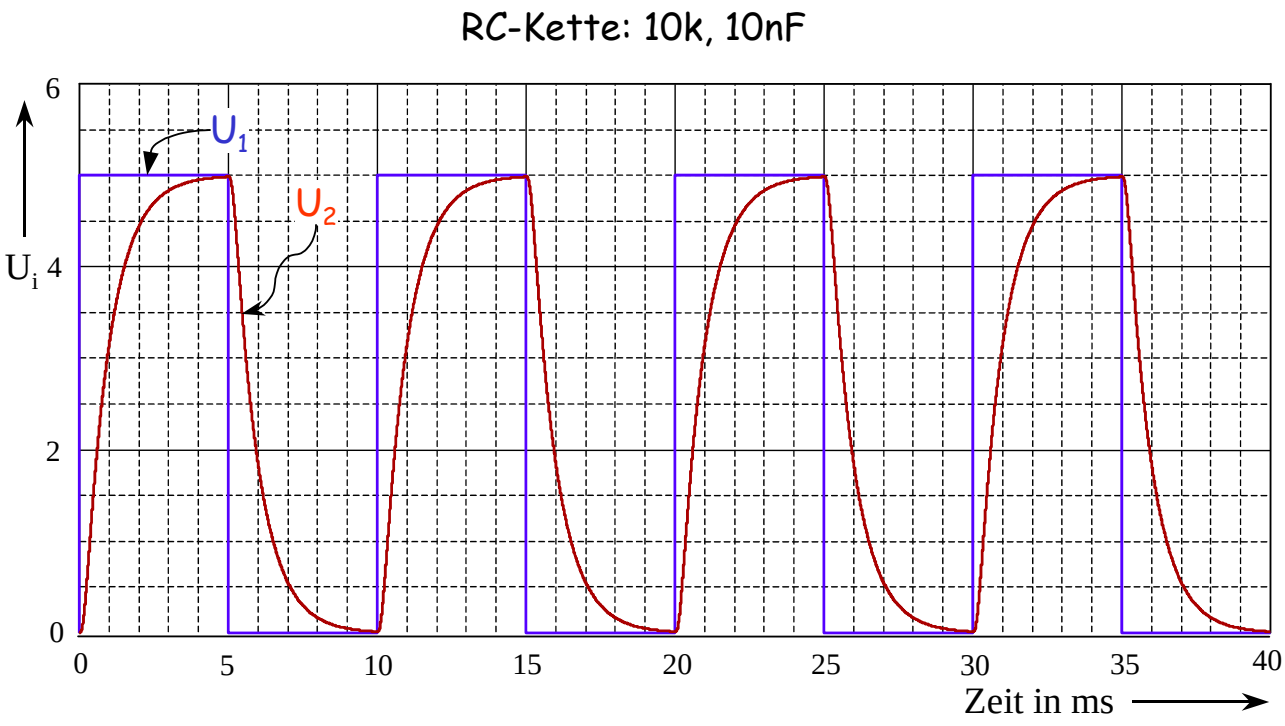




Bild 4.86: Simulation der Signalverzerrung an einer kurzen Leitung

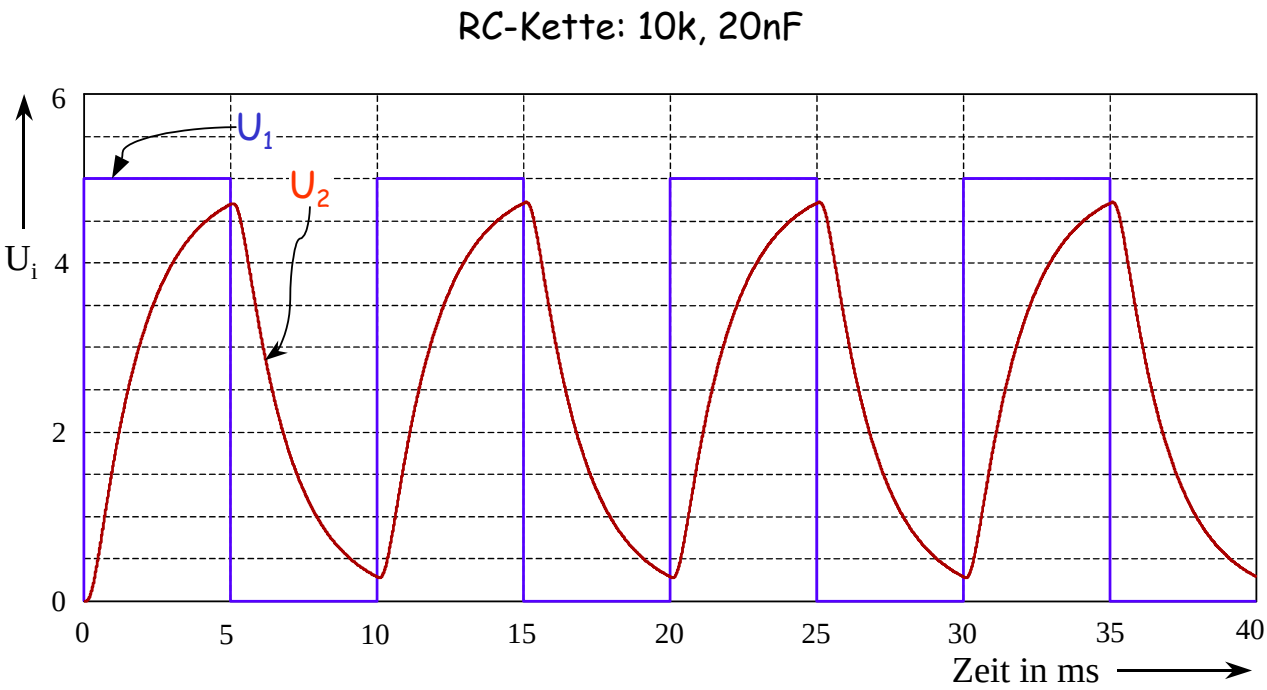



Bild 4.87: Simulation der Signalverzerrung an einer mittellangen Leitung

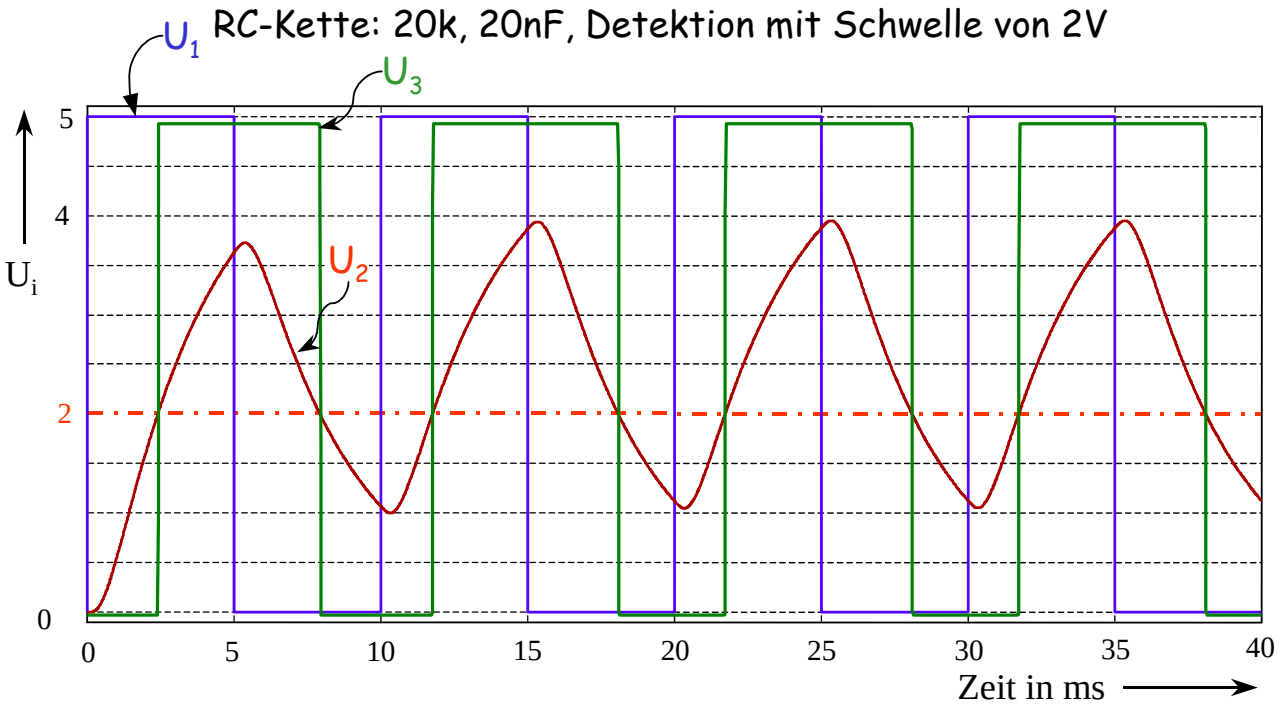



Bild 4.88: Simulation einer sehr langen Leitung mit empfängerseitiger Signaldetektion bei einer Schwelle von 2V

Die schon erheblichen Schrittverzerrungen in Bild 4.88 und Bild 4.89 können durch optimal Wahl der Detektionsschwelle deutlich reduziert werden wie das folgende Bild 4.90 zeigt. Obwohl hier sogar noch eine größere Leitungslänge angesetzt ist, gelingt eine nahezu ideale Signalrückgewinnung.

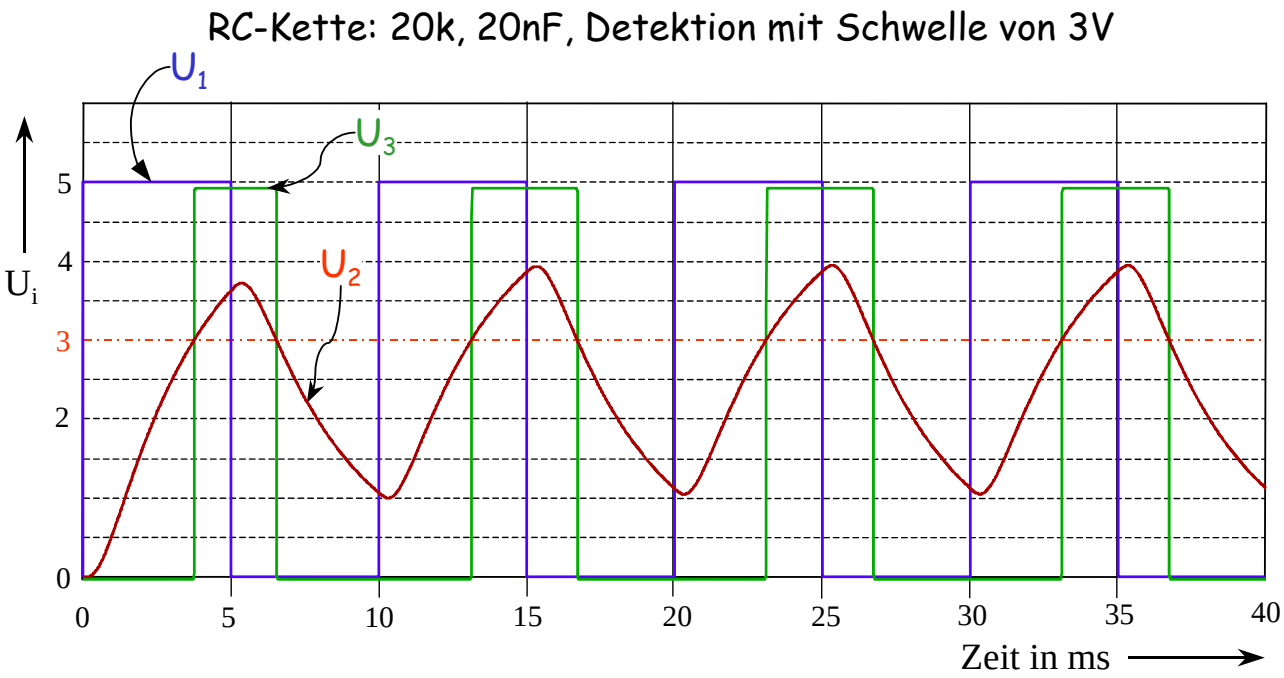



Bild 4.89: Simulation einer sehr langen Leitung mit empfängerseitiger Signaldetektion bei einer Schwelle von 3V

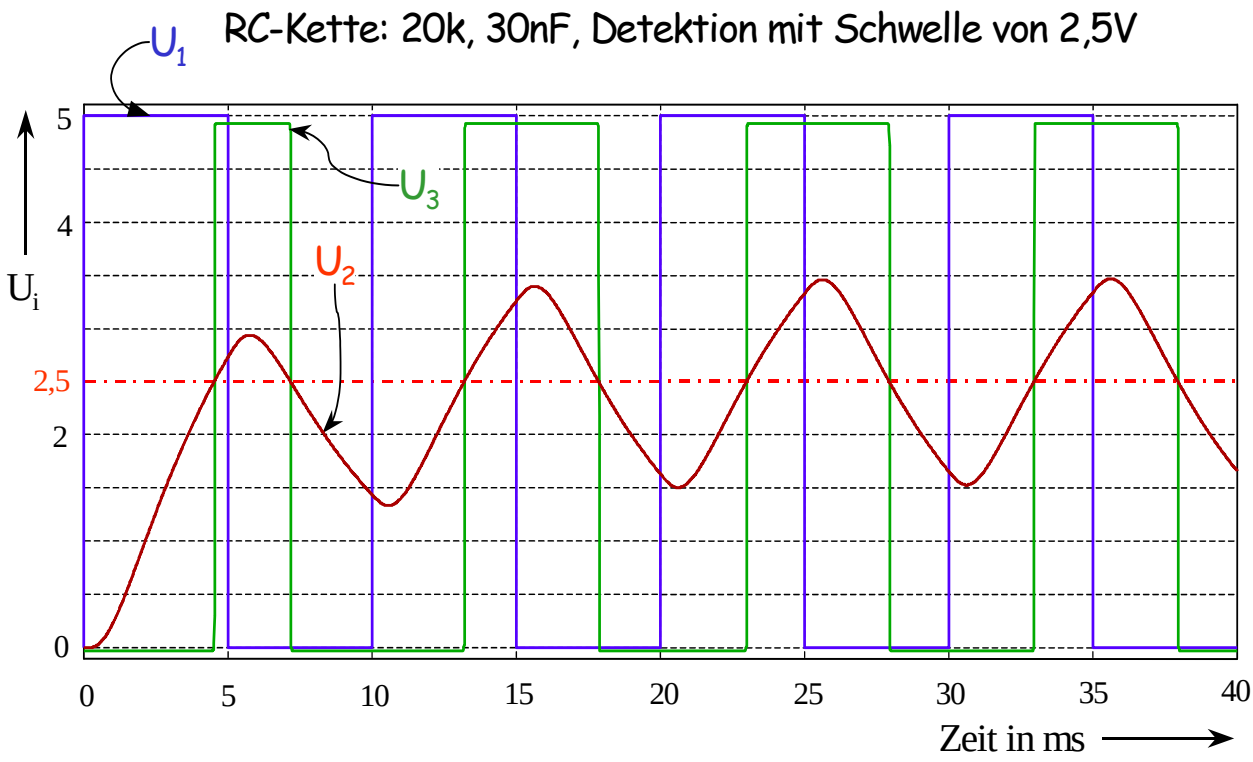



Bild 4.90: Simulation einer sehr langen Leitung mit empfängerseitiger Signaldetektion bei einer optimalen Schwelle von 2,5V

### 4.7.2.2 Bedienung integrierter A/D-Wandler am Beispiel des ADuC832/842

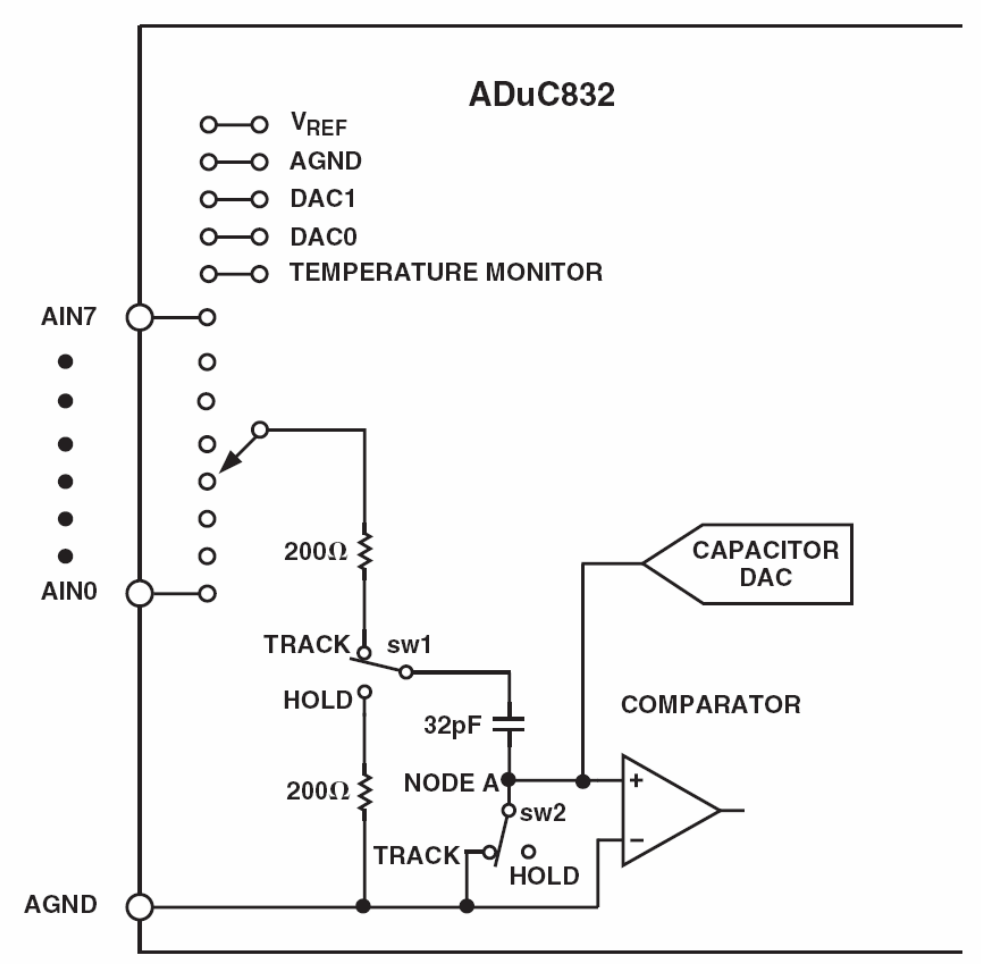
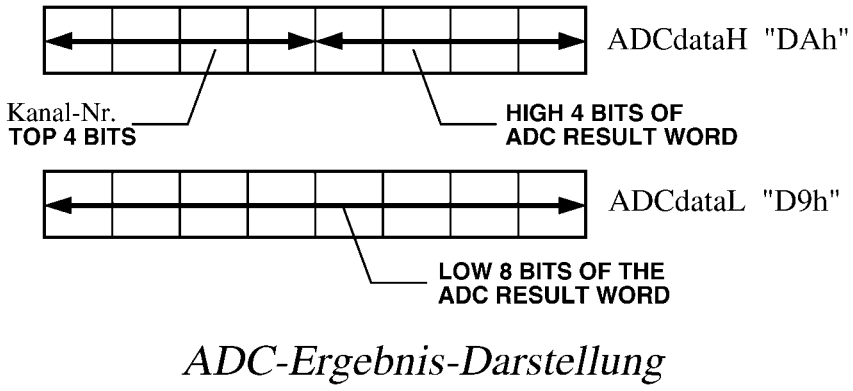



Bild 4.91: Vereinfachtes Funktionsschaltbild für den A/D-Wandler im ADuC832/842




*ADC-Ergebnis-Darstellung*

Bild 4.92: Belegung der beiden Spezialfunktionsregister ADCdataH und ADCdataL für das Wandelergebnis

Der Eingang des A/D-Wandlers im Mikroconverter ADuC832 kann mit einem von 8 Pins verbunden werden. Für die präzise Funktion des Wandlers müssen die jeweils an den Pins angeschlossenen Signalquellen hinreichend niederohmig sein. In der Regel verwendet man daher einen Operationsverstärker als Buffer. Für die Bedienung des Wandlers sind 5 neue SFR implementiert:

- ADCdataL Low-Byte des Wandelergebnisses
- ADCdataH obere 4bit des Wandelergebnisses + Kanalnummer (4bit)
- ADCCON1 Steuerregister 1
- ADCCON2 Steuerregister 2
- ADCCON3 Steuerregister 3

| Bit | Name | Description |
|---|---|---|
| ADCCON1.7 | MD1 | The Mode bit selects the active operating mode of the ADC.<br>Set by the user to power up the ADC.<br>Cleared by the user to power down the ADC. |
| ADCCON1.6 | EXT_REF | Set by the user to select an external reference.<br>Cleared by the user to use the internal reference. |
| ADCCON1.5<br>ADCCON1.4 | CK1<br>CK0 | The ADC clock divide bits (CK1, CK0) select the divide ratio for the PLL master clock used to generate the ADC clock. To ensure correct ADC operation, the divider ratio must be chosen to reduce the ADC clock to 4.5 MHz and below. A typical ADC conversion will require 17 ADC clocks.<br>The divider ratio is selected as follows:<br>**CK1 CK0 MCLK Divider**<br>0 0 8<br>0 1 4<br>1 0 16<br>1 1 32 |
| ADCCON1.3<br>ADCCON1.2 | AQ1<br>AQ0 | The ADC acquisition select bits (AQ1, AQ0) select the time provided for the input track-and-hold amplifier to acquire the input signal. An acquisition of three or more ADC clocks is recommended; clocks are selected as follows:<br>**AQ1 AQ0 #ADC Clks**<br>0 0 1<br>0 1 2<br>1 0 3<br>1 1 4 |
| ADCCON1.1 | T2C | The Timer 2 conversion bit (T2C) is set by the user to enable the Timer 2 overflow bit be used as the ADC convert start trigger input. |
| ADCCON1.0 | EXC | The external trigger enable bit (EXC) is set by the user to allow the external Pin P3.5 (CONVST) to be used as the active low convert start input. This input should be an active low pulse (minimum pulsewidth >100 ns) at the required sample rate. |

Tabelle 4.10:Bedeutung und Funktion der Bits im Spezialfunktionsregister ADCCON1
Adresse „EFh“, nicht bitadressierbar, “00“ nach Reset

Die Funktionen der Steuerregister sind aus den Tabellen 4.10-4.12 ersichtlich.

| Bit | Name | Description |
|---|---|---|
| ADCCON2.7 | ADCI | The ADC interrupt bit (ADCI) is set by hardware at the end of a single ADC conversion cycle or at the end of a DMA block conversion. ADCI is cleared by hardware when the PC vectors to the ADC Interrupt Service Routine. Otherwise, the ADCI bit should be cleared by user code. |
| ADCCON2.6 | DMA | The DMA mode enable bit (DMA) is set by the user to enable a preconfigured ADC DMA mode operation. A more detailed description of this mode is given in the ADC DMA Mode section. The DMA bit is automatically set to "0" at the end of a DMA cycle. Setting this bit causes the ALE output to cease, it will start again when DMA is started and will operate correctly after DMA is complete. |
| ADCCON2.5 | CCONV | The continuous conversion bit (CCONV) is set by the user to initiate the ADC into a continuous mode of conversion. In this mode, the ADC starts converting based on the timing and channel configuration already set up in the ADCCON SFRs; the ADC automatically starts another conversion once a previous conversion has completed. |
| ADCCON2.4 | SCONV | The single conversion bit (SCONV) is set to initiate a single conversion cycle. The SCONV bit is automatically reset to "0" on completion of the single conversion cycle. |
| ADCCON2.3<br>ADCCON2.2<br>ADCCON2.1<br>ADCCON2.0 | CS3<br>CS2<br>CS1<br>CS0 | The channel selection bits (CS3–0) allow the user to program the ADC channel selection under software control. When a conversion is initiated, the channel converted will be that pointed to by these channel selection bits. In DMA mode, the channel selection is derived from the channel ID written to the external memory.<br>**CS3 CS2 CS1 CS0 CH#**<br>0 0 0 0 0<br>0 0 0 1 1<br>0 0 1 0 2<br>0 0 1 1 3<br>0 1 0 0 4<br>0 1 0 1 5<br>0 1 1 0 6<br>0 1 1 1 7<br>1 0 0 0 Temp Monitor — Requires minimum of 1 µs to acquire<br>1 0 0 1 DAC0 — Only use with Internal DAC o/p buffer on<br>1 0 1 0 DAC1 — Only use with Internal DAC o/p buffer on<br>1 0 1 1 AGND<br>1 1 0 0 VREF<br>1 1 1 1 DMA STOP — Place in XRAM location to finish DMA sequence, see the section ADC DMA Mode.<br>All other combinations reserved |

Tabelle 4.11:Bedeutung und Funktion der Bits im Spezialfunktionsregister ADCCON2
Adresse „D8h“, bitadressierbar, “00“ nach Reset

| Bit | Name | Description |
|---|---|---|
| ADCCON3.7 | BUSY | The ADC Busy Status Bit (BUSY) is a read-only status bit that is set during a valid ADC conversion or calibration cycle. Busy is automatically cleared by the core at the end of conversion or calibration. |
| ADCCON3.6 | GNCLD | Gain Calibration Disable Bit.<br>Set to "0" to Enable Gain Calibration.<br>Set to "1" to Disable Gain Calibration. |
| ADCCON3.5<br>ADCCON3.4 | AVGS1<br>AVGS0 | Number of Averages Selection Bits.<br>This bit selects the number of ADC readings averaged during a calibration cycle.<br>AVGS1 AVGS0 Number of Averages<br>0 0 15<br>0 1 1<br>1 0 31<br>1 1 63 |
| ADCCON3.3 | RSVD | Reserved. This bit should always be written as "0." |
| ADCCON3.2 | RSVD | This bit should always be written as "1" by the user when performing calibration. |
| ADCCON3.1 | TYPICAL | Calibration Type Select Bit.<br>This bit selects between Offset (zero-scale) and Gain (full-scale) calibration.<br>Set to "0" for Offset Calibration.<br>Set to "1" for Gain Calibration. |
| ADCCON3.0 | SCAL | Start Calibration Cycle Bit.<br>When set, this bit starts the selected calibration cycle. It is automatically cleared when the calibration cycle is completed. |

Tabelle 4.12:Bedeutung und Funktion der Bits im Spezialfunktionsregister ADCCON3
Adresse „F5h“, nicht bitadressierbar, “00“ nach Reset

#### 4.7.2.2.1 Komplexes Programmierbeispiel

### Aufbereitung von Daten aus dem A/D-Wandler eines ADuC 832/842 und Übertragung über die serielle Schnittstelle zum PC

Aufgabenstellung:

Ein Mikrocontroller vom Typ ADuC842 wird mit einer Kerntaktfrequenz $f_c=2^{24}$Hz=16,7772 MHz betrieben. Die maximale Geschwindigkeit, mit der ein Befehl komplett abgearbeitet werden kann, beträgt $f_c$, d.h. ein Maschinenzyklus dauert ca. 59,6ns.

Es soll ein Programm zur Verarbeitung von analog/digital-gewandelten Eingangsspannungswerten erstellt werden, mit dem die Eingangspannungswerte in der Einheit Millivolt auf die Hyperterminalkonsole eines PC übertragen werden.

Die zu erfassende Eingangsspannung wird mit dem integrierten 12bit-A/D-Wandler des ADuC842 digitalisiert und das Ergebnis steht in zwei Registern ADCDATAH (obere 4bit) und ADCDATAL (untere 8bit) zur Verfügung. Die Skalierung erfolgt so, daß der ADW bei einer Eingangsspannung von 5V den maximal mit 12bit darstellbaren Binärwert 1111 1111 1111 = FFFh = 4095d liefert.

Die Arbeitsregister R5 und R4 werden zur Aufnahme des rechtsgeschobenen (halbierten) Wandelwertes verwendet.

Nach Initialisierung des ADuC842 (Start aus RESET) wird die serielle Schnittstelle für eine Baudrate von 19.200 vorbreitet. Dann wird mittels eines „Langzeittimers“ ein 3s-Intervall eingestellt, in dem die Datenübertragung zum PC erfolgt.

Danach wird der A/D-Wandler voreingestellt und auf „softwaregetriggerte“ Einzelwandlung vorbereitet.

Jetzt folgt ein Programmabschnitt, der jeden binären Wandelwert auf 5000mV=5V kalibriert. Dazu wird das 12-bit Wandelergebnis zunächst in zwei Arbeitsregister transferiert:

ADCDATAL => R6

ADCDATAH => R7

Das Wandelergebnis in Form eines 12bit-Binärwertes ist nun durch 4096 ($2^{12}$) zu dividieren und das Divisionsergebnis mit 5000 zu multiplizieren!

Hierbei wird ausgenutzt, daß Multiplikation und Division mit Zweierpotenzen sehr einfach durch Links- bzw. Rechtsschieben bewerkstelligt werden können. Eine Multiplikation mit 2,5 ergibt sich so z.B. durch Addition des einmal links und einmal rechts geschobenen Ursprungswertes. Das bringt hier einen erheblichen Rechenvorteil aufgrund der Tatsache, daß die Zahl 5000 als Produkt $(2\cdot 2{,}5)\cdot(4\cdot 2{,}5)^3$ darstellbar ist.

Die Arbeitsregister R5 und R4 sollen zur Aufnahme der halbierten (rechtsgeschobenen Zwischenwerte benutzt werden. R7 und R6 können zunächst mit den verdoppelten Zwischenwerten überschrieben werden und schließlich mit der Summe.

Eine weitere Vereinfachung der obigen Berechung ergibt sich daraus, daß aus der Zahl 5000 die Zweierpotenz $2\cdot 4^3=128$ extrahiert und mit der Division durch 4096 verrechnet werden kann, so daß schließlich nur noch

$$(\text{ADCDATAH,L})\cdot(2{,}5^4/32)$$

auszurechnen ist. Im Programm kann diese Berechnung vorteilhaft durchgeführt werden, indem in vier Schritten mit 2,5/2 multipliziert und das Ergebnis zum Schluß noch einmal rechts geschoben wird. Das nun auf 5000mV kalibrierte Ergebnis liegt noch im Binärformat den Registern R7, R6 mit eine m Maximalwert von 1388H vor. Zur weiteren Aufbereitung für das Senden über die seriel-

le Schnittstelle muß ein Wandlung in das BCD-Format erfolgen, wobei das Ergebnis wie folgt in die Register R5 und R4 geschrieben werden soll:

| R5 | | R4 | |
|---|---|---|---|
| Tausender | Hunderter | Zehner | Einer |

Zur Ausführung dieser Aufgabe wird zunächst das niederwertige Byte aus R6 wird mit Hilfe des `DIV AB`-Befehls in Hunderter, Zehner und Einer zerlegt. Dann wird nach Maßgabe der höherwertigen Bits (4...0) in R7 der zugehörige Dezimalwert 4096 (Bit 4), 2048 (Bit 3), 1024 (Bit 2), 512 (Bit 1), 256 (Bit 0) addiert, wobei jeweils eine Korrektur mit `DA A` erfolgen muß, so daß schließlich die gültigen BCD Ziffern in den 4 Halbbytes von R5 und R4 stehen.

Der letzte Aufbereitungsschritt besteht in der Erzeugung von ASCII-Codes, die vom „Hyperterminal des PC „verstanden“ werden. Die Ablage in den Registern des ADuC842 soll wie folgt sein:

| R7 | R6 | R5 | R4 |
|---|---|---|---|
| Tausender | Hunderter | Zehner | Einer |

Die Ziffern 0...9 werden im ASCII-Code durch Voranstellen von "3" im oberen Halbbyte repräsentiert.

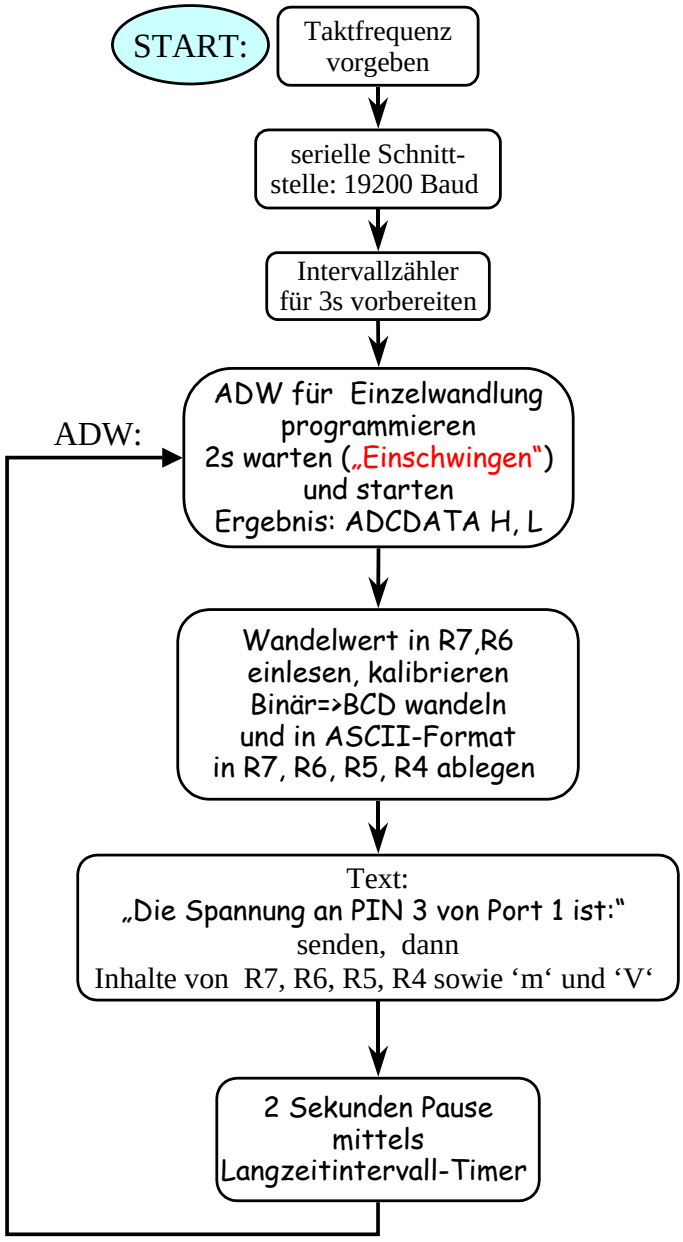



Bild 4.93: Flussdiagramm für das Gesamtprogramm

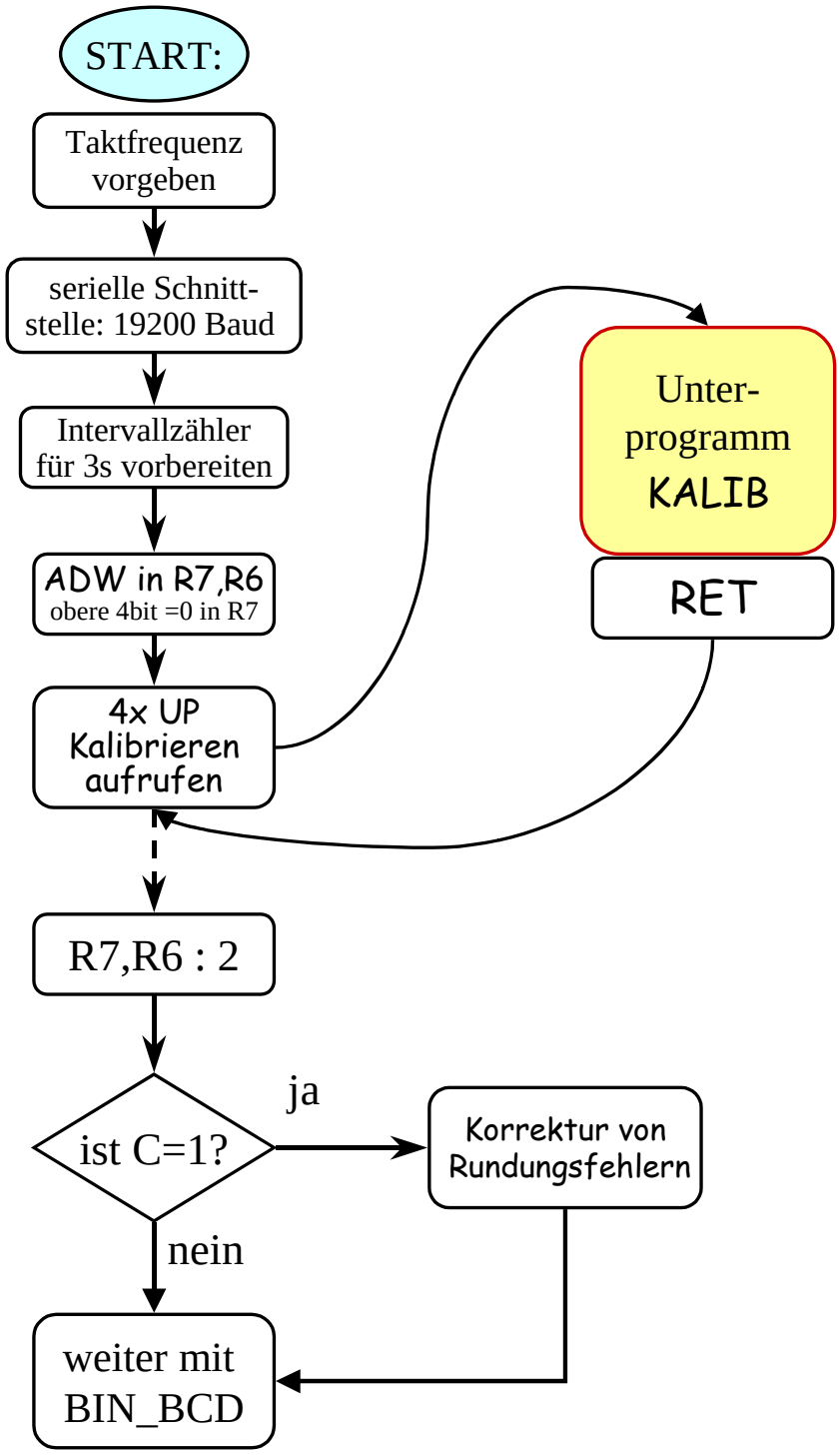
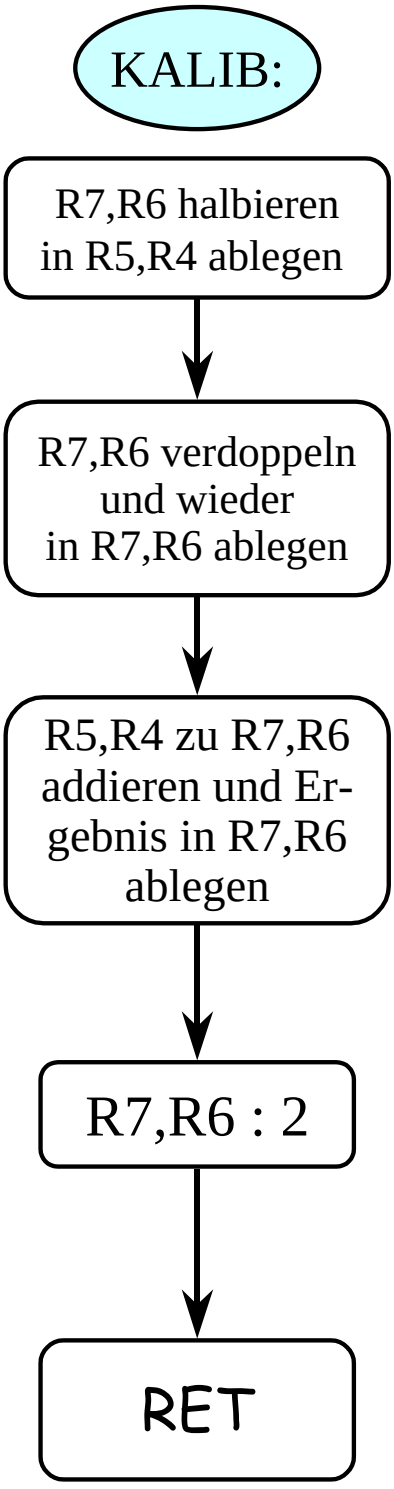



Bild 4.94: Kalibierablauf: (ADCDATAH,L)×5000/4096Bild 4.95:




U
nterprogramm
„Kalibrieren“
(R7,R6)·2,5/2

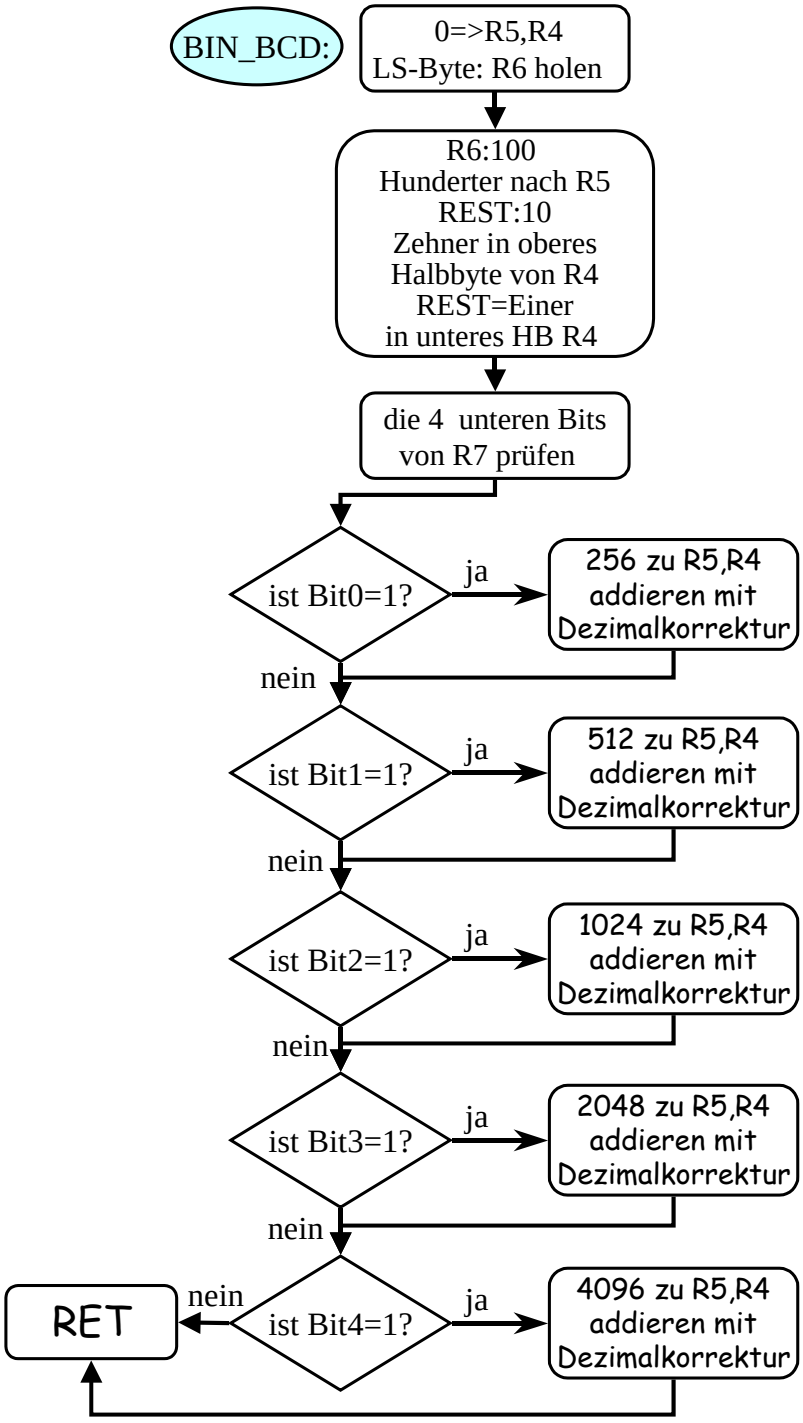



Bild 4.96: Binär-BCD-Umwandlung

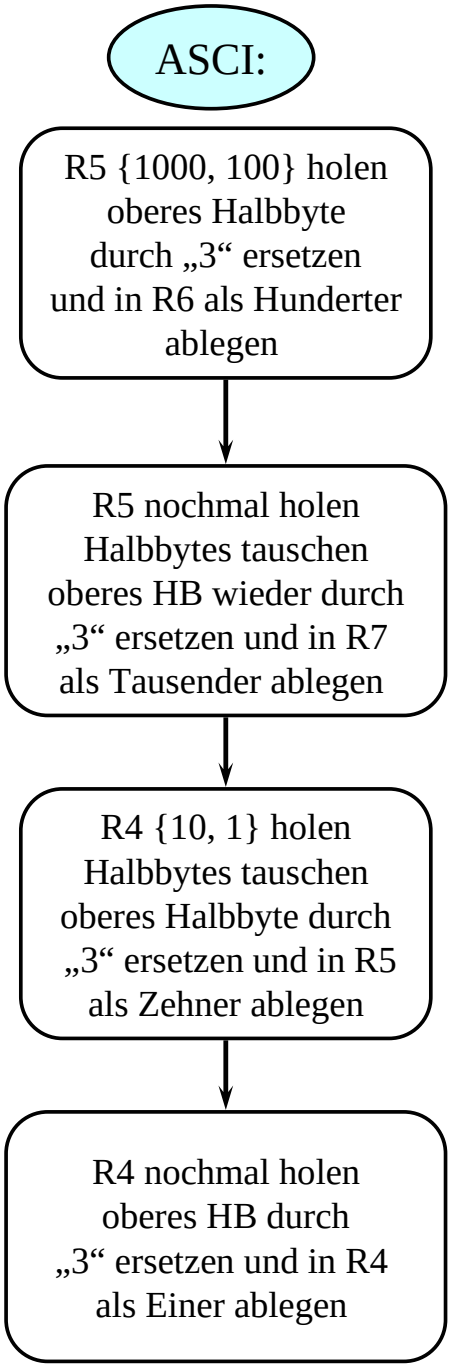



Bild 4.97: BCD zu ASCII

```
$Title (12bit-AD-Wandelergebnis seriell im ASCII-Code senden)
$Debug              ;Symbole im Object-Code
$XRef               ;Quer-Verweis-Liste am Ende
$Registerbank (0)       ;benutzte Registerbank
$Nomod51                ;Standard-SFRs ausblenden
$Include    (REG842.inc)    ;ADuC-SFRs einblenden

NAME   ADW_KOMP     ;Modul-Name
;*************************************************************
;Zielhardware: ADuC832/842
;
;Aufgabe: Wandlung des 12bit Binärwertes aus einer AD-Wandlung in
;eine vierstellige BCD-Zahl im Bereich 0...5000
;Der ADW liefert 12bit, d.h. Binärwerte von 000h...FFFh, die
;Dezimalwerten von 0...4095 entsprechen.
;Der Maximalwert der Eingangsspannung des ADW ist 2,5V. Auf der
;benutzten Projektplatine ist dem OP ein 2:1 Spannungsteiler vor-;
;geschaltet, so daß 5V dem Maximalwert 4095 entsprechen.
;An diese Gegebenheiten wird die Software angepaßt.
;Im ersten Schritt wird der Wandelwert auf 5000mV=5V kalibriert,
;d.h. ADCDATA wird durch 4096 (2^12) dividiert und mit 5000
;multipliziert. Dabei ist es vorteilhaft, eine Multiplikation mit
;2,5 dadurch zu realisieren, daß der Multiplikand einmal links
;(x2) und einmal rechts (:2) geschoben wird und dann diese beiden
;Schiebeergebnisse addiert werden.
;Die Multiplikation mit 5000 wird wie folgt zerlegt:
;(2x2,5)x(4x2,5)x(4x2,5)x(4x2,5)
;Hieraus kann die Zweierpotenz 2x4^3=128 extrahiert und mit der ;
;Division durch 4096 verrechnet werden.
;Dann hat man (ADCDATA/32)x(2,5^4).
;Die Berechnung wird vorteilhaft so durchgeführt, daß in vier
;Schritten ADCDATA mit 2,5/2 multipliziert wird und das Ergebnis
;dann einmal rechts geschoben wird.
;Es wird davon ausgegangen, daß das obere Halbbyte in R7 (Bits
;3...0) und das untere Byte in R6 steht.
;Die Register R5 und R4 werden zur Aufnahme der rechtsgeschobenen
;(halbierten) Werte verwendet.
;Nach der Berechnung 2,5/2 sind die Inhalte von R7, R6 mit dem
;Ergebnis überschrieben - die ursprünglichen Inhalte werden im
;weiteren nicht mehr benötigt.
;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
;Benutzte Register:
;R7 - oberes Halbbyte - wird zuerst mit doppeltem Wert,
;    dann mit dem Endergebnis überschrieben
;R6 - unteres Byte - wird zuerst mit dem doppelten Wert,
;    dann mit dem Endergebnis überschrieben
;R5 - halbiertes oberes Halbbyte
;R4 - halbiertes unteres Byte
;Am Ende des Programms:
;R7 - Tausender im ASCII-Format
;R6 - Hunderter im ASCII-Format
;R5 - Zehner im ASCII-Format
;R4 - Einer im ASCII-Format
```

```
;*************************************************************
ORG 0000H               ;muß nicht angegeben werden
JMP START               ;Sprung zum Programmanfang
ORG 0060H               ;Startadresse des OP-Codes oberhalb der
                        ;Interrupt-Vektoren festlegen

START:                  ;Programmanfang

MOV     PLLCON,#00h     ;Kerntakt auf 16,77MHz

;ADuC842 UART auf 19200 Baud einstellen bei Kerntakt 16,77MHz

MOV     SCON,#70h       ;Mode 1, mit SM2=1 (Stoppbit prüfen)
                        ;Empfang freigegeben (REN=1)
MOV     T3CON,#85h      ;T3 als Baudratengenerator, DIV=5
MOV     T3FD,#2Dh       ;Fraktionalteiler auf 45

;Intervall-Timer vorbereiten, damit er 2s Pause erzeugt

MOV     TIMECON,#00010000b  ;Sekundenintervall festlegen
                        ;Intervallzähler nicht freigeben
                        ;noch kein Takt - TIEN, TCEN später setzen
MOV     INTVAL,#02      ;2 Sekunden Pause vorgeben

;ADW-Programmierung zum Start von Einzelwandlungen

MOV     ADCCON1,#10111100b    ;ADW aktivieren und Takt auf
                              ;fc/2=8,38MHz
                              ;4 Takte zur Akquisition einstellen

CALL    Pause           ;Einschwingvorgang des ADW
                        ;nach Reset abwarten
ADW:
MOV     ADCCON2,#00010011b  ;Einzelwandlung (Bit 4) und Kanal 3
                        ;Bit 4 wird automatisch zurückgesetzt,
                        ;wenn die Wandlung fertig ist
W_END:
MOV     A,ADCCON3       ;lesen zum Prüfen, ob ADW fertig
JB      ACC.7,W_END     ;weiter prüfen, ob BUSY-Flag (Bit 7)
                        ;noch gesetzt ist
;sobald "BUSY"=0 ist kann der Wandelwert gelesen werden

MOV     R6,ADCDATAL     ;unteres Byte vom ADW
MOV     A,ADCDATAH      ;oberes Byte vom ADW
ANL     A,#0Fh          ;obere 4 Bits auf Null setzen
MOV     R7,A

;Der folgende Programmabschnitt wird genau viermal durchlaufen

CALL    KALIB
CALL    KALIB
CALL    KALIB
CALL    KALIB

;jetzt sind noch R7, R6 durch Zwei zu dividieren
```

```
CLR     C               ;vorsorglich
MOV     A,R7
RRC     A               ;rechts schieben, wobei Bit 0 => C
MOV     R7,A            ;Endergebnis für oberes Halbbyte in R7
MOV     A,R6
RRC     A               ;rechts schieben, wobei C => Bit 7
                        ;und Bit 0 => C kommt
MOV     R6,A            ;Endergebnis für unteres Byte in R6
JC      INKR            ;wenn C=1, Ergebnis um Eins vergrößern, um
                        ;Rundungsfehler abzumildern
JMP     BIN_BCD

INKR:

MOV     A,#00h
ADDC    A,R6            ;C zu R6 addieren
MOV     R6,A            ;neuen Inhalt in R6
MOV     A,#00h
ADDC    A,R7            ;eventuellen Überlauf zu R7 addieren
MOV     R7,A            ;und neuen Inhalt zurückspeichern

JMP     BIN_BCD

;Unterprogramm Kalibrieren

KALIB:
                        ;halbieren
CLR     C               ; vorsorglich Carry löschen
MOV     A,R7
RRC     A               ;rechtsherum rotieren Bit 0 => C
MOV     R5,A            ;oberes halbiertes Byte in R5
MOV     A,R6
RRC     A               ;halbieren, wobei C => Bit 7
                        ;und Bit 0 => C kommt
MOV     R4,A            ;unteres halbiertes Byte in R4

;verdoppeln

CLR     C
MOV     A,R6
RLC     A               ;mal Zwei, wobei Bit 7 => C
                        ;und 0 => Bit 0 kommt
MOV     R6,A            ;zurückspeichern (überschreiben)
MOV     A,R7
RLC     A               ;oberes Halbbyte verdoppeln, wobei
                        ;C => Bit 0 kommt
MOV     R7,A            ;zurückspeichern (überschreiben)

;0,5x und 2x addieren

MOV     A,R4
CLR     C           ;vorsorglich
ADD     A,R6        ;halbes und doppeltes unteres Byte addieren
MOV     R6,A        ;Ergebnis für unteres Byte in R6 ablegen
MOV     A,R5        ;halbes und doppeltes oberes Halbbyte addieren
ADDC    A,R7    ;eventuellen Überlauf aus unterem Byte mitaddieren
```

```
MOV     R7,A    ;Ergebnis für oberes Halbbyte in R7 ablegen

;durch Zwei teilen

CLR     C               ;vorsorglich
MOV     A,R7
RRC     A               ;rechts schieben, wobei Bit 0 => C
MOV     R7,A            ;Ergebnis oberes Halbbyte
MOV     A,R6
RRC     A               ;rechts schieben, wobei C => Bit 7 kommt
MOV     R6,A            ;Ergebnis für unteres Byte in R6 ablegen

RET

;Unterprogramm Binär-BCD-Umwandlung
;*************************************************************
;in diesem Programmabschnitt wird der auf 5000mV kalibrierte Wert
;(1388h) in den Registern R7, R6 in BCD-Format gewandelt und das
;Ergebnis wird in die Register R5 (Tausender-Hunderter) und R4
;(Zehner-Einer) geschrieben
;Das niederwertige Byte aus R6 wird zunächst mit Hilfe des DIV AB-
;Befehls in Hunderter, Zehner und Einer zerlegt. Dann werden nach
;Maßgabe der höherwertigen Bits in R7 (4...0) die Dezimalwerte
;4096, 2048, 1024, 512, 256 addiert, wobei jeweils eine Korrektur
;mit DA A erfolgt, so daß schließlich die gültigen BCD Ziffern in
;den Halbbytes von R5 und R4 stehen
;*************************************************************
BIN_BCD:

MOV     R5,#00h
MOV     R4,#00h
MOV     A,R6
MOV     B,#100          ;Inhalt von R6 durch 100 teilen
DIV     AB              ;wobei das ganzzahlige Ergebnis (Hunderter)
                        ;in A und der Rest in B steht
MOV     R5,A            ;Hunderter abspeichern
MOV     A,B             ;Rest (Zehner und Einer) in Akku holen
MOV     B,#10
DIV     AB          ;Rest durch 10 teilen (Zehner in A, Einer in B)
SWAP    A           ;Zehner in oberes Halbbyte
ADD     A,B         ;Einer dazuaddieren
MOV     R4,A        ;vorläufiges Ergebnis speichern

;jetzt die höherwertigen Bits in R7 prüfen

MOV     A,R7            ;höherwertiges Byte holen

;falls Bit 0 gesetzt, 256 zu R5, R4 addieren mit Dezimalkorrektur!

JNB     ACC.0,TEST1     ;falls "0", diesen Teil überspringen
                        ;und Bit 1 testen

;256 zu R5,R4 addieren

MOV     A,R4            ;Zehner und Einer holen
ADD     A,#56h          ;Achtung: Dezimalwert als HEX-Zahl!
DA      A               ;Dezimalkorrektur
```

```
MOV     R4,A            ;korrigierten Wert abspeichern
MOV     A,R5            ;Hunderter holen
ADDC    A,#02h          ;mit Carry aus Zehnern addieren
DA      A               ;Dezimalkorrektur
MOV     R5,A            ;Tausender - Hunderter abspeichern

TEST1:

MOV     A,R7            ;höherwertiges Byte wieder holen

;falls Bit 1 gesetzt, 512 zu R5, R4 addieren mit Dezimalkorrektur!

JNB     ACC.1,TEST2     ;falls "0", diesen Teil überspringen
                        ;und Bit 2 testen

;512 zu R5,R4 addieren

MOV     A,R4            ;Zehner und Einer holen
ADD     A,#12h          ;Achtung: Dezimalwert als HEX-Zahl!
DA      A               ;Dezimalkorrektur
MOV     R4,A            ;korrigierten Wert abspeichern
MOV     A,R5            ;Hunderter holen
ADDC    A,#05h          ;mit Carry aus Zehnern addieren
DA      A               ;Dezimalkorrektur
MOV     R5,A            ;Tausender - Hunderter abspeichern

TEST2:

MOV     A,R7            ;höherwertiges Byte wieder holen

;falls Bit 2 gesetzt, 1024 zu R5, R4 addieren mit Dezimalkorrektur

JNB     ACC.2,TEST3     ;falls "0", diesen Teil überspringen
                        ;und Bit 3 testen

;1024 zu R5,R4 addieren

MOV     A,R4            ;Zehner und Einer holen
ADD     A,#24h          ;Achtung: Dezimalwert als HEX-Zahl!
DA      A               ;Dezimalkorrektur
MOV     R4,A            ;korrigierten Wert abspeichern
MOV     A,R5            ;Hunderter holen
ADDC    A,#10h          ;mit Carry aus Zehnern addieren
DA      A               ;Dezimalkorrektur
MOV     R5,A            ;Tausender - Hunderter abspeichern

TEST3:

MOV     A,R7              ;höherwertiges Byte wieder holen

;falls Bit 3 gesetzt, 2048 zu R5, R4 addieren mit Dezimalkorrektur

JNB     ACC.3,TEST4     ;falls "0", diesen Teil überspringen
                        ;und Bit 4 testen

;2048 zu R5,R4 addieren

MOV     A,R4            ;Zehner und Einer holen
ADD     A,#48h          ;Achtung: Dezimalwert als HEX-Zahl!
```

```
DA      A               ;Dezimalkorrektur
MOV     R4,A            ;korrigierten Wert abspeichern
MOV     A,R5            ;Hunderter holen
ADDC    A,#20h          ;mit Carry aus Zehnern addieren
DA      A               ;Dezimalkorrektur
MOV     R5,A            ;Tausender - Hunderter abspeichern

TEST4:

MOV     A,R7            ;höherwertiges Byte wieder holen

;falls Bit 4 gesetzt, 4096 zu R5, R4 addieren mit Dezimalkorrektur

JNB     ACC.4,ASCI      ;falls "0", fertig => zur ASCII-Darstellung

;4096 zu R5,R4 addieren

MOV     A,R4            ;Zehner und Einer holen
ADD     A,#96h          ;Achtung: Dezimalwert als HEX-Zahl!
DA      A               ;Dezimalkorrektur
MOV     R4,A            ;korrigierten Wert abspeichern
MOV     A,R5            ;Hunderter holen
ADDC    A,#40h          ;mit Carry aus Zehnern addieren
DA      A               ;Dezimalkorrektur
MOV     R5,A            ;Tausender - Hunderter abspeichern

;ASCII-Codes zum Senden über serielle Schnittstelle erzeugen und
;wie folgt ablegen:

;       - Tausender in R7
;       - Hunderter in R6
;       - Zehner in R5
;       - Einer in R4
;Die Ziffern 0...9 werden im ASCII-Code durch voranstellen von "3"
;im oberen Halbbyte repräsentiert

ASCI:

MOV     A,R5            ;Tausender und Hunderter holen
ANL     A,#0Fh          ;oberes Halbbyte (Tausender) löschen
ADD     A,#30h          ;eine "3" voranstellen
MOV     R6,A            ;ASCII-Hunderter fertig in R6

MOV     A,R5            ;nochmals Tausender und Hunderter holen
SWAP    A               ;Tausender in unteres Halbbyte
ANL     A,#0Fh          ;oberes Halbbyte (Hunderter) löschen
ADD     A,#30h          ;eine "3" voranstellen
MOV     R7,A            ;ASCII-Tausender fertig in R7
;-------------------------------------------------------------
MOV     A,R4            ;Zehner und Einer holen
SWAP    A               ;Zehner in unteres Halbbyte
ANL     A,#0Fh          ;oberes Halbbyte (Einer) löschen
ADD     A,#30h          ;eine "3" voranstellen
MOV     R5,A            ;ASCII-Zehner fertig in R5

MOV     A,R4            ;nochmals Zehner und Einer holen
ANL     A,#0Fh          ;oberes Halbbyte (Zehner) löschen
ADD     A,#30h          ;eine "3" voranstellen
```

```
MOV     R4,A            ;ASCII-Einer fertig in R4
;---------------------------------------------------------------
;Dieser Programmabschnitt sendet den Text, der an der Marke TITEL
;steht und danach in einer neuen Zeile die Inhalte der Register
;R7,R6,R5,R4 und zum Schluß die Einheit mV [6Dh='m' und 56h='V']

SENDE:
MOV     DPTR, #TITEL    ;Textanfang
CALL    SEND_TXT        ;mit Senden beginnen
MOV     A,R7            ;Meßwert Tausender
CALL    AUSGABE
MOV     A,R6            ;Meßwert Hunderter
CALL    AUSGABE
MOV     A,R5            ;Meßwert Zehner
CALL    AUSGABE
MOV     A,R4            ;Meßwert Einer
CALL    AUSGABE
MOV     A,#'m'          ;m
CALL    AUSGABE
MOV     A,#'V'          ;V
CALL    AUSGABE

;wenn Alles gesendet ist, wird eine Pause von 3s eingelegt und
;wieder von vorne begonnen, d.h. mit einer neuen AD-Wandlung
;für die Pause wird der Intervall-Timer des ADuC842 benutzt

CALL    PAUSE
JMP     ADW             ;neuer Wandel- und Sendevorgang

;--------- Unterprogramm für 2s Pause ------------------------
PAUSE:
MOV     PLLCON,#07h     ;Taktteiler durch 128 zur Verlangsamung
ORL     TIMECON,#00000011b  ;Bit 0 und Bit 1 setzen, d.h. TCEN
                            ;um den Intervalltimer zu starten
                        ;und TIEN für Freigabe des Intervallzählers
WART:
MOV     A,TIMECON       ;Bit 2 = TII in TIMECON prüfen
JNB     ACC.2,WART      ;warten bis Interrupt-Bit gesetzt ist

MOV     TIMECON,#00010000b    ;Stopp und Reload Langzeittimer
MOV     PLLCON,#00h           ;jetzt wieder schneller
RET

;Text senden - das Ende wird vom ASCII-Zeichen '0' markiert

SEND_TXT:
CLR     A               ;Akku leeren
MOVC    A,@A+DPTR       ;Zeichen holen
JZ      FERTIG          ;wenn '0', dann ist der Text fertig gesendet
```

```
CALL    AUSGABE         ;über Schnittstelle senden
INC     DPTR            ;auf nächstes Zeichen zeigen
JMP     SEND_TXT        ;senden

FERTIG:

RET

;ein Zeichen senden

AUSGABE:

CLR     TI              ;Transmit-Interrupt-Flag löschen
MOV     SBUF,A          ;Senden starten
JNB     TI,$            ;warten bis Sendevorgang beendet ist
                        ;d.h. bis TI gesetzt wird
RET                     ;nächstes Zeichen

TITEL:

DB 10,10,13,'Die Spannung an PIN 3 von Port 1 ist:',10,10,13

DB '    => ',0

END
```

## 4.8 Mikrocontroller-Entwicklungswerkzeuge und –Umgebungen

### 4.8.1 Das Arbeiten mit dem Makro-Cross-ASSEMBLER[21] ASS51

Das hier vorgestellte Standard-Assemblerpaket beinhaltet alle Programme und Hilfsmittel, die zur **modularen** Softwareerstellung für Mikrorechner, die auf dem 8051-Kern basieren, notwendig sind.

Wie Bild 4.98 zeigt, wird zuerst eine Quelldatei (Sourcecode) mit einem Text-Editor erstellt und z.B. als MYPROG.A51 abgespeichert. Danach wird daß Assemblerprogramm (z.B. **ASS51.EXE**) aufgerufen.

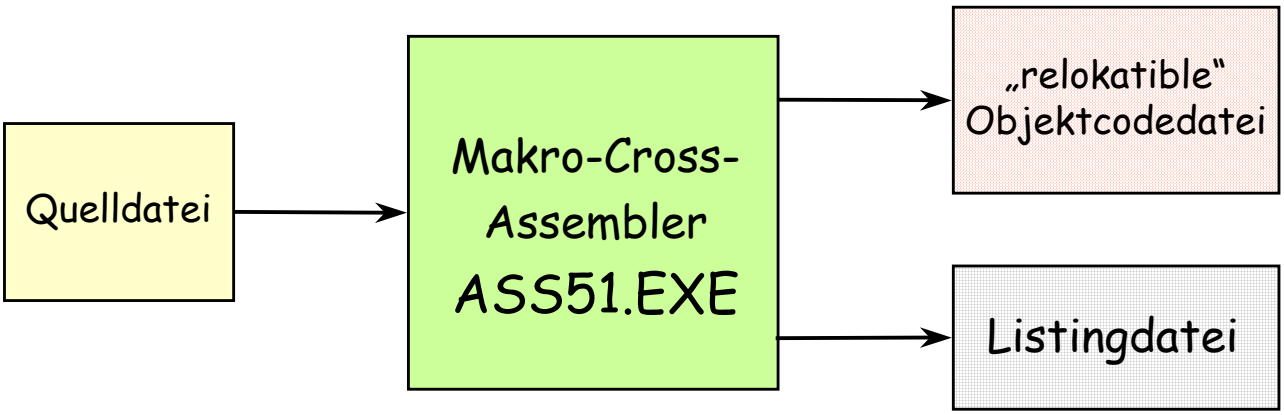



Bild 4.98: Symbolische Darstellung des Assembliervorgangs

[Im DOS-Fenster: **ASS51** MYPROG.A51]

Nach Ablauf des Assembliervorganges stehen eine „relokatible[22]“ Objektdatei

MYPROG.OBJ und eine Listing-Datei MYPROG.LST zur Verfügung.

Größere Projekte bestehen immer aus mehreren Quelldateien, die jede für sich assembliert wird. So entstehen mehrere relokatiblen Objektdateien die anschließend mit einem Programm, das „Linker/Locator“ **LINK.EXE** heißt, zusammengebunden und auf Programmspeicheradressen festgelegt werden.

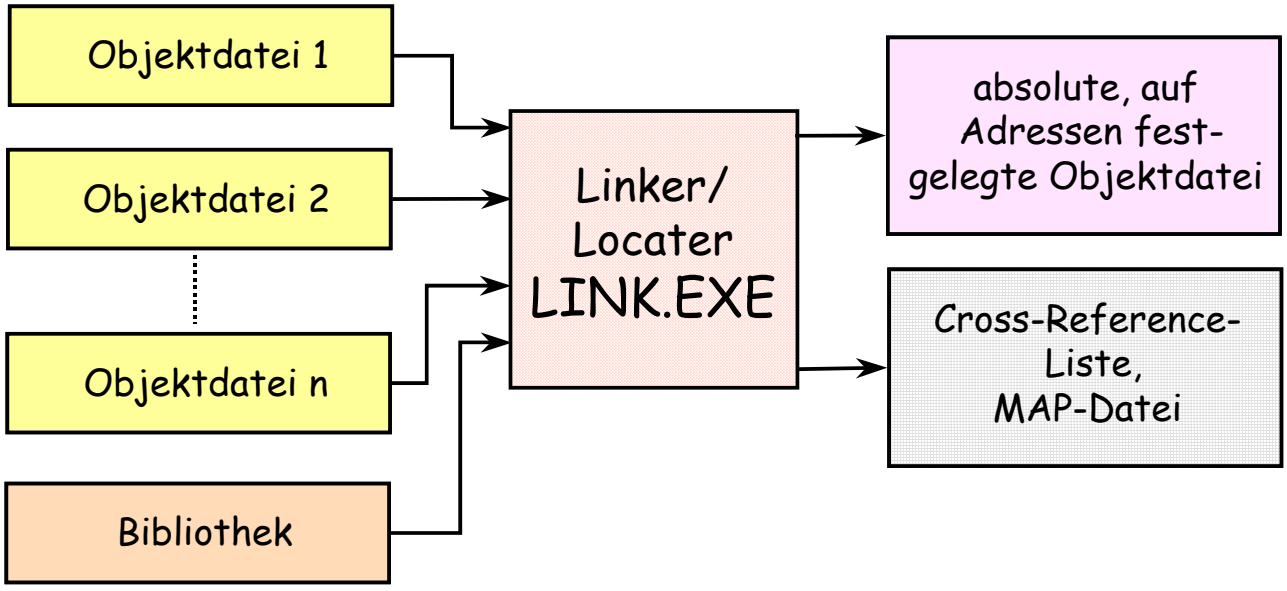



Bild 4.99: Zusammenfügen der Objektdateien und Adressenfestlegung

[21] das englische Wort assemble bedeutet zusammenbauen, zusammensetzen

[22] ortsveränderlich in Bezug auf die Lage im Programmspeicher des Mikrorechners

Ein Aufruf im DOS-Fenster ist z.B.

**LINK** Datei_1.obj, Datei_2.obj ... Dateil_n.obj **TO** Gesamt.obj

oder unter Verwendung einer Bibliothek

**LINK** Bibl_1.lib, Bibl_2.lib, Datei_x.obj **TO** Gesamt.obj

Eine zusammengebundene absolute, d.h. auf Programmspeicheradressen festgelegte Objektdatei kann zum Test mit Softwaresimulatoren oder auch für Emulatoren verwendet werden. Sie liegt hier im Intel-Objektformat vor, das nicht von allen Programmiergeräten gelesen werden kann.

Um generell Programmiergeräte bzw. entsprechende PC-basierte Software, die das Intel-Objektformat nicht „verstehen“, bedienen zu können, steht das Programm **O_H.EXE** zur Verfügung. Es nimmt eine Formatumwandlung vor, wobei das gewünschte Ausgangsformat als Parameter anzugeben ist. Ohne Angabe eines Parameters wird das Intel-HEX-Format erzeugt.

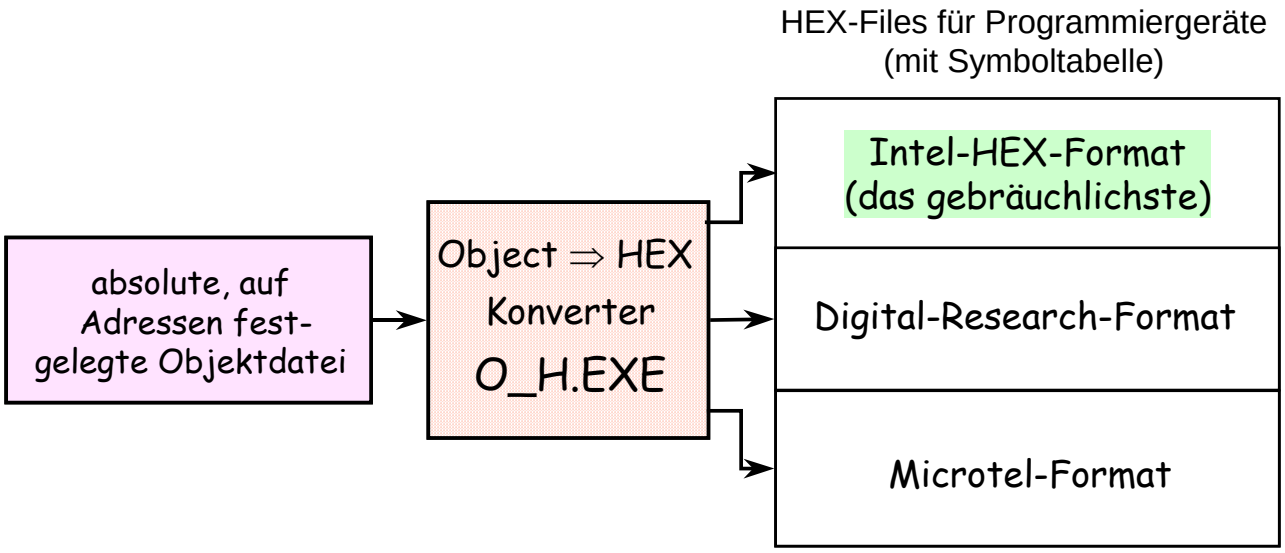



Bild 4.100: Arbeitsweise des Objekt-HEX-Konvertierungsprogramms O_H.EXE

Aufruf: **O_H** Gesamt.obj

Es entsteht eine Datei Gesamt.hex im Intel-HEX-Format.

Manche, einfache Programmiergeräte können auch das Intel-HEX-Format nicht verwenden, sondern benötigen das „kompakte“ BIN-Format (BIN steht für binär). Hierbei ist das Mikrorechner-Programm Byte für Byte in der richtigen Reihenfolge in einer Datei abgelegt, so daß keine Speicherung der Adressen nötig ist. Mit Hilfe des Programms **HEXBIN.EXE** kann aus einer HEX-Datei die entsprechende BIN-Datei erzeugt werden.

Aufruf: **HEXBIN** Gesamt.hex

Es entsteht eine Datei Gesamt.bin im Binärformat.

Mit dem Bibliotheksprogramm **LIB.EXE** können mehrere Objektdateien zu einer Bibliothek zusammengefaßt werden. Die Funktionen von **LIB** umfassen neben der Erzeugung von Bibliotheken unter anderem das Hinzufügen und Entfernen von Objektdateien, sowie das Auflisten aller eingebundenen Objektdateien.

Anwendungsbeispiel für die Bibliotheksfunktion:

Nach dem Aufruf **LIB.EXE** erscheint z.B. die Zeile:

MS-DOS MCS-51 Library Manager LIB51 V2.0

| | | |
|---|---|---|
| * | ;der Stern ist das “Prompt” für die Kommandos | |
| *h; | ;zeigt eine Kommandoübersicht | |
| **ADD** | {file[(module[,...])]} [,...] **TO** library_file | ;hinzufügen |
| **CREATE** | library_file | ;Bibliothek erzeugen |
| **DELETE** | library_file (module[,...]) | ;Bibliothek (Modul) löschen |
| **EXIT** | | ;beenden |
| **HELP** | | ;≡ h |
| **LIST** | {file[(module[,...])]} [,...] [**TO** file] [PUBLICS] | |
| **REPLACE** | {file[(module[,...])]} [,...] **IN** library_file | |

Beispiel:

*__LIST__ EMPF_DSP.lib ;Aufruf zum Auflisten der Bibliothek EMPF_DSP

PRAEAMBEL
TIMER
FREQUENZ
RADAR
SERIELL
DECISION
INTERRUPT
EINLESEN
*

#### 4.8.1.1 Assembler-Steueranweisungen (Direktiven)

Der Assembler ASS51 kennt verschiedene Direktiven, die dazu dienen, Symbole verschiedener Art zu definieren, Speicherbereiche zu reservieren bzw. zu initialisieren oder die spätere Position des Objektcodes im Programmspeicher des Ziel-Mikrorechners festzulegen.

Direktiven dürfen nicht mit Mikrorechnerbefehlen verwechselt werden; sie generieren mit Ausnahme von **DB** (define Byte) und **DW** (define Word) keinen Objektcode. Sie dienen vielmehr dazu, die Arbeitsweise des Assemblers zu beeinflussen und Informationen in die Objektdatei einzufügen (z.B. Segment-Definitionen).

Die Direktiven sind in folgende Kategorien eingeteilt:

- **Symbol-Definition**

  SEGMENT
  EQU
  SET

DATA
IDATA
XDATA
BIT
CODE

- **Speicherbereiche reservieren und initialisieren**

| | |
|---|---|
| DS | ;**D**ata **S**torage Reservation (Zahl der Bytes) |
| DB | ;**D**efine **B**yte: die folgenden Bytes in den Speicher schreiben |
| DW | ;**D**efine **W**ord: die folgenden Words (je 2Byte) in den Speicher schreiben |
| DBIT | ;**D**efiniere Reservierung von **BIT**s (Zahl der Bits) |

- **Direktiven für umfangreiche Programme mit vielen „Modulen“**

PUBLIC
EXTRN
NAME

- **Assembler-Zustandsdirektiven und Segmentauswahl**

| | |
|---|---|
| ORG | ;für „Origin“, d.h. Anfangsadresse bestimmen |
| END | ;schließt jedes Modul (Programmstück) ab |
| RSEG | ;aktiviert allgemein ein relokatibles Segment |
| CSEG | ;deklariert ein Segment im Programmspeicher |
| DSEG | ;deklariert ein Segment im internen Datenspeicher |
| XSEG | ;deklariert ein Segment im externen Datenspeicher |
| ISEG | ;deklariert ein Segment im internen indirekt adressierbaren Datenspeicher |
| BSEG | ;deklariert ein Segment im bitadressierbaren Bereich |
| USING | ;deklariert die zu verwendende Registerbank |

Der **ASS51** ist ein Zwei-Pass-Assembler. Im ersten „Pass“ (Durchlauf) werden Symbole registriert und die Länge der Befehle festgelegt. Im zweiten werden Vorwärtsreferenzen aufgelöst und der Objektcode wird erzeugt. Diese Struktur legt fest, daß Ausdrücke in ORG, DS und DBIT Anweisungen keine Vorwärtsreferenzen enthalten dürfen.

## Beispiele für Symbol-Definitionen

- **SEGMENT**

Mit der **SEGMENT**-Anweisung wird ein relokatibles Segment deklariert und ein optionaler Relokationstyp vergeben. Das Format ist wie folgt:

Rel_Seg_Name **SEGMENT** **Segment_Typ** [Reloc_Type]

Segmentnamen dienen dazu, Programmabschnitte zu definieren. Dies kann in mehreren Programmteilen (Modulen) erfolgen, wobei später die verschiedenen Segmente gleichen Namens und verträglichen Typs vom Linker/Locater **LINK** zu einem Segment kombiniert werden. Der Segmenttyp gibt an, welchem Adreßraum das Segment angehört:

**Segment-Typen**

| | |
|---|---|
| CODE | Programmspeicherbereich |
| XDATA | externer Datenspeicher |
| DATA | interner Datenspeicher (Adressen 0 bis 127) |
| IDATA | indirekt adressierbarer interner Datenspeicher (Adr. 0 bis 255) |
| BIT | bitadressierbarer Speicher (die Bytes 20H bis 2FH bzw. Bitadressen 00H...7FH im internen Datenspeicher) |

Die optionale Angabe eines Relokationstyps bewirkt die Art der Plazierung des Segments bei der Ausführung von **LINK**.

Es gibt folgende Relokationstypen:

| | |
|---|---|
| PAGE | spezifiziert ein Segment, dessen Anfangsadresse auf einem Vielfachen von 256 liegen muß. Die Festlegung der Adresse ist Sache von **LINK**. PAGE ist nur für Segmente des Typs CODE und XDATA zulässig. |
| INPAGE | spezifiziert ein Segment, das nach **LINK** komplett innerhalb eines 256-Byte-Blocks liegen muß. INPAGE ist nur für Segmente des Typs CODE und XDATA zulässig. |
| INBLOCK | spezifiziert ein Segment, das nach **LINK** komplett innerhalb eines 2048-Byte-Blocks liegen muß und ist nur für Segmente des Typs CODE gültig. |
| BITADRESSABLE | spezifiziert ein Segment, das von **LINK** innerhalb des bitadressierbaren Bereichs (20H bis 2FH) plaziert wird. BITADRESSABLE ist nur für Segmente des Typs DATA zulässig und limitiert diese auf eine Länge von 16 Byte. |
| UNIT | „Defaulteinstellung“, die ein Segment so spezifiziert, daß es auf einer „Einheit“ beginnt. Dies ist ein Bit für BIT-Segmente und ein Byte für alle anderen Segmente. |
| OVERLAYABLE | kennzeichnet Segmente, die mit anderen Segmenten überlagert werden können. Die Segmente müssen nach Spezifikationen der C-Compiler für 8051-Mikrocontroller deklariert werden. |

Bemerkung: Der Name eines Segments entspricht einem Segmentsymbol. Segmentsymbole in Ausdrücken repräsentieren die Basisadresse des Segments nach **LINK**.

Beispiel: Anlegen eines Stacks:

Da kein explizites Stacksegment existiert, kann ein beliebiges DATA- oder IDATA-Segment verwendet werden.

```
Name                                Typ       ;Kommentar
STACK       SEGMENT     IDATA
RSEG        STACK                 ;def. Segment auswählen
DS          10H                   ;16 Byte reservieren
MOV         SP,#STACK-1           ;Stackpointer laden
```

## Die EQU-Anweisung

Mit **EQU** kann ein Symbol definiert und diesem ein numerischer Ausdruck oder ein Registersymbol zugewiesen werden.

```
Symbol_Name_1     EQU     Ausdruck
Symbol_Name_2     EQU     Registersymbol
```

Der Ausdruck muß absolut oder einfach relokatibel sein und darf keine Vorwärtsreferenzen enthalten. Ein mit **EQU** definiertes Symbol kann als Operand in anderen Ausdrücken oder Adressen, abhängig vom Segmenttyp, verwendet werden. Der Segmenttyp wird dem des Ausdrucks gleichgesetzt. Ein typenloses Symbol (*NUMBER*) kann in allen Ausdrücken ohne Einschränkung verwendet werden. Ein mit **EQU** definiertes Symbol kann später nicht mehr neu definiert oder verändert werden.

## Die SET-Anweisung

**SET** arbeitet ähnlich wie **EQU**, mit der Ausnahme, daß damit definierte Symbole jederzeit einen neuen Wert erhalten können.

```
Symbol_Name_1     SET     Ausdruck
Symbol_Name_2     SET     Registersymbol
```

Die Zuweisung eines neuen Werts muß immer mit **SET** vorgenommen werden. Es ist daher nicht möglich, ein mit **SET** definiertes Symbol später mit der **EQU** neuerlich zu definieren und umgekehrt.

## Die BIT-Anweisung

Mit der **BIT**-Anweisung wird einem Symbol eine Bitadresse zugewiesen.

```
Symbol_Name       BIT     Bit_Adresse
```

Ein Bitsymbol darf später nicht mehr verändert werden.

### Die DATA-Anweisung

Mit **DATA** wird einem Symbol ein Wert zugewiesen, der sich auf den internen Datenspeicher des 8051 bezieht.

```
Symbol_Name          DATA          Ausdruck
```

Der numerische Ausdruck muß einen Wert zwischen 0 und 255 haben. Der Typ des Ausdrucks muß entweder *DATA* oder *NUMBER* (typenlos) sein. Ferner muß der Ausdruck absolut oder einfach relokatibel sein. Ein mit **DATA** erzeugtes Symbol darf später nicht mehr verändert werden.

### Die XDATA-Anweisung

Mit **XDATA** wird einem Symbol eine Adresse im externen Datenspeicher zugewiesen.

```
Symbol_Name          XDATA         Ausdruck
```

Der Typ des Ausdrucks muß entweder *XDATA* oder *NUMBER* (typenlos) sein. Jeder andere Typ führt zu einem Fehler. Der Ausdruck muß weiterhin entweder absolut oder einfach relokatibel sein. Ein mit **XDATA** erzeugtes Symbol darf später nicht mehr verändert werden.

### Die IDATA-Anweisung

Mit Hilfe der **IDATA**-Anweisung wird ein Symbol erzeugt, das sich auf einen indirekt adressierbaren internen Datenspeicherplatz bezieht.

```
Symbol_Name          IDATA         Ausdruck
```

Der Ausdruck muß den Typ *IDATA* oder *NUMBER* (typenlos) haben und entweder absolut oder einfach relokatibel sein. Ein **IDATA** erzeugtes Symbol darf später nicht mehr verändert werden.

### Die CODE-Anweisung

Mit Hilfe der **CODE**-Anweisung wird ein Symbol erzeugt, das sich auf eine Adresse im Programmspeicher bezieht.

```
Symbol_Name          CODE          Ausdruck
```

Ein Fehler tritt auf, wenn der Ausdruck nicht einen der Typen *CODE* oder *NUMBER* (typenlos) hat. Der Ausdruck muß absolut oder einfach relokatibel sein. Ein mit **CODE** erzeugtes Symbol darf später nicht mehr verändert werden.

## Anweisungen zum Initialisieren und Reservieren von Speicher

Die hier vorgestellten Anweisungen dienen zum Reservieren und Initialisieren von Wort-, Byte- und Bit-Einheiten. Der jeweils reservierte Bereich beginnt bei absoluten Segmenten an der momentanen Adresse, bei relokatiblen Segmenten bei dem momentanen „Offset“. Der aktuelle Adreßstand wird für jedes Segment getrennt geführt.

## Die DS-Anweisung

Mit **DS** kann eine zusammenhängende Anzahl von Bytes im Programmspeicher oder im internen oder externen Datenspeicher reserviert werden.

[Marke:] **DS** <numerischer Ausdruck>

Der numerische Ausdruck darf weder Vorwärtsreferenzen noch relokatible oder externe Symbole enthalten. Die **DS**-Anweisung ist außer in BIT-Segmenten in allen Segmenten gültig und bewirkt, daß der Adreßzähler im aktuellen Segment um den Betrag des Ausdrucks erhöht wird. Zu beachten ist, daß die obere Adreßgrenze des aktuellen Segments nicht überschritten wird; sie wird durch den jeweiligen Segment- und Relokationstyp bestimmt.

## Die DBIT-Anweisung

Mit **DBIT** kann eine zusammenhängende Anzahl von Bits in einem BIT-Segment reserviert werden:

[Marke:] **DBIT** <numerischer Ausdruck>

Der numerische Ausdruck darf weder Vorwärtsreferenzen noch relokatible oder externe Symbole enthalten. Die **DBIT**-Anweisung bewirkt, daß der Adreßzähler im aktuellen BIT-Segment um den Betrag des Ausdrucks erhöht wird. Zu beachten ist, daß hier Bit-Einheiten summiert werden, nicht Bytes.

## Die DB-Anweisung

Mit **DB** kann eine Folge von Bytes im Programmspeicher initialisiert werden.

[Marke:] **DB** Ausdruck[, Ausdruck,...]

Die Angabe einer Marke (engl: Label) ist – wie auch jeweils in den obigen Anweisungen - optional. Die Ausdrucksliste kann eine Folge von Zahlen, Symbolen, Zeichenketten oder Ausdrücken sein, die jeweils durch Komma zu trennen sind. Eine Zeichenkette innerhalb eines Ausdrucks darf höchstens zwei Zeichen beinhalten. Alleinstehende Zeichenketten dürfen jedoch auch mehr als zwei Zeichen beinhalten; wenn sie die Länge Null haben entsteht allerdings kein Objektcode. Einer vorhandenen Marke wird die Adresse des ersten Bytes der Objektliste zugeordnet. Ein Fehler tritt auf, wenn **DB** nicht innerhalb eines Code-Segments verwendet wird.

Eine spezielle Unterscheidung bei Zeichenketten, die einen Ausdruck einleiten, ist nicht notwendig, d.h. der ASS51 stellt automatisch fest, ob die Zeichenkette alleinstehend ist oder einem Ausdruck angehört.

## Die DW-Anweisung

Mit der **DW**-Anweisung können Zwei-Byte-Folgen (Word) im Programmspeicher initialisiert werden.

[Marke:] **DW** Ausdruck[, Ausdruck,...]

Die Angabe einer Marke ist wieder optional. Die Ausdrucksliste kann eine Folge von Zahlen, Symbolen, Zeichenketten oder Ausdrücken sein, die jeweils durch Komma zu trennen sind. Eine Zeichenkette darf höchstens zwei Zeichen beinhalten.

Alleinstehende Zeichenketten mit der Länge Null generieren analog zur **DB** keinen Objektcode. Einer vorhandenen Marke wird die Adresse des ersten Bytes der Objektliste zugeordnet. Ein Fehler tritt auf, wenn **DW** nicht innerhalb eines Code-Segments verwendet wird.

Eine spezielle Unterscheidung bei Zeichenketten, die einen Ausdruck einleiten, ist wie bei **DB** nicht notwendig.

## Anweisungen für umfangreiche Programme mit zahlreichen Modulen

(Programmabschnitte, die jeweils separat entwickelt werden)

Sinn der im folgenden diskutierten Anweisungen ist die Definition von Symbolen, die es erlauben, daß mit dem Linker/Locater **LINK** mehrere Objektdateien zu einem absoluten Modul zusammengefaßt werden können. Dabei ist es notwendig, Namen zu publizieren bzw. als **extern** - d.h. an anderer Stelle bereits eingeführt - zu deklarieren.

### Die PUBLIC-Anweisung

Mit **PUBLIC** werden Namen „publiziert“, die damit für andere Module verfügbar werden, indem sie dort mit der Anweisung **EXTRN** deklariert werden.

**PUBLIC** Symbol[, Symbol,...]

Nach der **PUBLIC**-Anweisung kann eine Anzahl von Namen, jeweils durch Komma getrennt, angegeben werden. Jeder der angegebenen Namen muß irgendwo im Quellprogramm definiert sein, wobei auch Vorwärtsreferenzen zulässig sind. Bei Register- und Segment-Symbolen ist **PUBLIC** nicht erlaubt.

### Die EXTRN-Anweisung

Mit **EXTRN** werden dem Assembler Symbole bekannt gegeben, die in einem anderen Programmstück (Modul) als dem gerade zu assemblierenden definiert wurden.

**EXTRN** Segment_typ(Symbol_liste)

**EXTRN**-Anweisungen können sich im Quellprogramm an beliebiger Stelle befinden. Jedem der externen Symbole ist ein Segmenttyp (CODE, DATA, IDATA, XDATA, BIT oder NUMBER) eindeutig zugeordnet, der die Verwendung entsprechend eingrenzt. Externe CODE-Symbole können als Ziel eines JMP- oder CALL-Befehls dienen, jedoch nicht als Ziel von MOV. **LINK** überprüft während des Binderlaufs den Segmenttyp des externen (publizierten) Symbols.

Beispiele:

```
EXTRN   CODE(CLR_FLG), DATA(BUFF_1)
EXTRN   CODE(ASCII_TAB, BIN_ASC)
EXTRN   NUMBER(TABELLE)
```

### Die NAME-Anweisung

Mit der **NAME**-Anweisung kann das während die Übersetzungslaufs erzeugte Objektmodul mit einem Namen versehen werden.

**NAME** Name_des_Objektmoduls

Der Name kann bis zu 40 Zeichen lang sein und sollte aus Gründen der Kompatibilität nicht mit einer Ziffer beginnen. Ansonsten gelten dieselben Regeln wie bei Symbolen. Zu beachten ist, daß lediglich der Name eines „Objektmoduls“ festgelegt wird und nicht der Dateiname, unter dem das Objektmodul abgelegt wird.

Wenn keine **NAME**-Anweisung im Quellprogramm gefunden wird, so wird der Name der Quelldatei (ohne Pfad und Dateierweiterung) als Name verwendet. Innerhalb eines Moduls darf nur eine **NAME**-Anweisung stehen.

## Status- und Segmentauswahl-Anweisungen

### Die END-Anweisung

Die **END**-Anweisung muß stets in der letzten Zeile eines Quellprogramms stehen und darf daher nur einmal verwendet werden. Ist dies nicht der Fall, so werden alle folgenden Textzeilen bis zum physikalischen Dateiende ignoriert, d.h. sie werden nicht übersetzt.

### Die ORG-Anweisung

Mit **ORG** wird die Startadresse für nachfolgende Maschinenbefehle oder Konstanten im Programmspeicher im aktuellen Segment festgelegt.

**ORG** Ausdruck

Der Ausdruck muß entweder absolut oder einfach relokatibel sein, d.h. es dürfen nur absolute oder dem aktuellen Segment zugehörende Symbole verwendet werden, und er darf keine Vorwärtsreferenzen haben. Der Assembler berechnet den Ausdruck und verändert den Adreßzähler entsprechend. Ist das aktuelle Segment ein absolutes Segment, so wird der neue Adreßzählerstand ein absoluter Wert. In einem relokatiblen Segment wird das Ergebnis des Ausdrucks als „Offset“ verwendet.

Mit der **ORG**-Anweisung wird der Adreßzähler-Stand verändert. Dabei wird kein neues Segment erzeugt. Eine mögliche Adreßlücke wird Teil des aktuellen Segments. In absoluten Segmenten kann der Adreßzähler nicht unter die Segmentbasis gelegt werden.

Mit den folgenden Anweisungen werden Segmente erzeugt und ausgewählt. So wird festgelegt, in welchem Adreßraum die nachfolgenden Mikrorechnerbefehle plaziert werden. Eine Einstellung gilt jeweils solange, bis eine neue Auswahl erfolgt.

## Die RSEG-Anweisung

Mit RSEG wird ein früher definiertes, relokatibles Segment aktiviert und wird damit zum aktuellen Segment.

**RSEG** Segment_Name

Absolute, d.h. unmittelbar auf Speicheradressen festgelegte Segmente werden mit folgenden Anweisungen erzeugt:

CSEG [AT absolute_Adresse]
DSEG [AT absolute_Adresse]
XSEG [AT absolute_Adresse]
ISEG [AT absolute_Adresse]
BSEG [AT absolute_Adresse]

Wird die optionale, absolute Basisadresse des Segments nach AT angegeben, so schließt der Assembler das zuletzt eingestellte Segment und erzeugt ein neues. Fehlt die AT-Angabe, so wird ein früheres Segment gleichen Segmenttyps aktiviert und fortgesetzt. Ist kein solches vorhanden, so wird ein neues Segment erzeugt und dessen Adreßzähler auf Null gesetzt. Der Ausdruck für 'absolute_Adresse' muß absolut und ohne Vorwärtsreferenzen sein.

Der Assembler führt für jedes Segment einen eigenen Adreßzähler, wobei ein absolutes CODE-Segment mit der Adresse 0 voreingestellt ist. Die Auswahl eines Segments bewirkt, daß der jeweils zuletzt vorhandene Stand des Adreßzählers für dieses Segment zum aktuellen Adreßzähler-Stand wird.

## Die USING-Anweisung

Mit **USING** wird dem Assembler angegeben, welche Registerbank (0..3) im nachfolgenden Programmteil zu verwenden ist.

**USING** Ausdruck

Der Ausdruck muß 0, 1, 2 oder 3 sein. Damit werden die absoluten Adressen für die danach vom Assembler verwendbaren ARn-Symbole festgelegt. AR0 ist z.B. die Adresse von Arbeitsregister R0 in der über **USING** gewählten Registerbank. Die Verwendung von ARn-Symbolen setzt somit eine vorherige **USING**-Anweisung voraus.

Bemerkung: Die **USING**–Anweisung schaltet keineswegs Registerbänke um. Dieses muß nach wie vor vom Programmierer gemacht werden. Die angegebene Registerbank wird lediglich im Objektcode markiert und zur weiteren Verwendung für **LINK** reserviert.

## Vorgehensweise bei der Erstellung umfangreicher Mikrorechnerprogramme mittels Modultechnik

- das Gesamtprogramm in geeignete etwa gleich große relokatible Unterabschnitte (Module) gliedern
- die einzelnen Module mit ASSM51 assemblieren
- gegebenenfalls die assemblierten Module (name.obj) in eine Bibliothek (Bibl.lib) mit **LIB.EXE** zusammenfassen

- alle Module inkl. Bibliotheken zusammenbinden und mit dem Linker/Locater **LINK.EXE** auf absoluten Adressen plazieren
- das resultierende Gesamtprogramm Gesamt.obj in einen Simulator oder Emulator laden und testen. Wenn nötig, Gesamt.obj z.B. in Intel-HEX-Format oder Binärformat konvertieren und über eine Programmiereinrichtung in den Programmspeicher des Ziel-Mikrorechners schreiben.
- Hard- und Softwaretest in der endgültigen Umgebung durchführen

## Beispiel für eine Programmdatei im Intel-Hex-Format

```
:03000000020000FB
:10000000758100C2D3C2D4750004D294C297C29540
:10001000D29700C2977800E500F2758000D295C2B1
:1000200095209642780074F274ADF2749AF274DE
:10003000DBF2758009D295C2957400F27400F274F7
:1000400001F274FFF275800BD295C295003096FCD8
:100050007400F27400F27400F27400F275800BD236
:1000600095C29502000078007400F274ACF274D16D
:10007000F27484F2758009D295C29578007400F20A
:1000800074AEF27464F2742BF2758008D295C29546
:10009000D200C200D200C200D200C200D200C20010
:1000A000D200C200D200C200D200C200D200C20000
:1000B000D200C200C200D200D200C200D200D200E0
:1000C000C200D200C200D200D200C200C200C200F0
:1000D0007400F27400F27401F274FFF275800BD2B6
:1000E00095C295A2009294120000A2009294120070
:1000F00000A2009294120000A2009294120000A2AA
:1001000009294120000A2009294120000A20092A9
:100110009412000A2009294120000A20092941285
:10012000000A2009294120000A20092941200001B
:10013000A2009294120000A2009294120000A20069
:1001400009294120000A2009294120000A2009294D5
:1001500012000A2009294120000A20092941200D9
:100160000A2009294120000A2009294120000A239
:1001700009294120000A2009294120000A2009239
:100180009412000A2009294120000A20092941215
:10019000000A2009294120000A2009294120000AB
:1001A000A2009294120000A2009294120000A200F9
:1001B0009294120000A2009294120000A200929465
:1001C00012000074FFD3D294C200200045794D0084
:1001D0000000D9FB0000009200B0E39200A200B33F
:1001E00082E3720033F5009291B290400DC29464A4
:1001F000FF7002D200E500020000D29464FF70029A
:10020000D200E5000200007B51000000000000DB8E
:10021000FB227400F27400F27400F27400F2758034
:100220000BD295C2957D60007C4400DCFDDDF8D2E8
:1002300094209603020000 7400F27400F27401F23C
:1002400074FFF275800BD295C295020000454853A9
:100250002D46534B2D56657273696F6E20323430C4
:1002600030626974 2F73206D6974205072616 5610A
:100270006D62656C203942353820756E6420505AA5
:1002800046202839 2C3429626592050312E363DAC
:1002900030204461756572746F6E203133322C34B6
:1002A000356B487A3B20556D736368616C74756E6D
:1002B000672064796E2E206F686E652052657365C5
:0102C00074C9
:00000001FF
```

Beispiel für eine Programmdatei im Binärformat

```
00: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
10: 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
20: 00 02 04 06 08 0A 0C 0E 10 12 14 16 18 1A 1C 1E
30: 00 03 06 09 0C 0F 12 15 18 1B 1E 21 24 27 2A 2D
40: 00 04 08 0C 10 14 18 1C 20 24 28 2C 30 34 38 3C
50: 00 05 0A 0F 14 19 1E 23 28 2D 32 37 3C 41 46 4B
60: 00 06 0C 12 18 1E 24 2A 30 36 3C 42 48 4E 54 5A
70: 00 07 0E 15 1C 23 2A 31 38 3F 46 4D 54 5B 62 69
80: 00 08 10 18 20 28 30 38 40 48 50 58 60 68 70 78
90: 00 09 12 1B 24 2D 36 3F 48 51 5A 63 6C 75 7E 87
A0: 00 0A 14 1E 28 32 3C 46 50 5A 64 6E 78 82 8C 96
B0: 00 0B 16 21 2C 37 42 4D 58 63 6E 79 84 8F 9A A5
C0: 00 0C 18 24 30 3C 48 54 60 6C 78 84 90 9C A8 B4
D0: 00 0D 1A 27 34 41 4E 5B 68 75 82 8F 9C A9 B6 C3
E0: 00 0E 1C 2A 38 46 54 62 70 7E 8C 9A A8 B6 C4 D2
F0: 00 0F 1E 2D 3C 4B 5A 69 78 87 96 A5 B4 C3 D2 E1
```

Vereinfachung des Assembliervorganges durch eine BATCH-Datei anstelle schrittweiser Eingaben in der DOS-Kommandozeile:

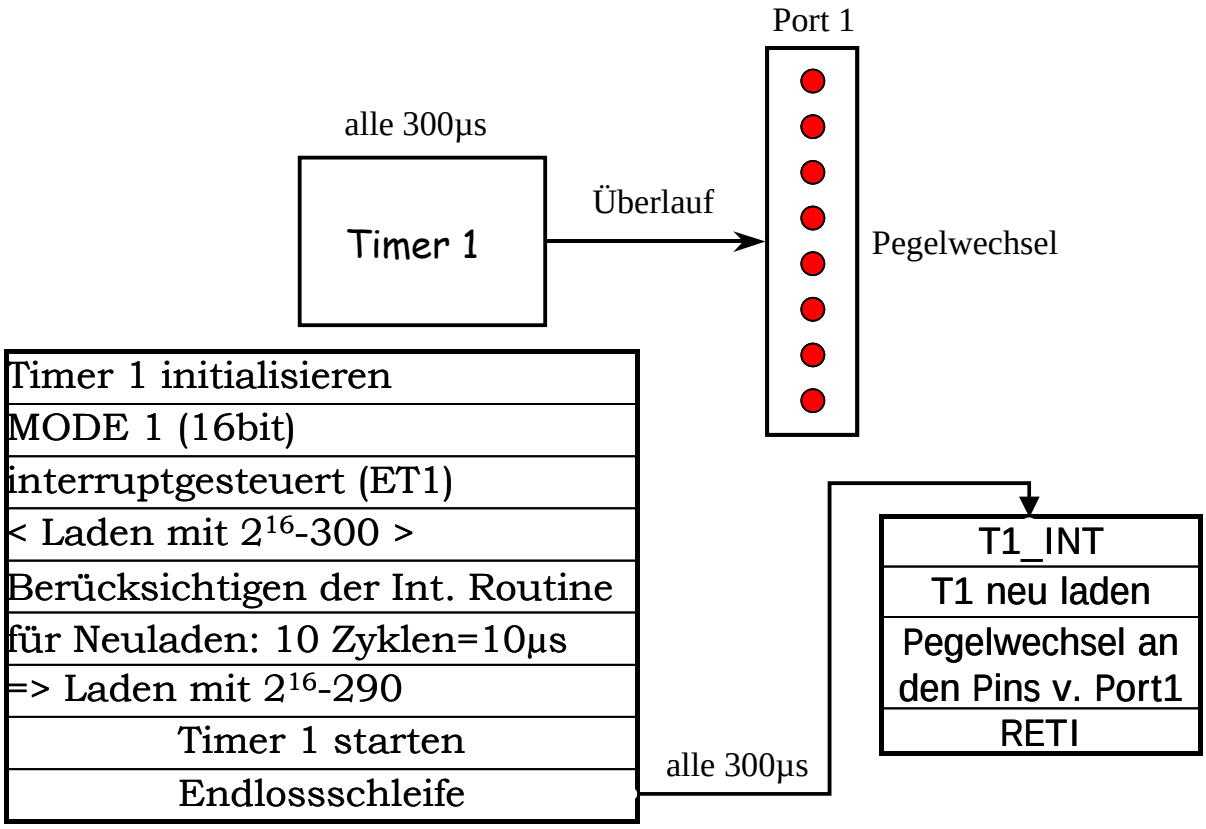
Die BATCH-Datei z.B. mit dem Namen a.bat führt folgende Schritte aus:

```
ass51 %1.a51                ;assemblieren   (%1 ( TIM_PORT)
link %1.obj to %1           ;link und locate
o_h %1                      ;Object-Hex-Umwandlung
hexbin %1.hex               ;HEX-BIN-Umwandlung
del %1                      ;nicht mehr benötigte Files löschen
del %1.m51
del %1.obj
```

```
Aufruf im DOS-Fenster für das Beispielprogramm: a TIM_PORT
```

Nach Ablauf stehen TIM_PORT.HEX und TIM_PORT.BIN zur Simulation oder zum Einschreiben in den Programmspeicher des Ziel-Mikrorechners zur Verfügung.

# Beispielprogramm (TIM_PORT)




```
$Title (Projekt: Timer 1 gibt Pulse an Port 1 aus   10.07.2003)
$Debug                          ;Symbole im Object-Code
$Xref                           ;Querverweis-Liste am Ende
$Registerbank (0)               ;benutzte Registerbank/baenke
$Nomod51                        ;8051-Registersatz abschalten
$Include (REG52.INC)            ;erweiterten Registersatz laden

NAME            TIM_PORT        ;Modul-Name
;*****************************************************************************
; Dieses Programm schaltet Pins am Port 1 im Rhythmus der Überläufe von Timer1
; Nach jeweils 300 Takten erfolgt ein Timer-Überlauf, der in einer Interrupt-
; Bedienungsroutine behandelt wird.
; Zielhardware: 80C51 kompatibler MC, z.B. ATMEL 89LS8252
; Für die Zeiterzeugung wird Timer 1 in Mode 1 (16bit Timer) mit Reload in der
; Interrupt-Routine benutzt, so daß alle 300µs ein Interrupt generiert wird
;#############################################################################
;Benutzte Register:             R2

USING 0                         ;Registerbank 0 benutzen
;-----------------------------------------------------------------------------
; Segment-Definitionen
;-----------------------------------------------------------------------------
C_AD_Seg        Segment         Code            ;Programmcode
Stack           Segment         Idata           ;ab Adr. 128...255 im 8052
D_AD_Seg        Segment         Data            ;Byte-Variable
B_AD_Seg        Segment         Bit             ;bitadressierbarer Datenspeicherbereich
```

```
;-------------------------------------------------------------------------------------------
; Allgemeine und Register-Zuweisungen
;-------------------------------------------------------------------------------------------
ZEIT_Z                  EQU         R2    ;hier nur als "Dummy"
;-------------------------------------------------------------------------------------------
; Bit-Zuweisungen
;-------------------------------------------------------------------------------------------
B1                      BIT         P1.0    ;Port-Pins mit Namen versehen
B2                      BIT         P1.1
B3                      BIT         P1.2
B4                      BIT         P1.3
B5                      BIT         P1.4
B6                      BIT         P1.5
B7                      BIT         P1.6
B8                      BIT         P1.7

;*****************************************************************************
; Segment im internen Datenbereich aktivieren
;-------------------------------------------------------------------------------------------
RSEG            Stack           ;im RAM
                DS 50           ;50 Stackplätze reservieren
;-------------------------------------------------------------------------------------------
;Segment im Datenbereich aktivieren
;-------------------------------------------------------------------------------------------
RSEG            D_AD_Seg

T1REL_H:        DS              1           ;Reload H-Byte für Timer 1
T1REL_L:        DS              1           ;REload L-Byte für Timer 1
;*****************************************************************************
; Interruptvektoren und Sprungziele
;-----------------------------------------------------------------
CSEG            AT 0000H        ;Programmstart nach Reset
                JMP ANFANG

CSEG            AT 001BH        ;Timer 1 Interrupt-Vektor (TF1)
                JMP T1_INT
;-------------------------------------------------------------------------------------------
; Segment im bitadressierbaren Bereich aktivieren, wird für FLAGs benötigt
;-------------------------------------------------------------------------------------------
RSEG            B_AD_Seg

FLG_B:          DBIT 5          ;fünf Plätze im bitadress. Bereich für Flags reservieren
                                ;vorsorglich, falls irgendwo benötigt
;*****************************************************************************
; Programm-Segment aktivieren
;-------------------------------------------------------------------------------------------
RSEG            C_AD_Seg                ;OP-Code beginnt hier

ANFANG:         MOV  SP,#Stack-1        ;Stack-Zeiger setzen
                CLR RS0                 ;Bank 0 wählen
```

```
                CLR RS1

                MOV T1REL_H,#0FEh       ;T1 Laden mit 2^16-300+10=65246 => FEDE
                MOV T1REL_L,#0DEh       ;290µs bis Überlauf +10µs für Interrupt-Bed.
;-------------------------------------------------------------------------------------------------------------
;Interrupt-Freigabe
;-------------------------------------------------------------------------------------------------------------
                MOV IE,#10001000b               ;T1- Interrupt freigeben
;-----------------------------------------------------------------------------------------
;Timer1 wird zur Zeitschritterzeugung benutzt: Hier 400µs-Schritte
;-----------------------------------------------------------------------------------------
INIT_T1:        MOV TMOD,#00010000b     ;Timer1 Betriebsart 1: 16bit
                                        ;TR1 noch "0" (Stopp)
                MOV TH1,T1REL_H         ;Vorladen
                MOV TL1,T1REL_L         ;mit Frequenzvorgabe
                SETB TR1                ;Timer 1 starten
                ANL P1,#01010101b       ;Port-Pins belegen
;-------------------------------------------------------------------------
;Endlosschleife
;*************************************************
ENDLOS:         NOP
                JMP ENDLOS
;-------------------------------------------------------------------------
; Timer-1-Interrupt-Bedienung
;-------------------------------------------------------------------------
T1_INT:         CLR TR1                 ;Timer 1 stoppen
                MOV TH1,T1REL_H         ;Nachladen
                MOV TL1,T1REL_L         ;mit Frequenzvorgabe

                SETB    TR1             ;Timer 1  wieder starten

                CPL     B1
                CPL     B2
                CPL     B3
                CPL     B4
                CPL     B5
                CPL     B6
                CPL     B7
                CPL     B8

                RETI
;-------------------------------------------------------------------------
DB              'Ein kurzes, einfaches Demonstrationsprogramm'
;-------------------------------------------------------------------------
END
```

### 4.8.1.2 Wie kommt das Programm in den Speicher eines Mikrorechners

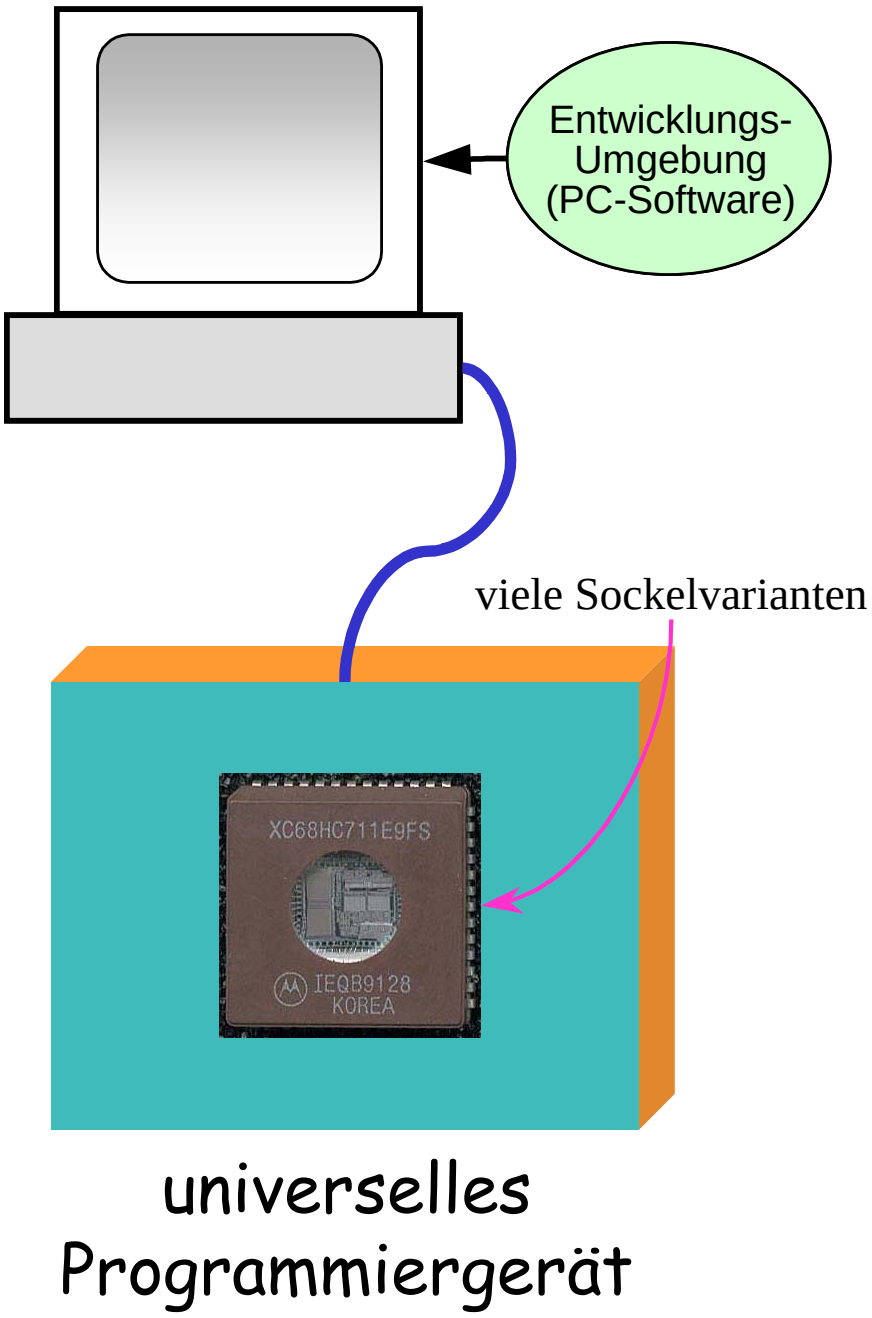



Bild 4.101: Programmiervorgang mit einem externen Universal-programmierer, z.B. DATA I/O 3900

**Universalprogrammierer**

Sobald ein Mikrorechner-programm im Intel-HEX-Format oder im Binärfor-mat vorliegt, kann es mit einem sogenannten **Uni-versalprogrammierer** z.B. direkt in das integrierte EPROM eines Mikrocon-trollers oder in ein separa-tes EPROM, das extern an das Bussystem des Mikro-controllers anzuschließen ist, geschrieben werden. Zum Programmieren muß der Baustein aus der An-wenderschaltung entnehm-bar sein und in einen ge-eigneten den Sockel des Programmiergerätes einge-steckt werden. Wegen der Vielfalt der Sockeltypen muß das Programmiergerät mit aufwendigem mechani-schem Zubehör ausgestattet sein. Dies trägt nicht unwe-sentlich zu den hohen Ko-sten eines solchen Gerätes bei, die sich im Bereich 10-20k€ bewegen. In den 80-er und 90-er Jahren war man leider auf solche Uni-versalprogrammierer an-gewiesen, um eine breite Palette von Mikrorechnern bedienen zu können. Konn-te man sich jedoch auf eine Rechnerfamilie (z.B. 8051) und wenige EPROM-Typen (25C64....27C512) spezialisieren, standen preisgünstige Lösungen im Bereich 100...1000€ zur Verfügung.

**Die „In-System“-Programmierung (ISP)**

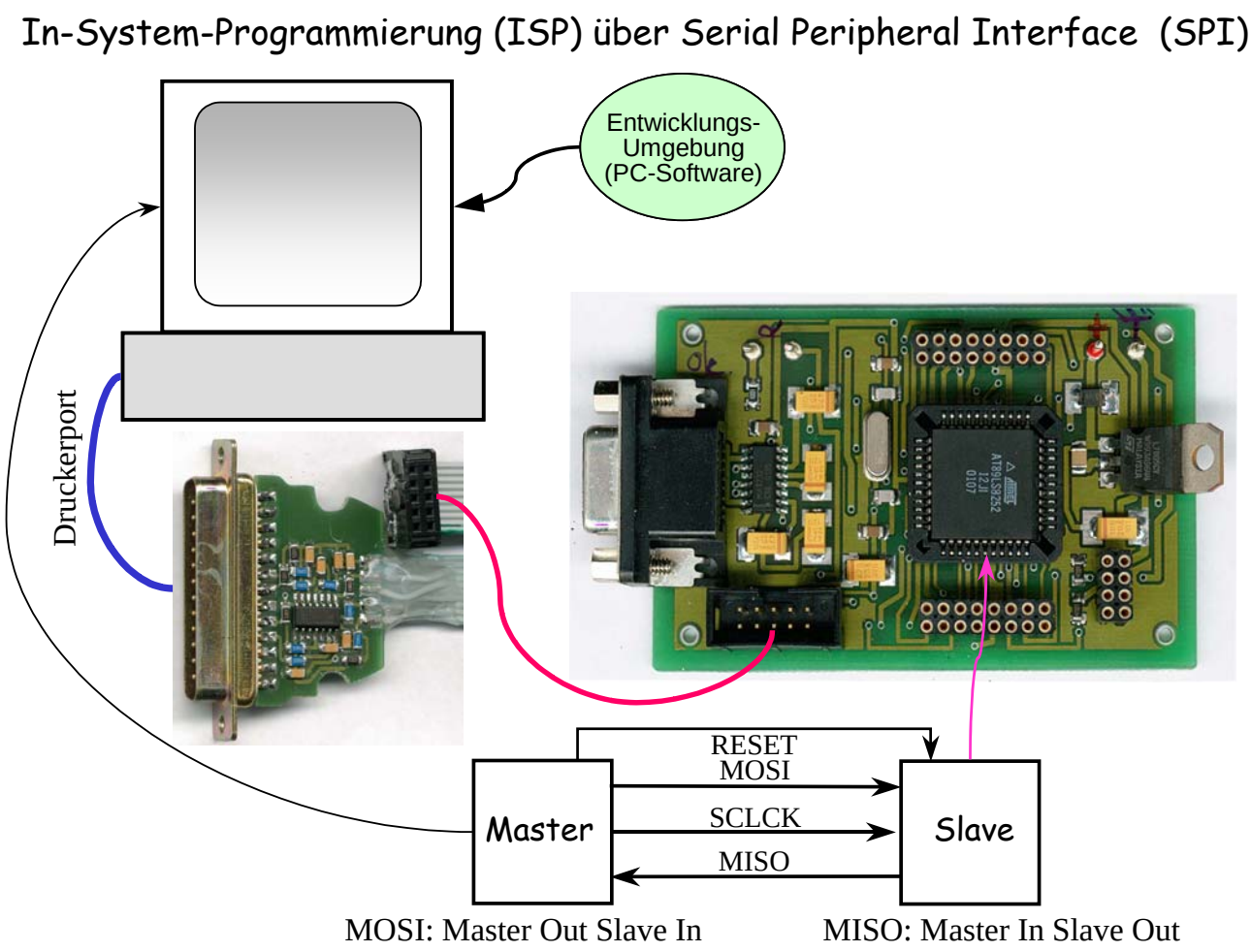



Bild 4.102: In-System Programmierung über PC-Druckerschnittstelle

Die Mehrzahl der heutigen Mikrorechner mit integriertem Flash-Programmspeicher (EEPROM) wird in der in Bild 4.102 dargestellten Weise programmiert. Der Mikrorechner enthält seine „Programmierfirmware“ die den gesamten Zeitablauf des Programmiervorganges einschließlich der nötigen Spannungserhöhung zur Ladung und Entladung der Floating-Gates steuert. Die Stecker des SPI sind genormt, d.h. stets so belegt wie CON1 in Bild 4.103 zeigt.

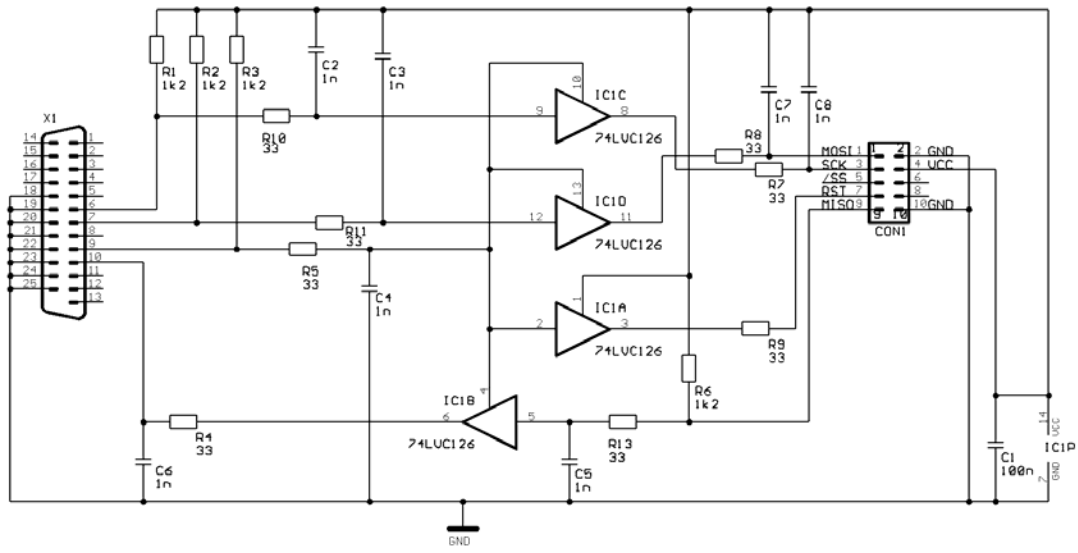



Bild 4.103: Schaltbild des SPI ⇔ Drucker-Interfaces

Außer einem Reset-Signal zur Initialisierung stehen zwei Datenleitungen und eine Taktleitung zur Verfügung. Über MOSI (Master Out Slave In) werden Adressen und OP-Code seriell in den

Mikrorechner transferiert, während über MISO (Master In Save Out) die Verifikation der in den Programmspeicher eingeschriebenen Information erfolgt.

Eine für den Anwender noch einfachere Art der Programmübertragung in den Speicher eines Mikrorechners ist in Bild 4.104 dargestellt. Hier wird die „normale“ serielle Schnittstelle benutzt. Außer dem V.24-Pegelwandler (hier ADM 202) wird zwischen Mirkocontroller und PC nur ein Standard Verbindungskabel für den seriellen Port benötigt. Der Programmiervorgang wird eingeleitet, wenn der Pin PSEN – wie in Bild 4.104 gezeigt – über eine Widerstand von 1kΩ an Masse gelegt ist und dann die Stromversorgung eingeschaltet wird. Vom PC kann jetzt das Programm übertragen werden (Download). Nach Beendigung der Übertragung wird PSEN freigeschaltet, d.h. offengelassen. Wenn jetzt ein Reset durchgeführt wird - in der Regel durch Aus- und Einschalten der Stromversorgung - startet das eingeschriebene Programm.

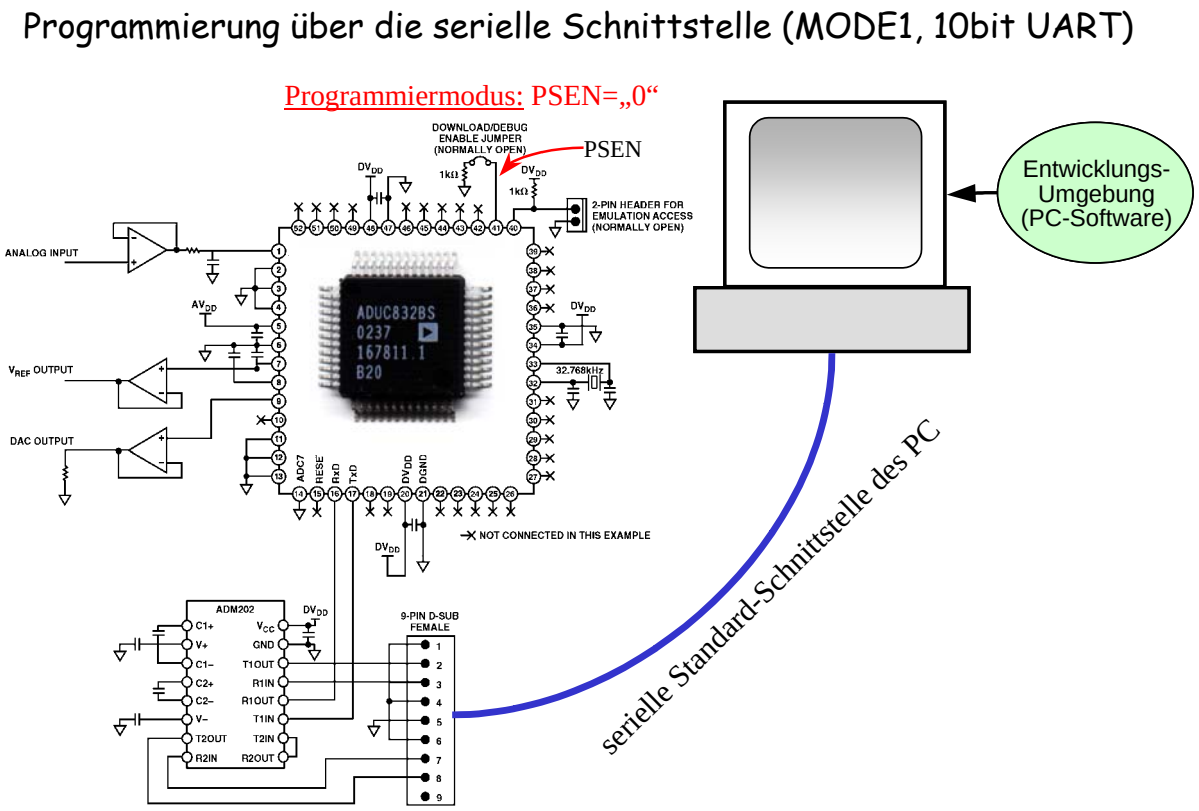



Bild 4.104: In-System-Programmierung über die serielle Standard-Schnittstelle

## Beispiel einer PC-basierten Entwicklungsumgebung

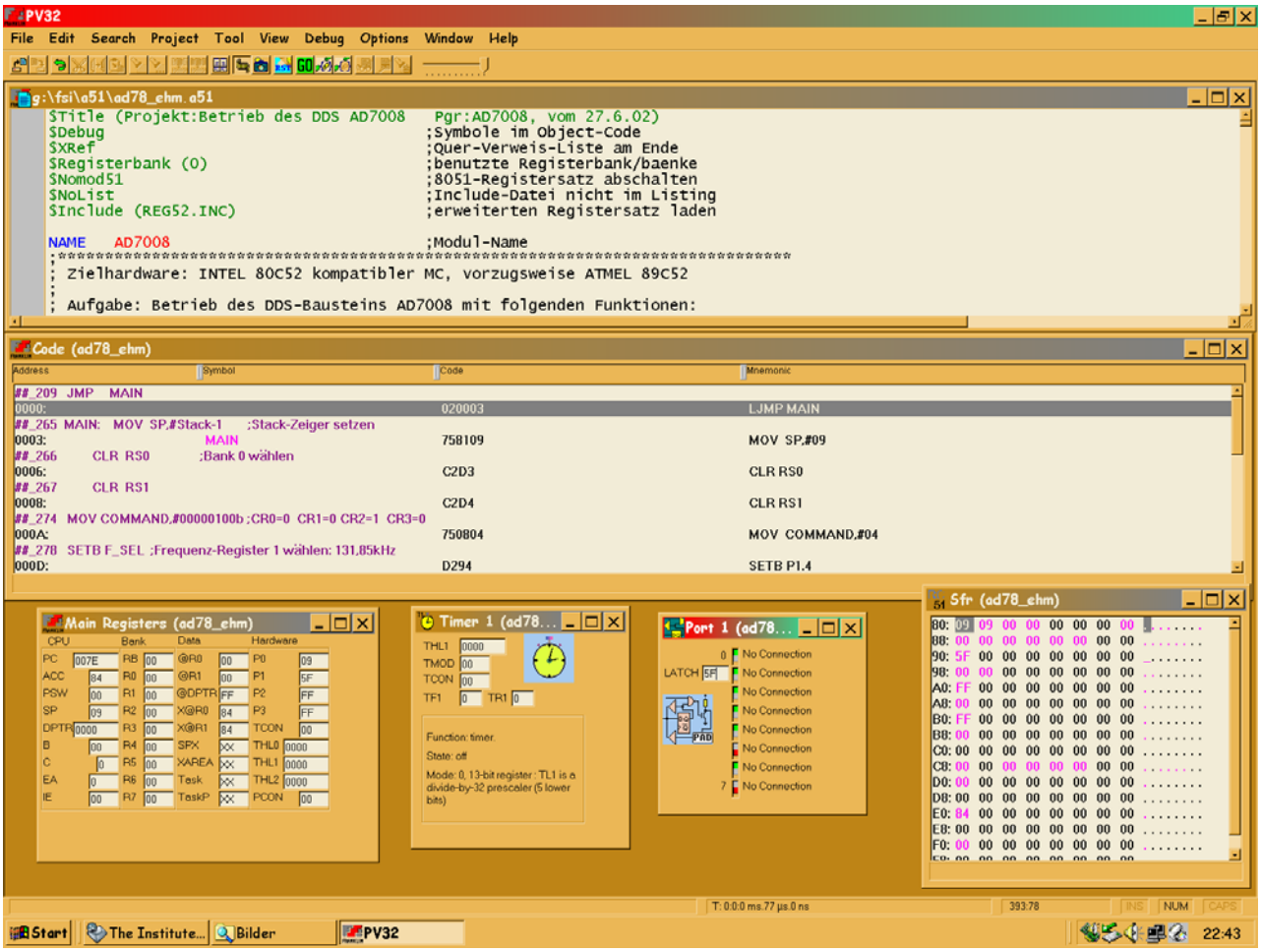



Bild 4.105: Vollständige Mikrocontroller–Entwicklungsumgebung

## 4.9 Digitaler Signalprozessor (DSP) am Beispiel: Motorola DSP56002

### 4.9.1 Die Mikrorechnerkonzepte CISC-RISC

Bis etwa 1986 war die Fachwelt von einer kaum noch zu übertreffenden Leistungsfähigkeit konventioneller Mikroprozessoren wie dem 68020 oder dem 80386 überzeugt. Diese Prozessoren basieren auf der Philosophie komplexer Befehlssätze (**CISC** ≡ **C**omplex **I**nstruction **S**et **C**omputer), deren Abarbeitung relativ große integrierte Mikroprogramme verlangt. Dafür sollte das **CISC**-Konzept dem Anwender durch entsprechend „mächtige“ Befehle die Programmierarbeit erheblich erleichtern.

Nach und nach kamen aber ganz andere Vorstellungen über optimale Mikroprozessorarchitekturen auf, die auf wesentlich vereinfachte Designs mit weniger Transistoren hinausliefen. Dabei waren Leistungssteigerungen bis zum Faktor fünf gegenüber herkömmlichen CISC-Maschinen im Gespräch. Die neuartigen Maschinen basierten auf drastisch vereinfachten Befehlssätzen und wurden als **R**educed **I**nstruction **S**et **C**omputer (**RISC**) eingeführt.

Umfangreiche Analysen verschiedenartiger Computerprogramme hatten im Vorfeld gezeigt, daß rund 80% der Befehle nur 20% des verfügbaren Befehlssatzes einer typischen CISC-Maschine nutzten. Weitere Untersuchungen führten zu der Erkenntnis, daß die am häufigsten angewandten Instruktionen meist einfache Operationen ausführten und auch die einfacheren Adressierungsarten benutzten. Würde man ausschließlich solche Befehle in besonderen Hardware-Architekturen beschleunigen, könnte man mit einer erheblichen Steigerung der Leistungsfähigkeit rechnen. Die Untersuchungen hatten klar bewiesen, daß sich der bisherige Aufwand in Richtung komplexer Befehlen nicht gelohnt hatte und das Ziel, die Belastung eines Programmierers oder den Compileraufwand zu verringern, nicht erreicht werden konnte.

#### Es müssen auch Hardwareaspekte berücksichtigt werden:

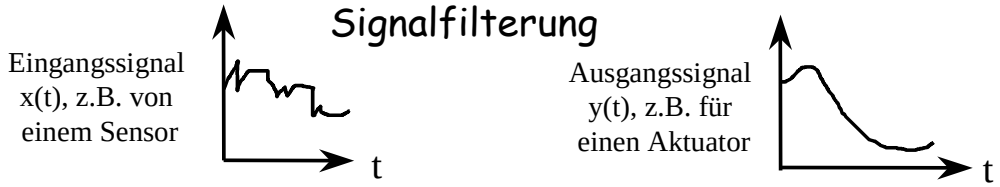
Wenn ein Mikrorechner nur einfache Befehle braucht, kann die Komplexität der zugehörigen Hardware reduziert werden. Daraus folgt, daß mit weniger Transistoren sogar höherer Leistung möglich wird. Der einfache Befehlssatz gestattet die Abarbeitung eines Befehls in einem einzigen Taktschritt. Komplexe Operationen müssen dann zwar aus einer Sequenz einfacher Befehle zu konstruiert werden, wobei aber gegenüber einer fest konfigurierten CISC-Maschine ein weites Feld für Optimierungen verbleibt.

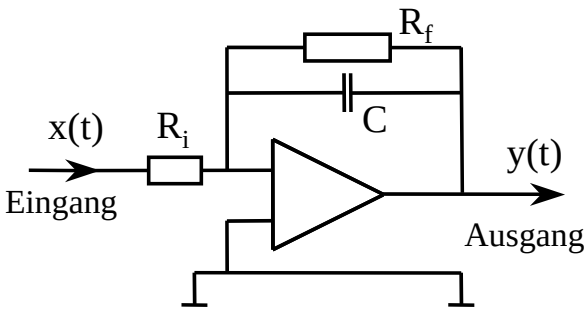
#### Was steckt hinter der „RISC-Philosophie“

- **Alle Befehle laufen in einem einzigen Taktschritt ab.**
  Der OP-Code muß eine feste Länge haben, die kleiner oder höchstens gleich der Breite der Datenbusse ist. Zusätzliche Operanden dürfen nicht vorkommen. Die Befehlsdecodierung muß direkt und einfach sein.
- **Speicherzugriffe erfolgen nur bei Lade- und Abspeicherbefehlen**.
  Wenn ein Befehl Speichermanipulationen durchführen muß, sind zwangsläufig mehrere Arbeitsschritte nötig. Ein CISC-Prozessor lädt dazu die Speicherdaten in ein Register, das Register wird manipuliert, und sein Inhalt wird anschließend wieder in den Speicher geschrieben. Eine solche Sequenz benötigt mindestens drei Taktschritte. Ein RISC-Prozessor hingegen bearbeitet Daten typischerweise direkt in Registern und beläßt sie dort. Daraus folgt, daß beim RISC-Konzept immer eine große Anzahl von Registern benötigt wird.
- Alle Einheiten zur Befehlsausführung sind **„festverdrahtet“**, d.h. es wird **kein Mikroprogramm** implementiert. Es ist praktisch **unmöglich**, Mikroprogrammierung einzusetzen, wenn man die komplette Befehlsausführung in **einem** Taktschritt verlangt.

### 4.9.2 Mikrorechner zur digitalen Signalverarbeitung mit Echtzeitanforderungen

Auf den ersten Blick denkt man beim Begriff „digitale Signalverarbeitung“ wahrscheinlich an die Abarbeitung komplizierter mathematischer Formeln, wie sie schon bei recht einfachen Aufgaben wie der Filterung von Signalen vorkommen. Der digitalen Signalverarbeitung haftet der Ruf abschreckender mathematischer Komplexität an. Eine Ursache dafür ist, daß der „normale“ Elektroingenieur in der Vergangenheit im Rahmen seiner Ausbildung und Berufspraxis mit analogen Filtern und deren recht einfacher mathematischer Beschreibung vertraut war.




Analoge Filterschaltung (Tiefpass)

## Mathematischer Zusammenhang für die analoge Signalverarbeitung

Ansatz mit komplexen Amplituden:

$$\frac{\underline{Y}}{\underline{X}} = -\frac{R_f}{R_i}\cdot\left[\frac{1}{1+ j\omega R_f \cdot C}\right], \text{ mit } y(t) = \mathrm{Re}\left\{\underline{Y}\cdot e^{j\omega t}\right\}, x(t) = \mathrm{Re}\left\{\underline{X}\cdot e^{j\omega t}\right\}$$

## Gesamtaufbau bei digitaler Signalverarbeitung

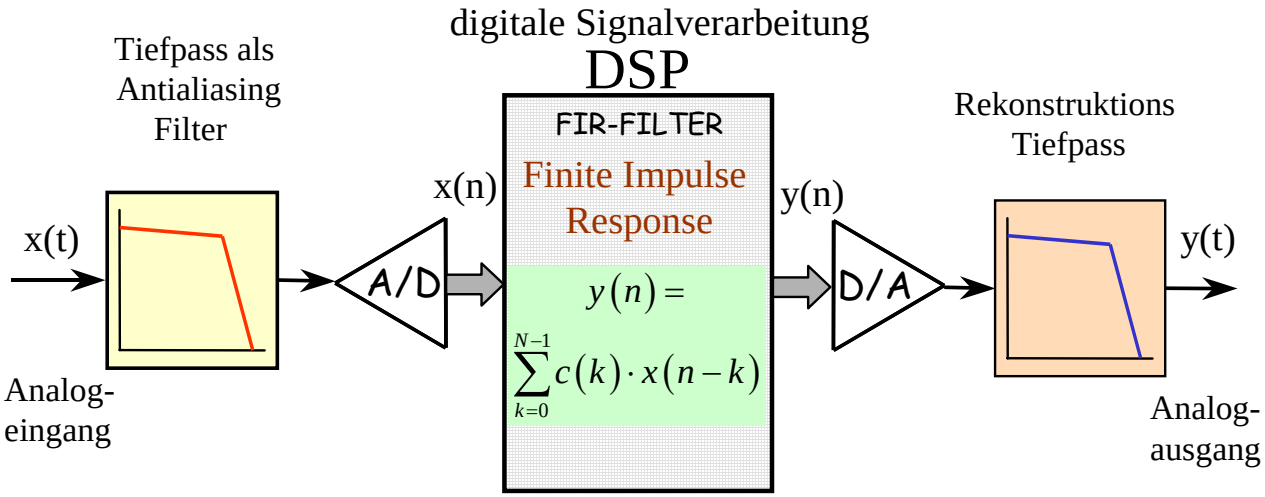



Bild 4.106: Gegenüberstellung analoge und digitale Signalverarbeitung

So ist z.B. ein analoges Tiefpaßfilter, bestehend aus einem Operationsverstärker, zwei Widerständen und einem Kondensator anhand einer äußerst einfachen Formel in seiner Funktion zu verstehen. Der Entwurf ist leicht und die Realisierung sehr kostengünstig – wie Bild 4.106 zeigt.

Das digitale Äquivalent hingegen benötigt mehrere elektronische Stufen zur Wandlung des analogen Eingangssignals in digitale Eingangsdaten, zur Verarbeitung der Daten und zur Rekonstruktion eines in der Regel benötigten analogen Ausgangssignals. Die Datenverarbeitung umfaßt meist eine Vielzahl digitaler Multiplikationen und Addition (MAC-Operationen), die nur mit umfangreicher Hardware in der benötigten Geschwindigkeit verfügbar sind.

Wo liegen unter diesen Aspekten eigentlich die Vorteile digitaler Signalverarbeitung?

- Die digitale Signalverarbeitung (DSV) leidet nicht unter Drift- und Alterungseffekten der Komponenten. Parameterstreuungen, die bei der Serienherstellung analoger Bauteile unvermeidlich sind, müssen in Analogschaltungen durch oft aufwendige Abgleichmaßnahmen unschädlich gemacht werden. Bei Digitalschaltungen zeigt bereits der Chiptest, der in der Halbleiterfertigung noch vor dem Gehäuseeinbau durchgeführt wird, ob der Baustein alle Anforderungen auch hinsichtlich der vorgesehenen Arbeitsgeschwindigkeit (Taktfrequenz) fehlerfrei erfüllt. Es gibt nichts abzugleichen und weder Drift noch Alterung spielen im Rahmen der üblichen Bauteillebensdauer unter „Normalbedingungen“ eine Rolle.
- Digitalbausteine haben von Natur aus eine hohe **Störresistenz**, denn alle rauschähnlichen Störungen, die betragsmäßig unterhalb der Schaltschwelle (bei CMOS z.B. 2,5V) bleiben, werden komplett unterdrückt. Auch nicht perfekt „gesiebte“ Versorgungsspannungen verursachen in Digitalschaltungen keine Probleme.
- Des weiteren kann in einem DSP auf einfache Weise die Möglichkeit eines Selbsttests implementiert werden. So kann ohne Ausbau aus der Schaltung in regelmäßigen Abständen die Funktionstüchtigkeit überprüft werden. Das ist nicht nur in sicherheitsrelevanten Anwendungen ein unschätzbarer Pluspunkt. Zudem sind z.B. bei Filteranwendungen die Filterkoeffizienten und damit die Filtercharakteristiken jederzeit änderbar. Es kann sogar zwischen völlig verschiedenen Funktionen per Software „umgeschaltet“ werden. Ein digitales Filter kann damit adaptiv auf Änderungen von Signalparametern wie z.B. Amplitude oder spektrale Zusammensetzung reagieren.

Die Schlüsseloperation der DSV wird durch die Abkürzung **MAC** (**M**ultiply and **Ac**cumulate) charakterisiert. **MAC**-Operationen sind Kernstücke der Faltung, der Korrelation sowie der Fourier-, Laplace- und auch der Wavelet-Transformation.

### 4.9.3 Zur Bedeutung der MAC-Operation

An dieser Stelle werden ohne weitere Erläuterung einige Beispiele von Algorithmen der digitalen Signalverarbeitung angeführt, bei denen **Multiplizier-Akkumulier-Operationen** das Kernstück bilden.

1. Diskrete Korrelation

$$C_{xy}(k) = \sum_{i=0}^{N-1} x(i) \cdot y(k+i) \tag{4.34}$$

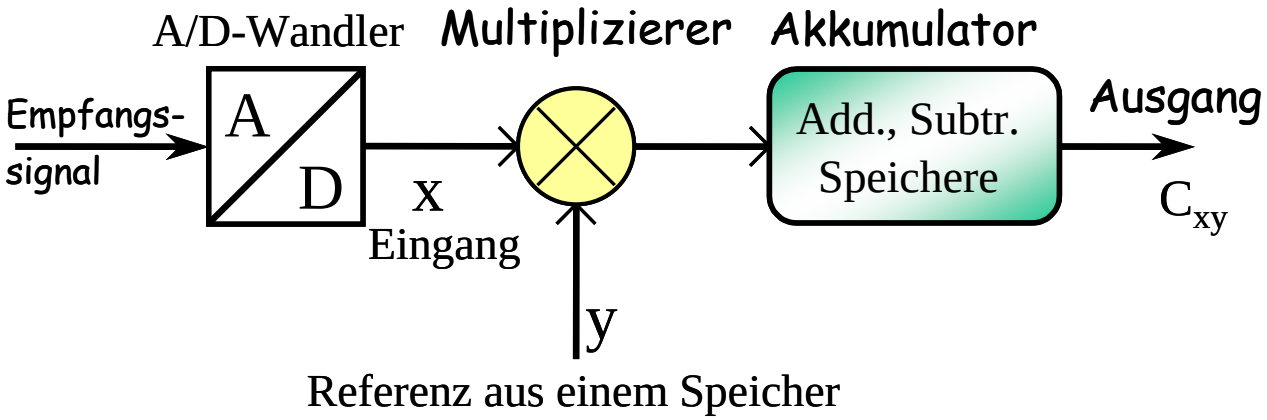



Bild 4.107: Grundaufbau eines digitalen Korrelationsempfängers

Im dargestellten Beispiel repräsentiert *x(k)* die digitalisierten Abtastwerte eines Empfangssignals, das mit der Abtastfrequenz $f_A=1/T$ abgetastet wurde, während die *y(k)* Referenzwerte aus einem Speicher sind. Zur Vereinfachung der Schreibweise ist es üblich, die Größe *T* (Abtastzeitraster) wegzulassen und statt *(kT)* oder *(iT)* nur *(k)* bzw. *(i)* zu schreiben. Man muß sich jetzt aber stets darüber klar sein das k, i nicht mehr dimensionslose Zählvariablen sind, sondern die Dimension „Zeit“ haben. Bei der diskreten Fouriertransformation wird uns Ähnliches bei der Darstellung des Spektrums begegnen. Hier muß man dann beachten das einer Zählvariablen wie z.B. *n* in Wirklichkeit eine Frequenz zugeordnet ist.

$$\begin{array}{lll} C'_{xy}(k) = x(\nu) \cdot y(\nu+k) & + & \sum_{i=0}^{\nu-1} x(i) \cdot y(i+k) \\ \updownarrow \qquad \updownarrow & & \updownarrow \\ S_\nu = \quad x \cdot y & + & S_{\nu\text{-}1} \end{array} \tag{4.35}$$

Anhand von (4.35) lassen sich die Schritte der MAC-Operation in einem Aufbau nach Bild 4.107 im Detail verfolgen. Der Akkumulator speichert jeweils die bislang aufgelaufene Summe $S_{\nu-1}$ zu der mit jedem Abtastwert ein neu berechnetes Produkt XY addiert wird. Nach *N* Abtastwerten ist das „Korrelationsergebnis“ für eine „Signalform“ der Dauer $N \cdot T$ komplett und kann in weiteren Funktionseinheiten eines digitalen Empfängers ausgewertet werden. In der Praxis wird die Bitbreite des Akkumulators meist deutlich größer als die des Produktes gewählt. Eine sinnvolle Konstellation bei einer A/D-Wandlerauflösung von 8bit (d.h. Produktbreite 16bit) ist z.B. eine Akkulänge von 32bit, so daß sich mit (4.11) und (4.35)

$$S_\nu = -2^{31} + 2^8 + \sum_{i=15}^{30} 2^i + x_7 y_7 \cdot 2^{14} + \sum_{i=0}^{6}\sum_{j=0}^{6} x_i y_j \cdot 2^{i+j} + \sum_{i=0}^{6} \overline{x_7 y_i} \cdot 2^{7+i} + \sum_{i=0}^{6} \overline{y_7 x_i} \cdot 2^{7+i} - s_{(\nu-1)31} \cdot 2^{31} + \sum_{i=0}^{30} s_{(\nu-1)i} \cdot 2^i$$

ergibt.

2. Diskrete Faltung

$$F_{xy}(k) = \sum_{i=0}^{N-1} x(i) \cdot y(k-i) \tag{4.36}$$

3. Diskrete Fouriertransformation (DFT)

$$G\left(\frac{n}{NT}\right) = \sum_{i=0}^{N-1} g(kT) \cdot e^{-j2\pi nk/N}, \quad \text{mit } n = 0,1...N-1 \tag{4.37}$$

4. Inverse diskrete Fouriertransformation (IDFT)

$$g(kT) = \frac{1}{N} \sum_{n=0}^{N-1} G\left(\frac{n}{NT}\right) \cdot e^{j2\pi nk/N}, \text{ mit } k = 0,1,...N-1 \tag{4.38}$$

Die Algorithmen der DFT und der IDFT werden hier ohne weitere Erläuterung nur hingeschrieben um die Bedeutung der MAC–Operation zu veranschaulichen. Die Argumente *(kT)* und *(n/NT)* sind vollständig ausgeschrieben, um klarzumachen, daß es sich bei *k* und *n* in einer vereinfachten Schreibweise nicht mehr um reine Zahlen sondern um dimensionsbehaftete Größen, nämlich Zeit bzw. Frequenz handelt.

5. Schnelle Fouriertransformation (engl.: fast FT $\Rightarrow$ FFT)

An dieser Stelle können Algorithmen der FFT noch nicht behandelt werden. Es geht lediglich wieder darum die Bedeutung der MAC–Operationen bei der FFT zu verdeutlichen. Ausgehend von (4.37) wird in der vereinfachten Schreibweise ein Eingangsvektor *g(k)* der Länge *N* im Zeitbereich angesetzt, was einem Signalausschnitt der Dauer $N \cdot T$ entspricht. Als Ergebnis erhält man ebenfalls einen Vektor der Länge *N* im Frequenzbereich, der die Spektrallinien im Abstand *1/NT* darstellt. Insgesamt wird also ein Spektrum der Breite $f_A = 1/T$ beschrieben.

$$G(n) = \sum_{k=0}^{N-1} g(k) \cdot e^{-j2\pi nk/N}, \quad \text{mit } n = 0,1...N-1 \tag{4.39}$$

Man sieht, daß (4.39) komplex ist, so das jede Multiplikation in Wirklichkeit zweimal, nämlich für Real- und Imaginärteil auszuführen ist. Führt man die Abkürzung

$W = e^{-j\frac{2\pi}{N}}$ ein, dann sind darin alle festen Größen enthalten und aus (4.39) wird

$$G(n) = \sum_{k=0}^{N-1} g(k) \cdot W^{nk}, \quad \text{mit } n = 0,1...N-1 \tag{4.40}$$

Die Größe $W^{nk}$ wird auch Drehfaktor (engl.: twiddle factor) genannt und spielt in der Abarbeitung von FFT-Algorithmen eine wichtige Rolle. Für die Berechnung wird $W^{nk}$ in Real- und Imaginärteil zerlegt.

$$W^{nk} = e^{-j2\pi\frac{nk}{N}} = \cos\left(2\pi\frac{nk}{N}\right) - j\sin\left(2\pi\frac{nk}{N}\right), \quad \text{mit} \quad k,n \in \{0,1,...N-1\} \tag{4.41}$$

Man sieht, daß auch bei reellen Zeitfunktionen die DFT in der Regel komplex ist. Die komplette direkte Ausrechnung von (4.40) bedeutet somit eine zweifache *NxN*-Matrix-Multiplikation, d.h. es fallen $2N^2$ Multiplikationen an. Die besondere Leistung von FFT-Algorithmen besteht nun darin, die Zahl der Multiplikationen durch geschickte Ausnutzung von Redundanzen erheblich zu reduzieren, nämlich auf $\boldsymbol{N \cdot ld(N)}$. Bei den gebräuchlichsten FFT-Algorithmen ist *N* eine Zweierpotenz, so

daß sich z.B. bei $N=2^{10}=1024$ eine Reduktion von $2^{21}=2.097.152$ Multiplikationen auf $2^{10}\cdot 10=10.240$ ergibt.

6. Quadrieren

   Neben der Multiplikation wird in der DSV, insbesondere in Kommunikationssystemen auch gelegentlich das Quadrieren benötigt, z.B. bei der sogenannten „geometrischen" Addition $z=\sqrt{x^2+y^2}$. Um das Berechnen der Wurzel wollen wir uns an dieser Stelle nicht kümmern.

Wir betrachten eine $N$-stellige Binärzahl in Zweierkomplementdarstellung

$$A=-a_n\cdot 2^{N-1}+\sum_{i=0}^{N-2}a_i\cdot 2^i \tag{4.42}$$

Mit Hilfe von (4.11) erhalten wir, in dem wir dort $B=A$ setzen und unter Beachtung der Tatsache, daß das boolesche Produkt gleicher Binärvariabeln $\boldsymbol{a_i\cdot a_i=a_i}$ ist, sofort das Ergebnis:

$$A^2=-2^{2N-1}+a_{N-1}\cdot 2^{2N-2}+2^N+\sum_{i=0}^{N-2}\overline{a_{N-1}\cdot a_i}\cdot 2^{i+N}+\sum_{i=0}^{N-2}\sum_{j=0}^{N-2}a_i\cdot a_j\cdot 2^{i+j} \tag{4.43}$$

Für $N=17$ soll die Richtigkeit von (4.43) beispielhaft verdeutlicht werden:

$$A^2=-2^{33}+a_{16}\cdot 2^{32}+2^{17}+\sum_{i=0}^{15}\overline{a_{16}\cdot a_i}\cdot 2^{i+17}+\sum_{i=0}^{15}\sum_{j=0}^{15}a_i\cdot a_j\cdot 2^{i+j} \tag{4.44}$$

Man erhält stets korrekte positive 32-stellige Binärzahlen als Ergebnis, wenn der Übertrag in die Stelle $2^{34}$ ignoriert wird. Dazu betrachten wir zunächst positive Zahlen, d.h. $a_{16}=0$

$$A^2=-2^{33}+2^{17}+\sum_{i=0}^{15}1\cdot 2^{i+17}+\sum_{i=0}^{15}\sum_{j=0}^{15}a_i\cdot a_j\cdot 2^{i+j}=-2^{33}+2^{17}+\left[2^{32}+2^{31}+2^{30}+...+2^{17}\right]+\sum_{i=0}^{15}\sum_{j=0}^{15}a_i\cdot a_j\cdot 2^{i+j}$$

$$=-2^{33}+2^{33}+\sum_{i=0}^{15}\sum_{j=0}^{15}a_i\cdot a_j\cdot 2^{i+j}=\sum_{i=0}^{15}\sum_{j=0}^{15}a_i\cdot a_j\cdot 2^{i+j}$$

Man sieht, daß das Ergebnis immer positiv ist und sich aus der Doppelsumme über die Binärvariablen $a_i\cdot a_j$ ergibt. Ein Quadrierer kann im Vergleich zum Multiplizierer vorteilhafter in Hardware realisiert werden, wenn man dabei ausnützt, daß beide Operanden gleich sind.

Jetzt untersuchen wir negative Zahlen, d.h. $a_{16}=1$

$$A^2=-2^{33}+2^{32}+2^{17}+\sum_{i=0}^{15}\overline{1\cdot a_i}\cdot 2^{i+17}+\sum_{i=0}^{15}\sum_{j=0}^{15}a_i\cdot a_j\cdot 2^{i+j}$$

$$=-2^{33}+2^{32}+2^{17}+\left\{\overline{a_{15}}\cdot 2^{32}+\overline{a_{14}}\cdot 2^{31}...+\overline{a_0}\cdot 2^{17}\right\}+\sum_{i=0}^{15}\sum_{j=0}^{15}a_i\cdot a_j\cdot 2^{i+j}$$

Man sieht , daß die Summe in der geschweiften Klammer maximal den Wert $\{2^{33}\text{-}2^{17}\}$ und minimal den Wert Null annehmen kann, so daß das maximale Ergebnis

$$A^2=-2^{33}+2^{32}+2^{17}+2^{33}-2^{17}+\sum_{i=0}^{15}\sum_{j=0}^{15}a_i\cdot a_j\cdot 2^{i+j}=2^{32}+\sum_{i=0}^{15}\sum_{j=0}^{15}a_i\cdot a_j\cdot 2^{i+j}$$

sein wird. Dafür muß natürlich gelten $a_i=0$ für $i=0..15$, so daß einfach $A^2=2^{32}$ ist.

Wenn der Inhalt der geschweiften Klammer Null ist, hat man

$$A^2=-2^{33}+2^{32}+2^{17}+\sum_{i=0}^{15}\sum_{j=0}^{15}a_i\cdot a_j\cdot 2^{i+j}=-2^{33}+2^{32}+2^{17}+\sum_{i=0}^{15}\sum_{j=0}^{15}a_i\cdot a_j\cdot 2^{i+j}\,.$$

Es darf natürlich kein negatives Ergebnis beim Quadrieren entstehen. Immer wenn die geschweifte Klammer Null ergibt, muß $a_i$=1 sein für $i$=0..15. Dann liefert

$$\sum_{i=0}^{15}\sum_{j=0}^{15} a_i \cdot a_j \cdot 2^{i+j} = \left(2^{16}-1\right)^2 = 2^{32}-2^{17}+1 \Rightarrow A^2 = -2^{33}+2^{32}+2^{17}+2^{32}-2^{17}+1, \text{ also } A^2 = +1$$

Zwei extreme Zahlenbeispiele sollen den Sachverhalt noch mal abschließend illustrieren:

Beispiel 1: $A$=–1

$$a_i = 1 \text{ für } i = 0...15, \Rightarrow A^2 = \left(-2^{16}+2^{16}-1\right)^2 = 1$$

$$A^2 = -2^{33}+2^{32}+2^{17}+\{0\}+\left\{2^{32}-2^{17}+1\right\} = 1$$

Beispiel 2: A=$-2^{16}$

$$a_i = 0 \text{ für } i = 0...15, \Rightarrow A^2 = \left(-2^{16}\right)^2 = 2^{32}$$

$$A^2 = -2^{33}+2^{32}+2^{17}+\left\{2^{33}-2^{17}\right\}+\{0\} = 2^{32}$$

7. Digitale Filterung

Anhand von (4.34) wurde die mathematische Beschreibung der diskreten Korrelation eingeführt. Die Rechenleistung, die für eine derartige, im Grunde recht einfache Operation erforderlich ist, kann enorm sein, wenn hohe Arbeitsgeschwindigkeit für „Echtzeitbetrieb“ gefordert wird. In einem Kommunikationssystem muß der Empfänger das ankommende Empfangssignal „lückenlos“ verarbeiten können. Im Fall des Korrelators muß somit die MAC-Geschwindigkeit der Abtastrate des A/D-Wandlers entsprechen.

Noch besser als am Korrelatorbeispiel lassen sich die Grenzen der Leistungsfähigkeit eines DSP an digitalen Filtern aufzeigen. Dies wird im folgenden verdeutlicht, ohne daß ein Verständnis der Filtertheorie erforderlich ist.

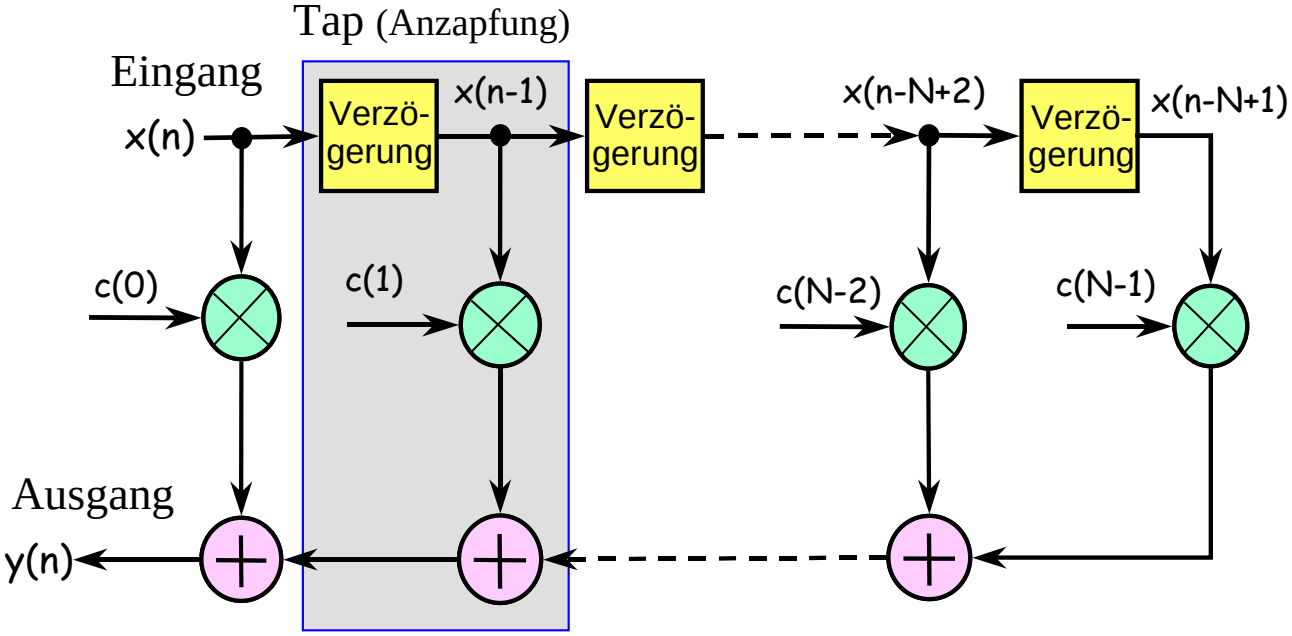



Bild 4.108: Struktur eines digitalen FIR-Filters (Finite Impulse Response)

Die Eigenschaften eines Digitalfilters nach Bild 4.108 werden durch die Zahl $N$ der Anzapfungen (Taps) und die Filterkoeffizienten $c(i)$ bestimmt.

Grundsätzlich werden in einem digitalen Filter *N* Abtastwerte des Eingangssignals *x(n)* gespeichert. Die Verzögerungen sind in einfacher Weise mit Flip-Flops zu realisieren, deren Taktfrequenz durch die Abtastrate vorgegeben ist. Zur Berechnung **eines** Ausgangswertes *y(n)* müssen mit jedem Abtasttakt **alle gespeicherten** Eingangswerte mit den zugehörigen Filterkoeffizienten *c(i)* multipliziert und alle Ergebnisse der Multiplikationen müssen aufsummiert werden. Formelmäßig bedeutet dies

$$y(n) = \sum_{k=0}^{N-1} c(k) \cdot x(n-k). \tag{4.45}$$

Obwohl (4.45) der Gleichung (4.34) sehr ähnlich sieht, gibt es einen beträchtlichen Unterschied. Denn während im Fall des Korrelators die MAC-Geschwindigkeit lediglich der Abtastrate des A/D-Wandlers entsprechen muß, erfordert das Auftreten eines jeden neuen Abtastwertes *x(n)* aus dem A/D-Wandler in Bild 4.106 die komplette Berechnung der Summe nach (4.45), d.h. *N* Multiplikationen und *N* Akkumulationen.

Ein typisches Kennzeichen digitaler Filter ist eine hohe Anzahl von Taps, z.B. 100 und mehr. Da ein gewöhnlicher DSP aber nur eine MAC-Operation pro Maschinenzyklus (≡ Taktzyklus) ausführt, würde bei einer Abtastrate von 1MHz (Grenzfrequenz des zu verarbeitenden Signals $f_g$< 500kHz) für $N$=100 bereits eine Rechenleistung von $100 \cdot 10^6$ Instruktionen pro Sekunde, also 100 MIPS benötigt. Moderne DSPs bieten – bei realistischer Betrachtungsweise – maximal etwa 1000 MIPS. Es ist deshalb nicht verwunderlich, daß man beim Bau von DSP höchste Anstrengungen unternimmt um die MAC-Geschwindigkeit zu optimieren. Ein wichtiger Ansatz dazu ist die Implementierung effizienter Multiplikationsalgorithmen in Hardware, so daß eine möglichst parallele Abarbeitung erzielt wird. Im folgenden Abschnitt werden Hardwarestrukturen zur schnellen Parallelmultiplikation detailliert betrachtet.

### 4.9.4 Parallelmultiplizierer

#### Vorzeichenlose Integermultiplikation

$$X \cdot Y = \left( \sum_{j=0}^{N-1} x_j 2^j \right) \cdot \left( \sum_{i=0}^{N-1} y_i 2^i \right) = \sum_{j=0}^{N-1} \sum_{i=0}^{N-1} x_i \cdot y_j \cdot 2^{i+j}$$

für ein Beispiel mit $N$=8 ergibt sich

$$X \cdot Y = \sum_{j=0}^{7} \sum_{i=0}^{7} x_i \cdot y_j \cdot 2^{i+j} \tag{4.46}$$

Die Doppelsumme kann wie folgt in einem Rechenschema dargestellt werden:

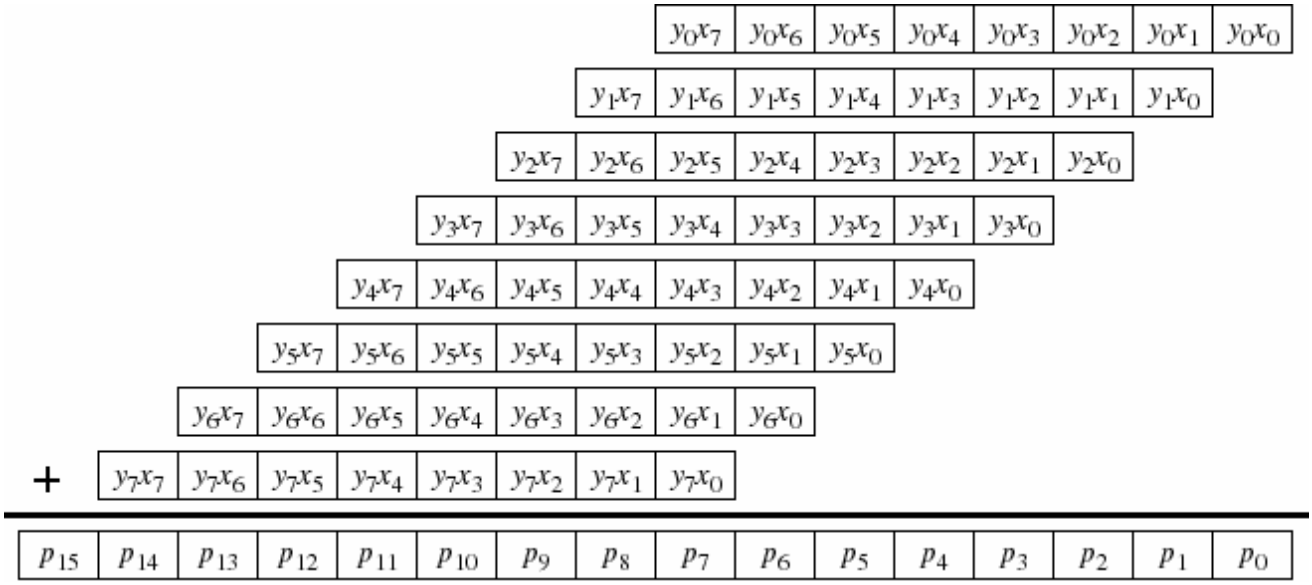



Bild 4.109: Berechnung des Produkts zweier vorzeichenloser 8-bit-Integerzahlen

Das in Bild 4.109 dargestellte Schema läßt sich direkt auf eine Hardware, bestehend aus UND-Gattern für die Binärproduktbildung $x_i \cdot y_j$ und Volladdierer mit entsprechender Carry-Behandlung abbilden. Auf diese Weise entsteht eine Art „Ripple-Carry-Multiplizierer“.

Die eingerahmten, mit UND-Gattern erzeugten Binärprodukte $x_i \cdot y_j$, werden den im folgenden Bild 4.110 dargestellten Addiererketten zugeführt. Jede Box (mit +) stellt einen Volladdierer dar, der jeweils oben die Eingänge für die Binärvariablen und rechts für Carry hat. Nach unten wird die Summe ausgegeben und nach links der Übertrag in die nächsthöhere Zweierpotenz. Wie man sieht, hat man hier siebenmal den 8-bit-Ripple-Carry-Addierer nach Bild 4.15 vor sich. Obwohl alle 56 Volladdierer parallel vorhanden sind, können sie nicht wirklich parallel arbeiten, da die „unteren“ Addierer auf die Ergebnisse der oberen warten müssen. Das Carrybit $p_{15}$ kann somit im ungünstigsten Fall vom Carry-Ausgang des ersten Volladdierers (rechts oben in Bild 4.110) beeinflußt werden. Dann erhält man unter der Annahme, daß die Volladdierer, unabhängig vom jeweils benutzen Eingang, stets die gleiche Verzögerung $\tau_{VA}$ aufweisen, insgesamt eine Laufzeit von $20 \cdot \tau_{VA}$. Da ein Volladdierer auch unter Nutzung der vorteilhaften Transfergatetechnologie noch mindestens 4 Gatterlaufzeiten aufweist, ist die dargestellte Lösung hinsichtlich der Rechengeschwindigkeit nicht befriedigend wie das folgende Beispiel zeigt: Unter Annahme einer Gatterlaufzeit von 1ns könnte der Multiplizier höchsten mit einer Frequenz von 12,5 MHz arbeiten – und das bei „nur“ 8bit Wortlänge.

Wie man Bild 4.110 entnehmen kann, wächst die Laufzeit entsprechend mit der Bitbreite, d.h. für jedes weitere Bit kommt in jeder Zeile ein VA hinzu und unten muß eine weitere ganze Zeile eingefügt werden. Dadurch wächst die Laufzeit pro weiteres Bit um $3\tau_{VA}$. Bei 16bit würde die Geschwindigkeit nach dem obigen Beispiel somit auf 1GHz/(20+24)·4=5,68 MHz absinken, obwohl 240 Volladdierer parallel vorhanden sind.

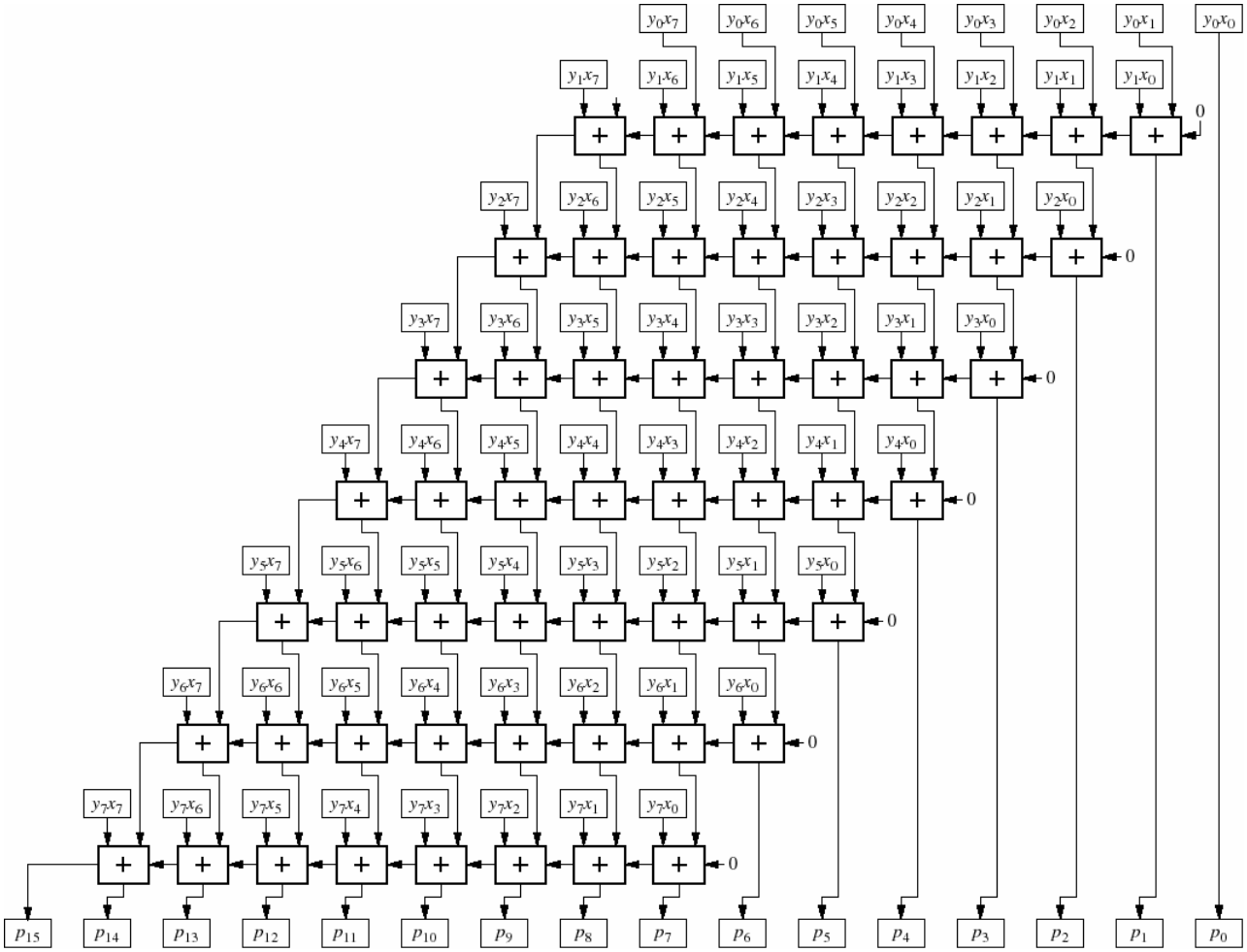


Bild 4.110: 8-bit-Multiplizierer für vorzeichenlose Ganzzahlen mit Ripple-Carry-Addition

## Parallelmultiplikation vorzeichenbehafteter Zweierkomplementzahlen

Aus den obigen Betrachtung wird klar, daß es notwendig ist, Überlegungen zum Aufbau effizienterer Architekturen anzustellen. Zudem muß auch das Rechnen mit vorzeichenbehafteten Zahlen einbezogen werden. Dazu wurde bereits das Produkt von zwei vorzeichenbehafteten Binärzahlen in Zweierkomplementdarstellung

$$A \cdot B = \left( -a_{N-1} \cdot 2^{N-1} + \sum_{i=0}^{N-2} a_i 2^i \right) \cdot \left( -b_{N-1} \cdot 2^{N-1} + \sum_{i=0}^{N-2} b_i 2^i \right) \qquad (4.47)$$

so umgeformt, daß bis auf das höchstwertige Bit nur noch vorzeichenlose Binärvariablen bzw. Zweierpotenzen vorkamen – s. Gleichung (4.11):

$$A \cdot B = -2^{2N-1} + 2^N + a_{N-1} \cdot b_{N-1} \cdot 2^{2N-2} + \sum_{j=0}^{N-2} \sum_{i=0}^{N-2} a_i \cdot b_j \cdot 2^{i+j} +$$
$$+ \sum_{i=0}^{N-2} \overline{a_{N-1} b_i} \cdot 2^{i+N-1} + \sum_{i=0}^{N-2} \overline{b_{N-1} a_i} \cdot 2^{i+N-1} \qquad (4.48)$$

Des weiteren wurde ein Beispiel für die Multiplikation von zwei 8-bit-Zahlen *A* und *B* angegeben, mit einem 16 bit langen Produkt als Ergebnis:

$$P = A \cdot B = -2^{15} + 2^8 + a_7 \cdot b_7 \cdot 2^{14} + \sum_{j=0}^{6} \sum_{i=0}^{6} a_i \cdot b_j \cdot 2^{i+j}$$
$$+ \sum_{i=0}^{6} \overline{a_7 b_i} \cdot 2^{i+7} + \sum_{i=0}^{6} \overline{b_7 a_i} \cdot 2^{i+7} \qquad (4.49)$$

In Kapitel 3 wurde auch schon festgestellt, daß die Produktbildung der Komponenten $a_i \cdot b_j$ mittels UND-Gattern und die der negierten Komponenten $\overline{b_i \cdot a_j}$ mittels NAND-Gattern durchgeführt werden kann. Wenn man die einzelnen Binärprodukte nach ihrer Wertigkeit ordnet, entsteht das im folgenden Bild 4.111 dargestellte Matrixschema, das im Vergleich mit Bild 4.110 (vorzeichenlose Multiplikation) etwas unregelmäßiger aussieht. Zur Bestimmung des Endergebnisses wird jetzt ein ganz anderer Weg beschritten, der die langsame Ripple-Carry-Addition vermeidet, d.h. es erfolgt keine zeilenweise Addition mehr.

In einem ersten Schritt wird – wie Bild 4.112 zeigt - die gesamte Matrix möglichst vollständig mit Halb- und Volladdierern „bedeckt“. Man erkennt, daß in diesem Beispiel 6 mit X gekennzeichnete Bitstellen im ersten Schritt unbearbeitet bleiben. Die für Schritt 1 dargestellten Halb- und Volladdierer berechnen jetzt parallel die Summen- und Übertragsbits, die sich aus der jeweiligen Eingangsbelegung ergeben, d.h. die maximale Verzögerung für Schritt 1 ist genau eine Volladdiererlaufzeit, obwohl 16 Volladdierer und 6 Halbaddierer vorhanden sind. Die Ergebnisse sind jetzt Summenbits $s_i$, die an der jeweiligen Position (wegen gleicher Binärwertigkeit) verbleiben, bzw. Übertragsbits $c_{i+1}$, die eine Position nach links zur nächsthöheren Zweierpotenz weitergegeben werden. Auf diese Weise entsteht die für Schritt 2 dargestellte reduzierte Matrix mit nur noch 6 statt 9 Zeilen. In gleicher Weise wird jetzt diese Matrix auch wieder möglichst vollständig mit Halb- und Volladdierern „bedeckt“.

# Schema zur schnellen parallelen Multiplikation

Beispiel: 8bit Zweierkomplementzahlen (Multiplikation mit Vorzeichen)

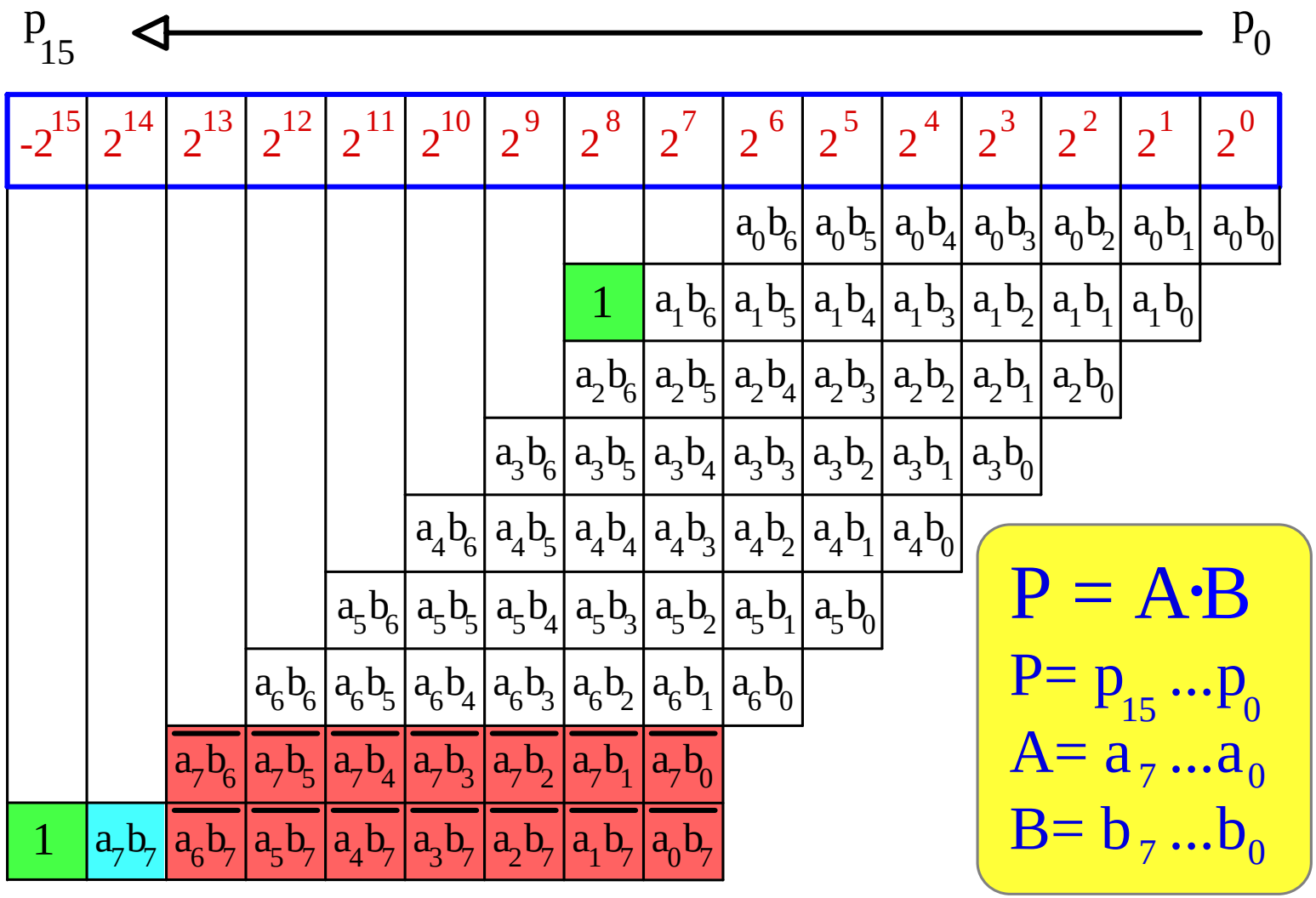



Bild 4.111: Aufstellung eines Rechenschemas gemäß Gleichung (4.49)

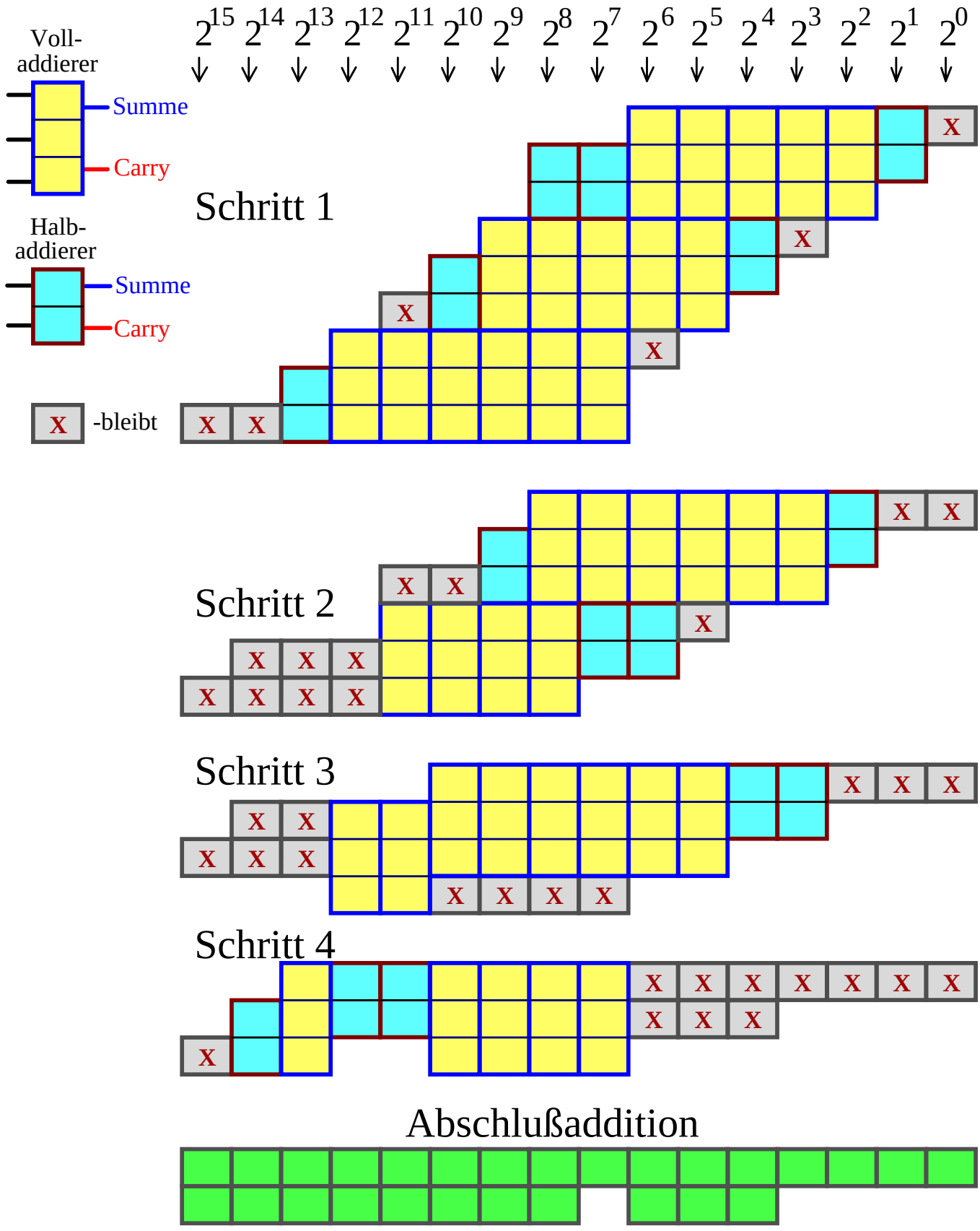



Bild 4.112: Schematische Darstellung der Abarbeitung der Matrix aus Bild 4.111 mit Hilfe von 15 Halb- und 39 Volladdierern in 4 Schritten

Im dargestellten Beispiel werden dazu 10 Volladdierer und 4 Halbaddierer eingesetzt, die nun wiederum parallel die Ergebnisse aus Schritt 1 bearbeiten. Nach spätestens einer Volladdiererlaufzeit stehen auch hier Ergebnisse in Form von Summen- und Übertragsbits zur Verfügung, die in gleicher Weise wie oben beschrieben in Schritt 3 eingebracht werden. Wie man sieht ist die Matrix jetzt auf 4 Zeilen reduziert.

Mit 8 Voll- und 2 Halbaddierern erfolgt eine weitere Reduzierung auf 3 Zeilen, so daß nach einem letzten Schritt 4 mit 5 Voll- und 3 Halbaddierern nur noch zwei Zeilen verbleiben. Man sieht, daß die unteren 4bit des Ergebnisses bereits vorliegen, so daß nur noch zwei 12 bit lange Vektoren zu addieren sind. Hierzu kann z.B. ein kaskadierter Carry-Look-Ahead-Addierer, bestehend aus drei 4-bit-Bausteinen, verwendet werden.

Die Rechenzeit beträgt dann insgesamt 4 Volladdiererlaufzeiten zuzüglich der Zeit, die der Abschußaddierer benötigt. Wird dieser auf Basis von Bild 4.21 realisiert, dann weist er maximal 9 Gatterlaufzeiten auf, was in etwa zwei Volladdiererlaufzeiten entspricht. Die gesamte Rechenzeit beläuft sich somit auf 6 Volladdiererlaufzeiten zuzüglich einer UND- bzw. NAND-Gatterverzögerung zur Binärproduktbildung.

Im Vergleich mit Bild 4.110 wird der Vorteil der neuen Architektur, die in der Literatur auch als „Wallace-Tree“ bezeichnet wird, deutlich: Die Verzögerung ist von 20 auf 6 $\tau_{VA}$ reduziert. Zudem kommt man mit 39 Voll- und 15 Halbaddierern aus, während in Bild 4.110 immerhin 56 Volladdierer benötigt werden. Wichtig ist auch, daß mit der neuen Architektur nach Bild 4.112 vorzeichenbehaftete Zahlen multipliziert werden können.

Mit dem Aufbau nach Bild 4.112 sind die Möglichkeiten zur Erhöhung der Rechengeschwindigkeit aber noch keineswegs ausgeschöpft. Durch die Anwendung von Pipelining kann eine weitere, deutliche Steigerung erreicht werden. Die Realisierung erfolgt so, daß die Ausgangssignale der Addierer eines jeden Schrittes in Bild 4.112 nicht unmittelbar mit den Eingängen der Addierer des nächsten Schrittes verbunden werden, sondern einem getakteten Register zur Zwischenspeicherung zugeführt werden. Es sind also 4 Register mit jeweils an die Matrixgröße angepaßter Länge nötig, d.h. 50 bit nach Schritt 1 und 27 bit nach Schritt 4. Obwohl jetzt noch eine Flip-Flop-Setup-Zeit hinzukommt, ergibt sich dennoch eine bedeutende Geschwindigkeitssteigerung, weil in jedem Schritt neue Daten bearbeitet werden können. Denn nachdem z.B. die Ergebnisse aus Schritt 1 ins Register 1 übernommen wurden, sind die Addierer von Schritt 1 frei und können sofort mit neuen Eingangswerten für das nächste zu berechnende Produkt belegt werden. Inzwischen verarbeiten die Addierer in Schritt 2 die in Register 1 zwischengespeicherten Werte weiter und liefern ihre Ergebnisse an Register 2. Man sieht, daß nach 4 Takten nur noch die Abschußaddition erfolgen muß und die Pipeline gefüllt ist, so daß mit jedem Takt ein neues vollständiges Ergebnis geliefert wird. Entscheidend ist, daß sich die Durchlaufzeiten der einzelnen Schritte jetzt nicht mehr addieren, sondern nur noch der „langsamste“ Schritt die Geschwindigkeit begrenzt. Im obigen Beispiel würde der kaskadierte Carry-Look-Ahead-Addierer aus drei 4-bit-Bausteinen die größte Durchlaufzeit von ca. 2 $\tau_{VA}$ aufweisen, so daß sich eine Geschwindigkeitssteigerung um etwa den Faktor 3 durch das Pipelining ergäbe. Natürlich kann der Abschlußaddierer auch noch schneller gemacht werden, indem man keine Kaskadierung von 4bit-Einheiten verwendet, sondern das Carry-Look-Ahead Prinzip durchgehend realisiert. Der Hardwareaufwand kann dadurch beträchtlich wachsen, aber die Durchlaufzeit kann aber bis auf drei Gatterlaufzeiten reduziert werden. Im obigen Beispiel würde das schon keinen Sinn mehr machen, weil ein Volladdierer 4 Gatterlaufzeiten aufweist und somit die maximale Geschwindigkeit bestimmen würde.

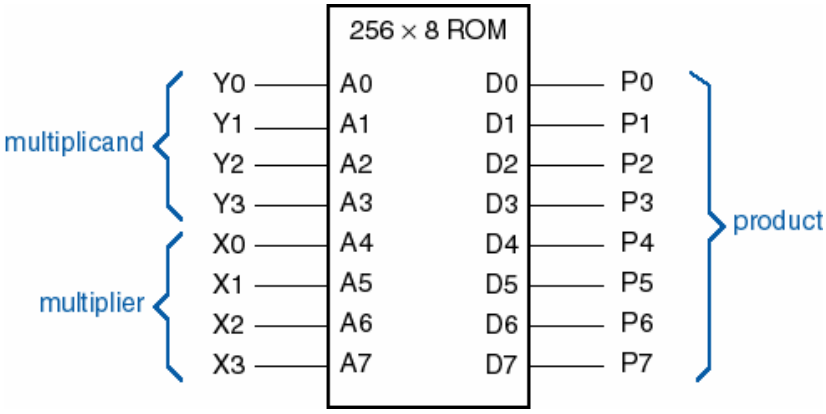



Bild 4.113: Ein 256-Byte-ROM als Multiplizierer

Wie Bild 4.113 und Bild 4.114 zeigen, läßt sich mit einem Festwertspeicher in einfacher Weise ein sehr schneller Multiplizierer realisieren. Der Speicher (ROM, PROM, EPROM, EEPROM) hat eine 8bit-Adresse und speichert 8bit breite Daten. Wenn man nun z.B. die unteren 4 Adreßbits (A0...A3) als Multiplikand (*Y*) und die oberen 4 (A4...A7) als Multiplikator (*X*) auffaßt, dann sind genau 256 Kombinationen für das Produkt möglich, die an den zugehörigen Speicherplätzen abgelegt werden müssen. Die Rechenzeit ist die maximale Zugriffzeit auf eine Speicherstelle.

Es ist sofort einzusehen, daß diese Methode, einen Multiplizierer zu bauen, auf **kleine Wortbreiten** beschränkt ist. Wollte man nämlich einen **32x32bit** Multiplizierer auf diese Weise realisieren, wäre ein Baustein mit 64 Adreßbits und einer Kapazität von $2^{64}$ (das wären **mehr als** $18 \cdot 10^{18}$) Worten zu je 64 bit erforderlich.

```
00: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
10: 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
20: 00 02 04 06 08 0A 0C 0E 10 12 14 16 18 1A 1C 1E
30: 00 03 06 09 0C 0F 12 15 18 1B 1E 21 24 27 2A 2D
40: 00 04 08 0C 10 14 18 1C 20 24 28 2C 30 34 38 3C
50: 00 05 0A 0F 14 19 1E 23 28 2D 32 37 3C 41 46 4B
60: 00 06 0C 12 18 1E 24 2A 30 36 3C 42 48 4E 54 5A
70: 00 07 0E 15 1C 23 2A 31 38 3F 46 4D 54 5B 62 69
80: 00 08 10 18 20 28 30 38 40 48 50 58 60 68 70 78
90: 00 09 12 1B 24 2D 36 3F 48 51 5A 63 6C 75 7E 87
A0: 00 0A 14 1E 28 32 3C 46 50 5A 64 6E 78 82 8C 96
B0: 00 0B 16 21 2C 37 42 4D 58 63 6E 79 84 8F 9A A5
C0: 00 0C 18 24 30 3C 48 54 60 6C 78 84 90 9C A8 B4
D0: 00 0D 1A 27 34 41 4E 5B 68 75 82 8F 9C A9 B6 C3
E0: 00 0E 1C 2A 38 46 54 62 70 7E 8C 9A A8 B6 C4 D2
F0: 00 0F 1E 2D 3C 4B 5A 69 78 87 96 A5 B4 C3 D2 E1
```

Bild 4.114: Alle vorkommenden Produkte sind im Speicher abgelegt

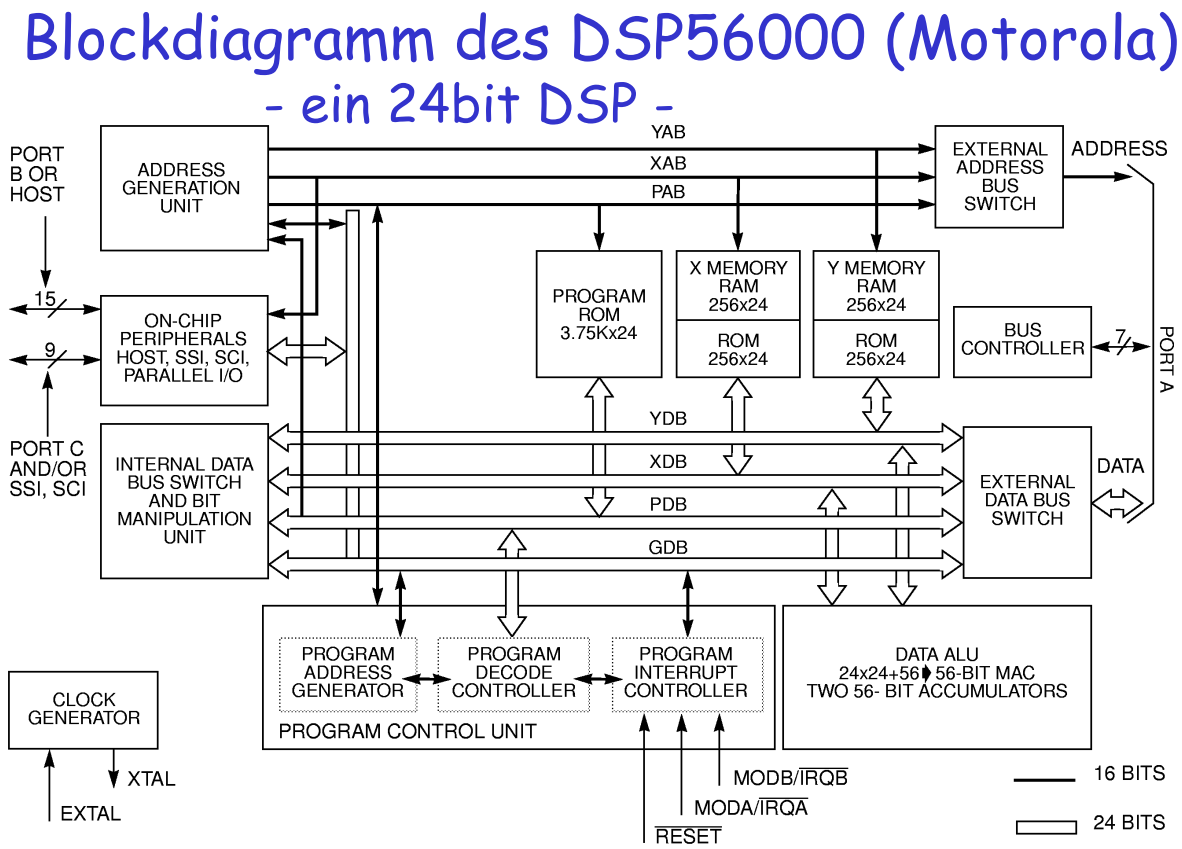



Bild 4.115: Der DSP 56000 im Überblick

Obwohl man DSV prinzipiell mit jedem Mikrorechner durchführen könnte, wären die Resultate meistens enttäuschend. Wegen der typischen Architektur von Standard-Mikroprozessoren oder Mikrocontrollern ließen sich höchstens Signalfrequenzen von nur wenigen hundert Hertz bewältigen. Für eine leistungsfähige DSP-Architektur werden daher, grundlegend neue Ansätze benötigt. Dabei werden andererseits aber bewährte Grundstrukturen nicht verworfen, sondern weiterentwickelt und an die neuen Aufgaben angepaßt. Die wesentlichen Charakteristika, die einen DSP ausmachen und ihn vom Mikroprozessor oder Mikrocontroller unterscheiden, werden im folgenden am Beispiel der DSP5600x-Famile des Herstellers Motorola betrachtet.

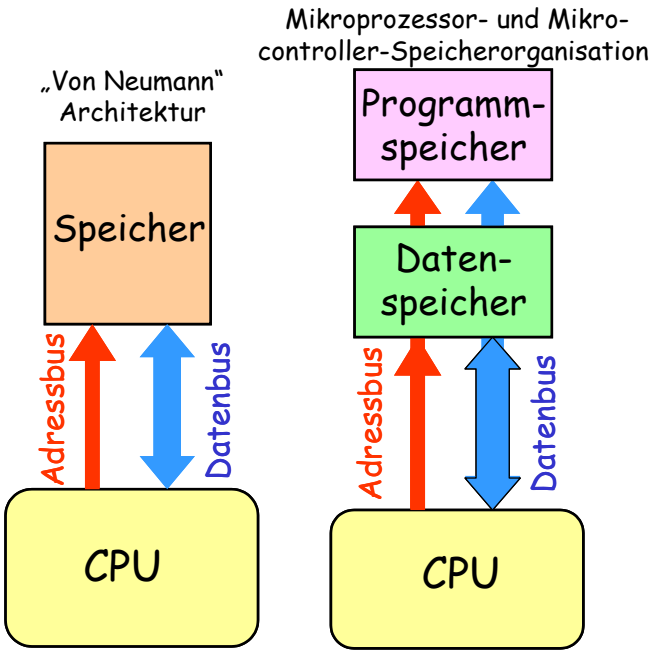



Bild 4.116:Mikroprozessor-Speicherorganisation

Der DSP56000 hat eine Datenwortlänge von 24 bit und verfügt über eine **Harvard**-Architektur mit **vier** separaten internen Bussen für Daten- und OP-Code-Transfer (GDB ≡ General-Purpose, z.B. für Bitmanipulationen; PDB ≡ Programm; XDB ≡ X-Datenspeicher; YDB ≡ Y-Datenspeicher). Drei separate interne **16bit-Adressbusse** dienen dem parallelen Zugriff auf Programmspeicher (PAB), X-Datenspeicher (XAB) und Y-Datenspeicher (YAB).

## Harvard-Architektur eines digitalen Signalprozessors

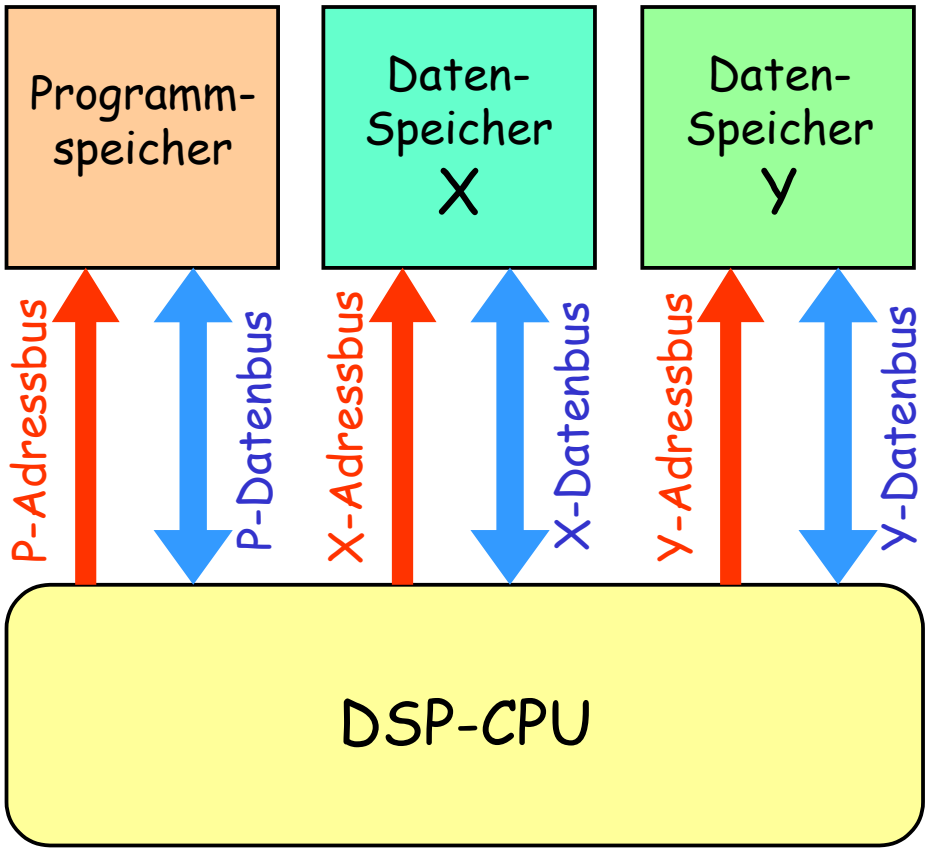



Bild 4.117: Typische Harvardarchitektur in einem DSP

Die Adreßbusse werden aus der *AGU* ≡ **A**ddress **G**eneration **U**nit gespeist, die in Bild 4.118 im Überblick dargestellt ist.

#### 4.9.4.1 Die AGU im DSP

In der AGU gibt es **drei Registergruppen**, die aus jeweils **8 Registern** aufgebaut sind.

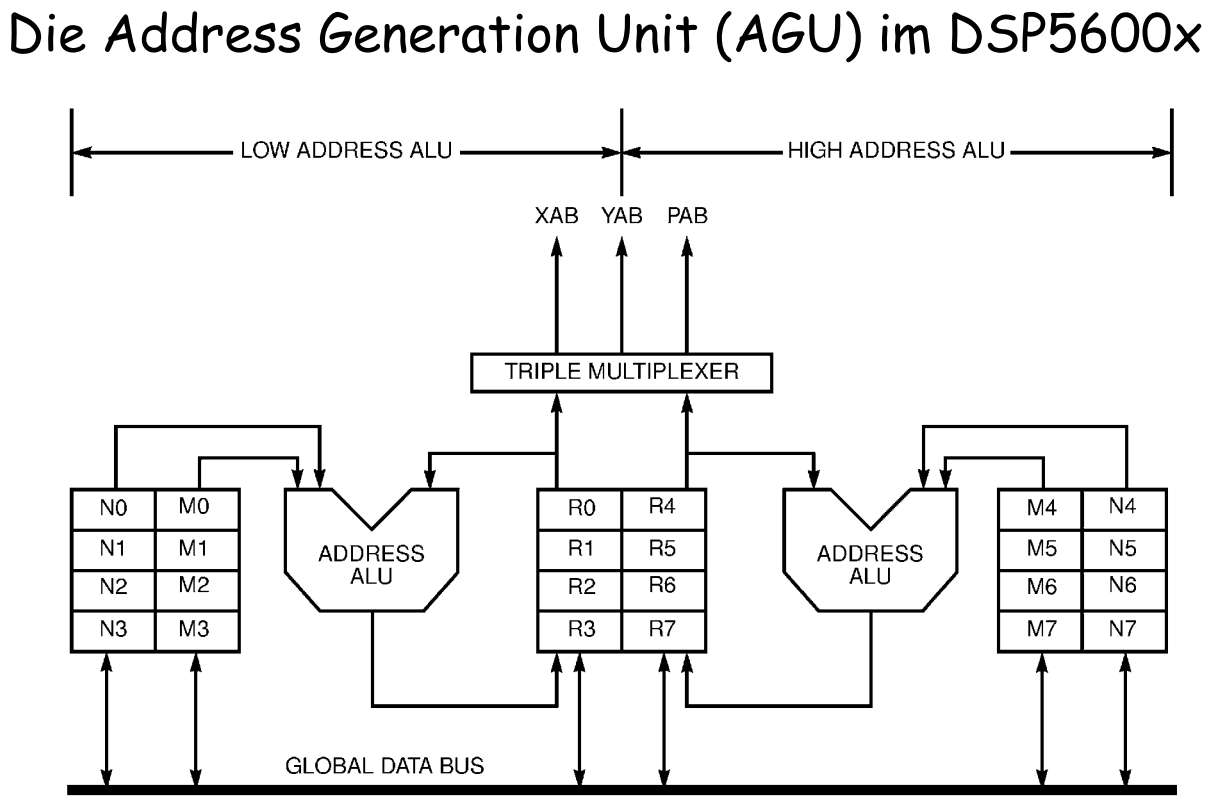



Bild 4.118: Aufbau der Adreßerzeugungseinheit im DSP 5600x

- Adreßzeiger-Register R0...R7,
- Adreß-Offsetregister N0...N7,
- Adreß-Modifikationsregister M0...M7.

Die AGU gestattet verschiedene Adressierungsarten für den Datenzugriff. Funktionell ist sie **zweigeteilt**, um gleichzeitig Datenzugriffe im X- und Y-Datenspeicherbereich zu ermöglichen. Bei der Adreßberechnung werden normale Zweierkomplement-Arithmetik, Modulo-Arithmetik und "Reverse-Carry"-Arithmetik unterschieden. Letztere findet ausschließlich bei der Ausführung von Algorithmen der FFT (Fast Fourier Transform) Anwendung.

Da gemäß Bild 4.118 zwei ALUs mit je 3 Volladdierern verfügbar sind, können in einem Maschinenzyklus zwei Adreßberechnungen stattfinden, und zwar für jede 2-er-Kombination aus den Adreßbussen XAB, YAB, PAB.

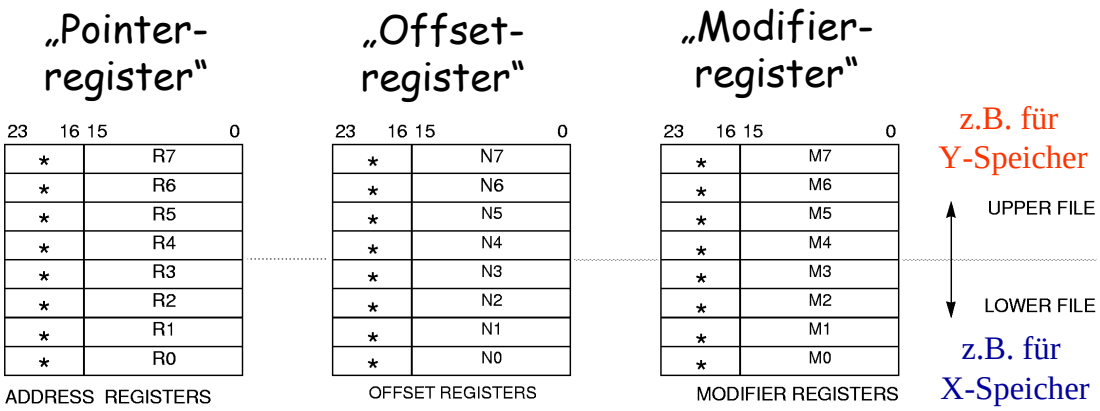



Bild 4.119: Das Programmiermodell der AGU

Es gibt folgende 3 Adressierungsarten

- Adreßregister direkt
- Adreßregister indirekt
- spezial

Beispiele für indirekte Registeradressierung

- $(R_n)$ indirekt ohne Pointermanipulation
- $(R_n)+$ Postinkrement
- $(R_n)-$ Postdekrement
- $-(R_n)$ Prädekrement
- $(R_n)+N_n$ Postinkrement mit Offset
- $(R_n)-N_n$ Postdekrement mit Offset
- $(R_n+N_n)$ „indexed" durch Offset: $EA$[23] <= Inhalt von $R_n+N_n$

Anwendungsbeispiele

**MOVE A1, X:(R0)**

bringe A1 an den Platz im X-Datenspeicher, der von R0 adressiert wird

R0 enthält X-Adresse, z.B. $1000
N0 beliebig
M0 $FFFF

[23] effektive Adresse

**MOVE B0, Y:(R1)+**

bringe B0 an den Platz im Y-Datenspeicher, der von R1 adressiert wird und erhöhe anschließend R1 um Eins (Postinkrement)

**entsprechend:MOVE B0,Y:(R1)–**

**MOVE X1,X:(R2)+N2**

Bringe zuerst X1 an den X-Speicherplatz, der von R2 adressiert wird. Dann wird R2 um den Inhalt von N2 erhöht und beim nächsten Aufruf der Instruktion kommt X1 an den X-Speicherplatz, der vom neuen Inhalt von $R2:=R2_{alt}+N2$ adressiert wird.

## Modulo-Funktionen zur Konstruktion von Ringspeichern

Der M-Registersatz bietet besondere Funktionen zum effizienten Aufbau von Schleifen in einem Datenspeicher. Hier wird die Modulo-Funktion untersucht.

| Register $M_n$ | **Funkion** |
|---|---|
| 0000 | Reverse-Carry ⇒ für FFT |
| 0001 | Modulo 2 |
| 0002 | Modulo 3 |
| m | Modulo (m+1) |
| ..... | ..... |
| 32.767 | Modulo 32.768 |
| ..... ..... | reserviert |
| 65.535 $FFFF | Modulo 65.536 (linear) ⇒ nach Reset |

Wenn ein Modifier-Register $M_n$ den Inhalt m hat, dann ist der „Modulus“ M=m+1.

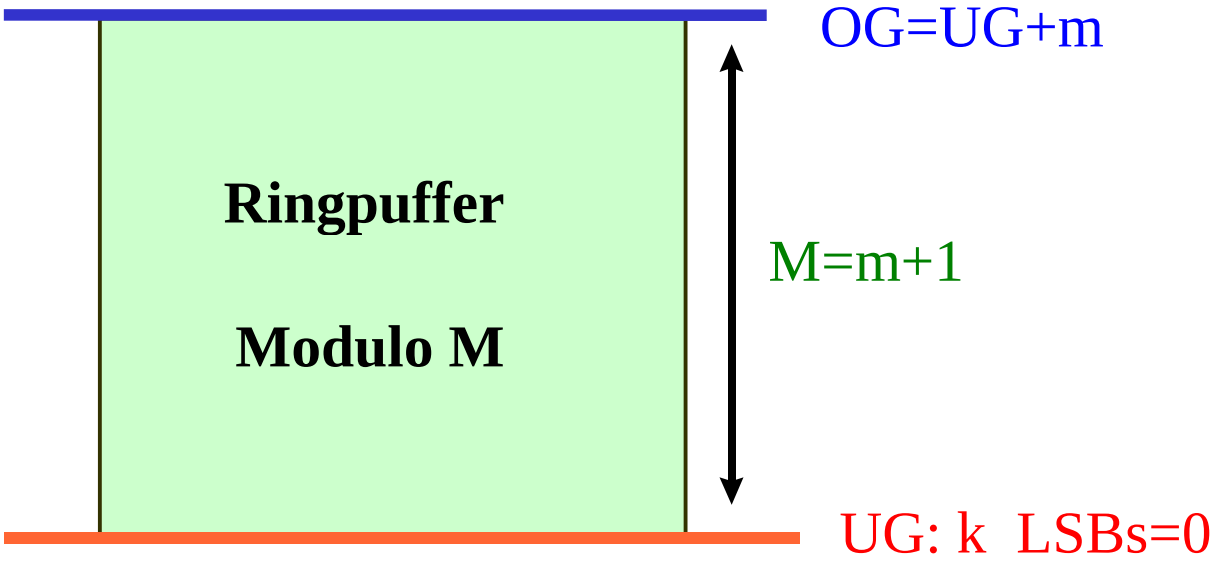



Bild 4.120: Implementierung eines Ringpuffers mit Hilfe der M-Register der AGU

Die Untergrenze UG ist nicht identisch mit den Pointer $R_n$, sondern erfordert Nullen in den unteren **k** Bits, wobei $2^k \geq M$ sein muß.

**Beispiel:** **M=21∈(16...32) ⇒ k=5 =>UG=xxxx00000**

Der Ringpuffer kann mit $(R_n) \pm 1$ durchlaufen werden, oder auch mit größerer Schrittweite, z.B. mit einem Offset $\pm N_n$.

Beispiel

```
MOVE   X0,X:(R2)+N2
M2=20  Modulus: M=21
N2=10  Offset
R2=75  Pointer
```

wegen ld{21} > 4 und <5 ist k = 5
Die Untergrenze hat also **mindestens** fünf Nullen in den LSBs.

Der Pointer R2 liegt im Bereich $2^6$=64....$2^7$=128

Die Modulo-Buffer-Untergrenze ist somit **64**.

Untergrenze UG: 64 → Obergrenze OG: 64+20 = 84

Ablauf: **X0→X:(75)**

**(R2)+N2** **X0→X:(85) geht nicht, da 85 > 84 außerhalb des Buffers ist**

Aktion: **Modulus wird abgezogen**

$R2_{neu}$: **R2+N2-(M2+1) => 75+10-(21)=64 (Untergrenze)**

weiter: **R2: 64+10 = 74**

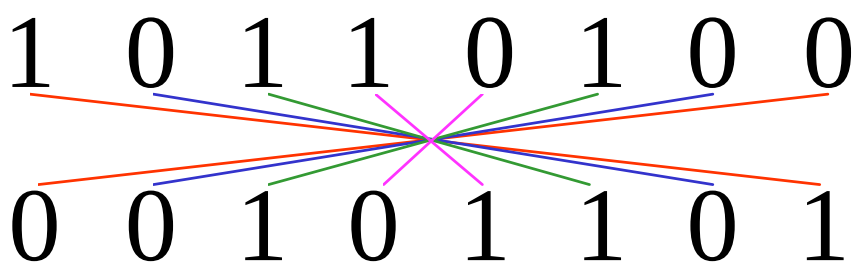
**R2: 74+10 = 84 (Obergrenze)**

$R2_{neu}$: **R2+N2-(M2+1) => 84+10-(21)=73**

Der Vollständigkeit halber, ohne weitere Erläuterung des Zwecks:

Bit-Reversing (Reverse-Carry-Arithmetik)

**$M_n$=$0000**



### 4.9.4.2 Die ALU im DSP

Die entscheidende Rechenleistung eines DSP wird in der Daten-ALU bereitgestellt. Im hier betrachteten Beispiel enthält sie zwei 56-bit-Akkumlatoren **A** und **B**. Beide Akkus sind aus drei Untereinheiten A0, A1 und A2 bzw. B0, B1 und B2 zusammengesetzt. A2 und B2 waren bei den ersten 56000-Bausteinen nur **8 bit** lange Erweiterungen. Bei den meisten Nachfolgetypen wurden jedoch hier ebenfalls **24 bit** implementiert. Bei einem 48-bit-Wort enthalten die Register A1 und B1 jeweils das **MSW** (≡ **M**ost **S**ignificant 24-bit-**W**ord). A0 und B0 enthalten entsprechend das **LSW** (≡ **L**east **S**ignificant 24-bit-**W**ord).

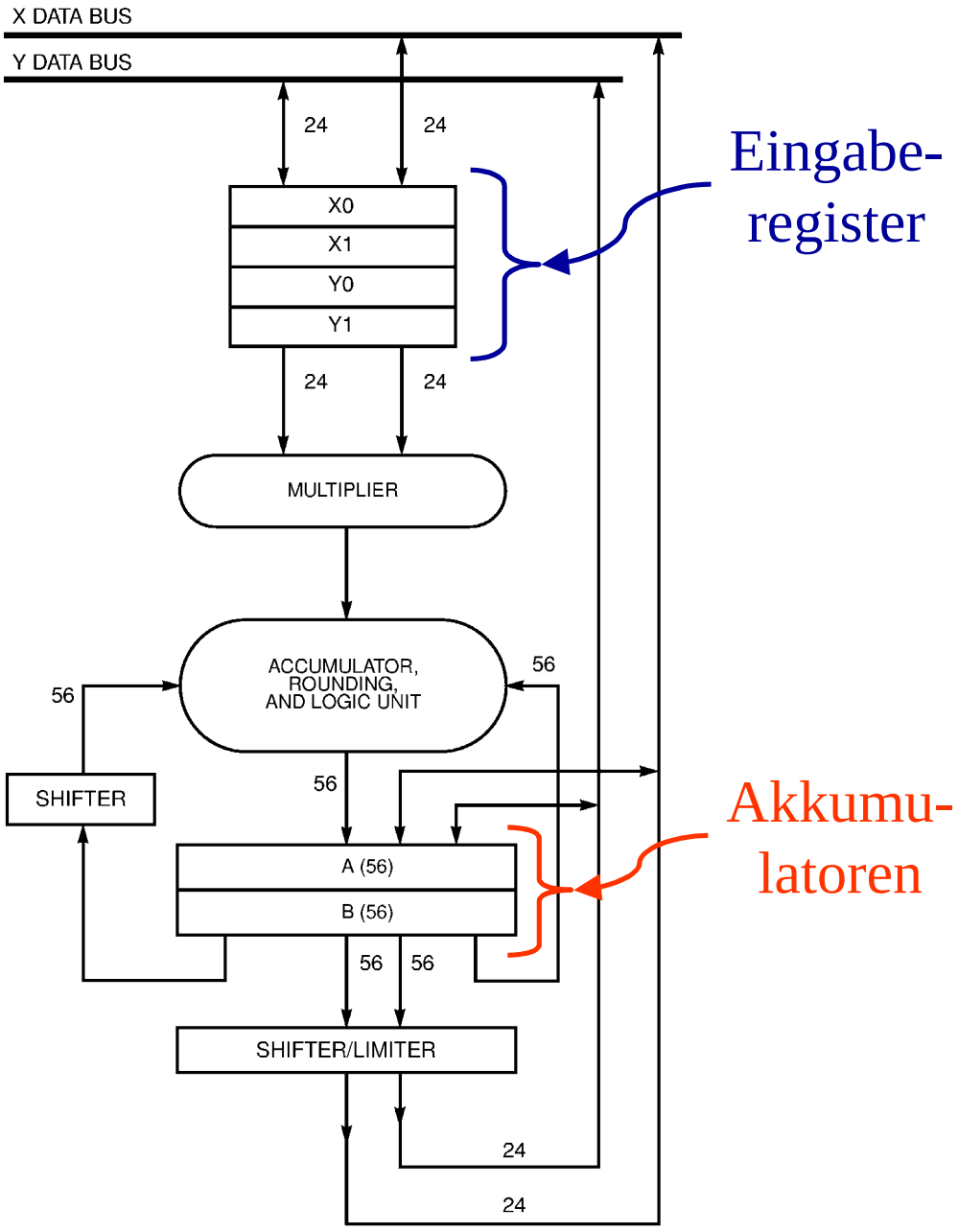



Bild 4.121: Blockdiagramm der ALU-Funktionen

Der DSP verfügt in der ALU des weiteren über vier Register X0, X1 und Y0, Y1 mit jeweils 24bit Länge, die bei Bedarf zu zwei 48 bit-Registern **X** und **Y** verbunden werden können. Diese Register können Speicherdaten über ihre zugehörigen Busse lesen oder ausgeben und sind die **Quellen** für **MAC**-Operationen. Bei der Ausführung eines **MAC**-Befehls werden zwei 24 bit-Werte z.B. aus den Registern X0 und Y0 miteinander multipliziert und das 48bit-Ergebnis wird einem der 56 bit Akkus (**A** oder **B**) additiv oder subtraktiv zugeführt. Gleichzeitig können die involvierten X und Y-Register durch parallele Datentransfers aus den zugehörigen Datenspeichern neu geladen werden. Dies läuft alles in einem einzigen Maschinenzyklus ab, d.h. wenn der DSP mit 100MHz getaktet wird, in 10ns.

In der ALU gibt es ferner zwei „Schiebeeinrichtungen“ (Shifter) zur Bitmanipulation und für die „dynamische“ Skalierung von Festkommadaten – siehe Bild 4.121. Zudem sorgt ein Begrenzer (Limiter) dafür, daß grobe arithmetische Fehler, die normalerweise bei binären Überläufen (...11111 ⇒...00000) entstehen würden, vermieden werden. Anstatt des Überlaufs nach ...00000 wird durch den Limiter der Maximalwert, der mit den verfügbaren Bits darstellbar ist, festgehalten und dieser Zustand wird mit einem „Flag“ im **C**ondition **C**ode **R**egister **(CCR)** signalisiert.

Die Pinbelegung des DSP56000 weist verschiedene Signalgruppierungen auf – siehe Bild 4.122. Es gibt drei Ports A, B und C mit zugehörigen Bussteuersignalen, zwei Interrupt-Pins, einen Reset-Pin sowie die Anschlüsse EXTAL, XTAL für die Takterzeugung. Zur Stromversorgung stehen 15 VCC- und 24 GND-Anschlüsse zur Verfügung Die integrierten Peripheriekomponenten (z.B. Timer und serielle Schnittstellen) sind nach dem bewährten Prinzip des „Memory Mapping“ (– vgl. SFRs beim Mikrocontroller 8051) im Adreßbereich $FFC0...$FFFF des X-Datenspeichers abgebildet.

# Pinfunktionen und Pinanordnung beim DSP5600x

| Functional Group | Number of Pins |
|---|---|
| Port A Data Bus | 24 |
| Port A Address | 19 |
| Port A Bus Control | 7 |
| Port B Host Interface | 15 |
| Port C Synchronous Comm. Interface | 3 |
| Port C Synchronous Serial Interface | 6 |
| Interrupt and Mode Control | 4 |
| PLL and Clock | 7 |
| On-chip Emulation (OnCE) | 4 |
| Power (VCC) | 16 |
| Ground (GND) | 24 |
| Timer | 1 |
| Reserved | 2 |
| Total (for the PGA package) | 132 |

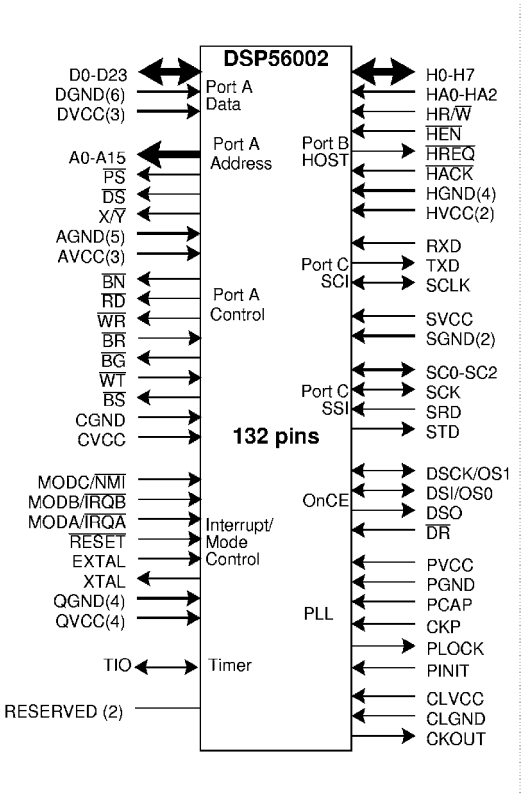



Bild 4.122: Die Anschlussbelegung des DSP5600x

Die Kommunikation nach außen erfolgt über **einen** 16bit-Adressbus und **einen** 24-bit-Datenbus sowie über die seriellen Schnittstellen.

Per Programm kann der DSP56000 veranlaßt werden, eine feste Anzahl von Wartezyklen bei externen Speicherzugriffen oder beim Zugriff auf langsame Peripheriekomponenten wie A/D- oder D/A-Wandler einzufügen.

An Port C stehen zwei voneinander unabhängige serielle Schnittstellen zur Verfügung, die sowohl synchronen wie auch asynchronen Vollduplexbetrieb zulassen.

#### 4.9.4.3 Das Programmiermodell für die ALU des DSP56000

Bei der Programmierung eines DSP5600x wird die Verwandtschaft mit der 16-bit-Mikroprozessorarchitektur des 68000 (Motorola) deutlich. Der Registersatz des DSP56000 gliedert sich in drei Hauptbereiche, die jeweils den Funktionseinheiten AGU (Adreßberechnungs-Einheit), ALU und dem Programm-Steuerwerk (Control Unit CU) zugeordnet sind – vgl. auch Bild 4.126. Die CU wurde bislang noch nicht behandelt; sie wird im Anschluß vorgestellt.

Die Daten-ALU sieht mit den beiden Akkumulatoren und der X/Y-Registerkombination einer Mi-

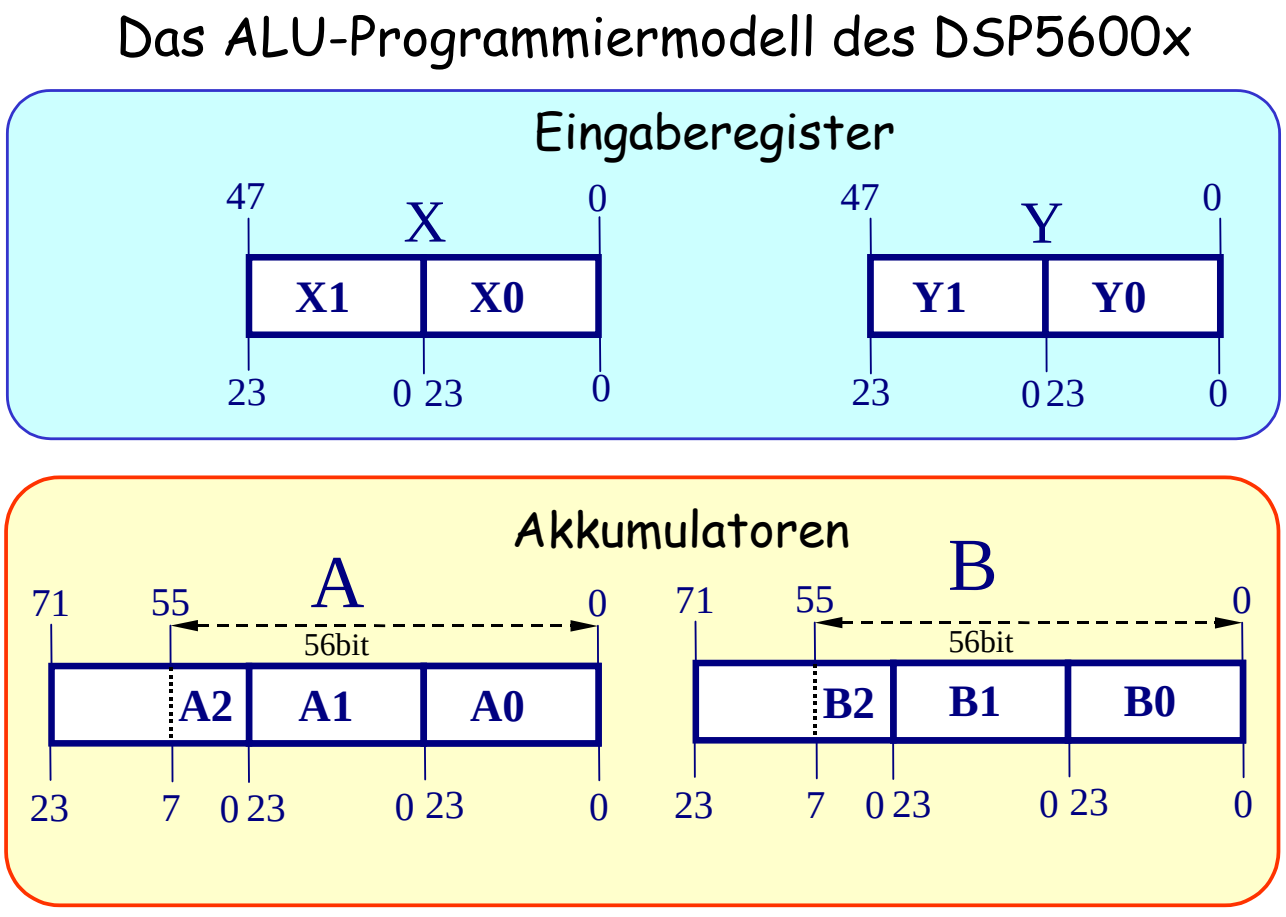



Bild 4.123: Modell zur ALU-Bedienung im DSP 5600x

kroprozessor-Architektur ziemlich ähnlich. Diese aufgabenorientierte Struktur resultiert aus der Spezialisierung von DSPs auf Multiplizier-Akkumulier-Operationen, die bekanntlich in der DSV die Schlüsselrolle spielen. Bild 4.123 gibt einen Überblick über Aufbau und Verwendung der Akkumulatoren **A** und **B** sowie der Register **X** und **Y**. Die Register **X** und **Y** liefern die Eingangsdaten für die ALU. Sie können als vier separate 24-bit-Register **X0**, **X1**, **Y0** und **Y1** verwendet werden oder auch als zwei 48bit Register **X** und **Y**, die jeweils durch Aneinanderreihen von **X1X0** und

**Y1Y0** definiert werden können. **X1** und **Y1** enthalten jeweils die höherwertigen Datenworte. Die Register dienen als Pufferspeicher zwischen den X- und Y-Datenbussen und der **MAC**-Einheit. Auf diese Weise liefern sie die Quelloperanden an die ALU, während gleichzeitig neue Operanden für den nächsten auszuführenden Befehl geholt werden können. Über die zugehörigen Datenbusse kann auch jederzeit der Registerinhalt ausgelesen werden.

Die Akkumulatoren sind vorzugsweise Ziele von **MAC**-Operationen. Generell sind drei Operanden in eine **MAC**-Operation eingebunden: Eingangseitig je ein 24bit Wort aus den **X**- und **Y**-Registern (**X0,X1,Y0,Y1**). Es wird eine vorzeichenbehaftete Multiplikation in fraktaler Zweierkomplementdarstellung durchgeführt und das 48bit lange Produkt wird in Fraktaldarstellung zum Inhalt eines der Akkumulatoren **A** oder **B** addiert (dritter Operand). Das Ergebnis wird in den entsprechenden Akku in einer Wortlänge von 56bit zurückgespeichert – von dieser Wortlänge stammt im übrigen die Bezeichnung „DSP 56000“. Die 8-bit-Erweiterung der Akkus erlaubt das Erfassen von bis zu **256** Überläufen. Die erweist sich in vielen Anwendungen als zu knapp, so daß neuere Versionen der DSPs hier ebenfalls auf 24 bit erweitert worden sind.

Ein **MAC**-Ergebnis wird so abgelegt, daß der Akku-Teil **A0** bzw. **B0** die niederwertigsten 24 bit (**LSP** ≡ **L**east **S**ignificant **P**roduct) und der Teil **A1** bzw. **B1** die höherwertigen 24 bit (**MSP** ≡ **M**ost **S**ignificant **P**roduct) erhält. Überläufe werden in der Erweiterung **A2** bzw. **B2** abgelegt.

#### 4.9.4.4 Das Programmsteuerwerk CU

Das Programmsteuerwerk (**CU** ≡ Control Unit) verfügt über drei 24 bit breite Kernregister, die den Programmadreßzähler (**PC** ≡ **P**rogram **C**ounter), die Statusinformation (im **SR** ≡ **S**tatus **R**egister) und die Betriebsart (im **OMR** ≡ **O**peration **M**ode **R**egister) beinhalten. Zwei zusätzliche Register mit den Bezeichnungen LC ≡ LOOP COUNTER und LA ≡ LOOP ADDRESS erlauben der Hardware unmittelbar Schleifenoperationen ohne **“Software Overhead“** durchzuführen. Sogenannte “**DO-LOOPs“**, die – wie schon an zahlreichen Beispielen gezeigt wurde - in der DSV sehr häufig sind, können mit der CU-Erweiterung ohne Zeitverzug ablaufen. Des weiteren verfügt die CU über 16 Stackregister mit 32 bit Breite und den zugehörigen Stackpointer.

Wegen der Bedeutung in der DSV wird hier exemplarisch die Hardware DO-LOOP-Steuerung näher betrachtet.

Für den Programmierer sind folgende Register aus Bild 4.124 bei Hardware DO-LOOPs von Bedeutung:

- PC Program Counter
- LA LOOP-Adresse (Adresse des letzten Befehls des Loops)
- LC LOOP-Counter (Anzahl der Durchläufe)

Der Stack hat 16 Plätze mit je 32bit und ist aufgeteilt in 2 Bänke SSH und SSL zu je 16bit. Hier können jeweils die 16 bit des PC und des SR untergebracht werden, so daß ein Unterprogramm oder ein Interrupt nur einen Stackplatz belegt.

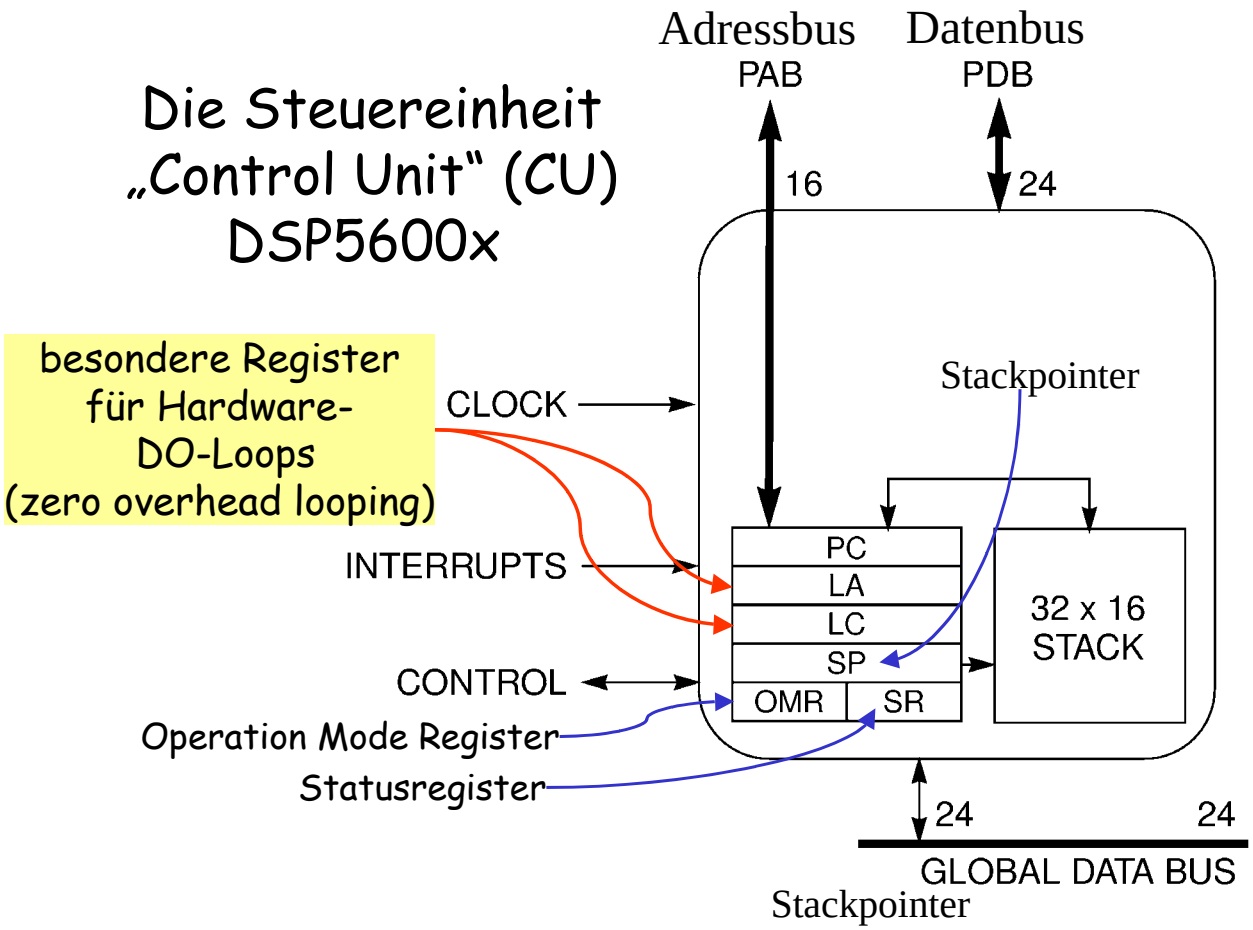



Bild 4.124: Übersicht über die Control Unit (CU) des DSP 5600x

Bei Hardware DO-LOOPs kommen die beiden neuen Register **LA** und **LC** hinzu, so daß zwei Plätze benötigt werden.

Eine typische Software-Schleife könnte z.B. wie folgt aussehen:

```
FOR i=1 to N
  1. Befehl
.
.
.
letzter Befehl
NEXT i
```

Dagegen wird ein Hardware-DO-LOOP, der z.B. eine Korrelationsfunktion mit 1000 Abtastwerten realisiert, wie folgt im DSP programmiert:

```
        DO #1000, END1
        MAC X0,Y0,A   X:(R0)+,X0   Y:(R4)+,Y0
END1
```

das Korrelationsergebnis steht jetzt im Akku **A**

Wie schon angedeutet, müssen bei Start eines Hardware-DO-LOOPs vier 16-bit-Register auf den Stack gebracht werden:

| **PC**<br>Programmzähler | **SR**<br>Statusregister | **LA**<br>Loopadresse | **LC**<br>Loopcounter |
|---|---|---|---|
| ⇑ | ⇑ | ⇑ | ⇑ |

kommen auf den STACK bei DO-LOOPs

Details der Abarbeitung eines DO-LOOPs

DO #N, END1

⇓ ⇓

LC LA (letzter Befehl im LOOP)

**PC ⇒ Stack** ; markiert den LOOP-Anfang

**SR ⇒ Stack**

**LA ⇒ Stack** ; für Verschachtelung (Nesting)

**LC ⇒ Stack** ; für Verschachtelung (Nesting)

Folgende Schritte laufen im LOOP bei jedem OP-Code-Holen ab:

Vergleich: **PC ⇔ LA**

wenn **PC = LA** ;letzter LOOP-Befehl liegt vor

wenn **LC > 1** ;LC erniedrigen und weitermachen

LC:=LC-1

PC vom Stack holen und 1. LOOP-Befehl ausführen

Dieser Ablauf wiederholt sich so oft, bis

**LC = 1**

dann erfolgt noch ein genau **ein** Durchlauf

danach: LA, LC, SR vom Stack zurückholen

Programm läuft weiter bei PC:=LA+1

Aufgrund der parallel arbeitenden rechenfähigen Funktionseinheiten AGU und ALU und der Harvard-Architektur mit drei vollständigen **separaten** Bussystemen ist folgende Befehlsstruktur möglich, die wohl am besten den Leistungssprung vom Standard-Mikroprozessor zum DSP verdeutlicht.

| Opcode | Operanden | X-Daten | Y-Daten |
|---|---|---|---|
| MAC | X0,Y0,A | X:(R0)+,X0 | Y:(R4)+,Y0 |

Man sieht, daß parallel zur MAC-Operation zwei Datentransfers ausgeführt werden, die dafür sorgen, daß bei Wiederholung der Instruktion (typischerweise in einem LOOP) - nachdem die Pointer R0 und R4 jeweils um Eins erhöht worden sind – neue Daten in den Registern X0 und Y0 vorhanden sind, die nun in die MAC-Operation einfließen.
Alternativ zum oben behandelten Hardware-DO-LOOP kann zur effizienten Programmierung einer Korrelationsfunktion auch der Wiederholungsbefehl *REP* – s. Liste Befehle zur Programmablaufsteuerung – verwendet werden. Dadurch wird der Programmcode noch kürzer:

```
REP #1000
MAC X0,Y0,A    X:(R0)+,X0    Y:(R4)+,Y0
```

Dies geht jedoch nur dann, wenn die gesamte Schleife aus nur einer einzigen Instruktion besteht. Eine REP-Instruktion ist zudem - im Gegensatz zum DO-LOOP - nicht unterbrechbar.


Bild 4.125: Aufbau einer typischen DSP-Instruktion

#### 4.9.4.5 Befehlssatz des DSP56000 in einer Übersicht

Arithmetische Befehle

| | |
|---|---|
| ABS | Absolute Value |
| ADC | Add Long with Carry |
| ADD | Addition |
| ADDL | Shift Left and Add |
| ADDR | Shift Right and Add |
| ASL | Arithmetic Shift Left |
| ASR | Arithmetic Shift Right |
| CLR | Clear an Operand |
| CMP | Compare |
| CMPM | Compare Magnitude |
| DEC | Decrement by One |
| DIV | Divide Iteration |

| | |
|---|---|
| INC | Increment by One |
| MAC | Signed Multiply-Accumulate |
| MACR | Signed Multiply-Accumulate and Round |
| MPY | Signed Multiply |
| MPYR | Signed Multiply and Round |
| NEG | Negate Accumulator |
| NORM | Normalize |
| RND | Round |
| SBC | Subtract Long with Carry |
| SUB | Subtract |
| SUBL | Shift Left and Subtract |
| SUBR | Shift Right and Subtract |
| Tcc | Transfer Conditionally |
| TFR | Transfer Data ALU Register |
| TST | Test an Operand |

## Logische Verknüpfungen/Schiebeoperationen

| | |
|---|---|
| AND | Logical AND |
| ANDI | AND Immediate to Control Register |
| EOR | Logical Exclusive OR |
| LSL | Logical Shift Left |
| LSR | Logical Shift Right |
| NOT | Logical Complement |
| OR | Logical Inclusive OR |
| ORI | OR Immediate to Control Register |
| ROL | Rotate Left |
| ROR | Rotate Right |

## Befehle zur Bitmanipulation

| | |
|---|---|
| BCLR Bit | Test and Clear |
| BSET Bit | Test and Set |
| BCHG Bit | Test and Change |
| BTST Bit | Test on Memory and Registers |

## Befehle zum LOOP-Aufbau

| | |
|---|---|
| DO | Start Hardware Loop |
| ENDDO | Exit from Hardware Loop |

## Transferbefehle (Moves)

| | |
|---|---|
| LUA | Load Updated Address |
| MOVE | Move Data Register |
| MOVEC | Move Control Register |
| MOVEM | Move Program Memory |
| MOVEP | Move Peripheral Data |

## Befehle zur Programmablaufsteuerung

| | |
|---|---|
| II | Illegal Instruction |
| Jcc | Jump Conditionally |
| JMP | Jump |
| JCLR | Jump if Bit is Clear |
| JSET | Jump if Bit is Set |
| JScc | Jump to Subroutine Conditionally |
| JSCLR | Jump to Subroutine if Bit is Clear |
| JSR | Jump to Subroutine |
| JSSET | Jump to Subroutine if Bit is Set |
| NOP | No Operation |
| REP | Repeat Next Instruction |
| RESET | Reset On-Chip Peripheral Devices |
| RTI | Return from Interrupt |
| RTS | Return from Subroutine |
| STOP | Stop Processing (Low Power Stand-By) |
| SWI | Software Interrupt |
| WAIT | Wait for Interrupt (Low Power Stand-By) |

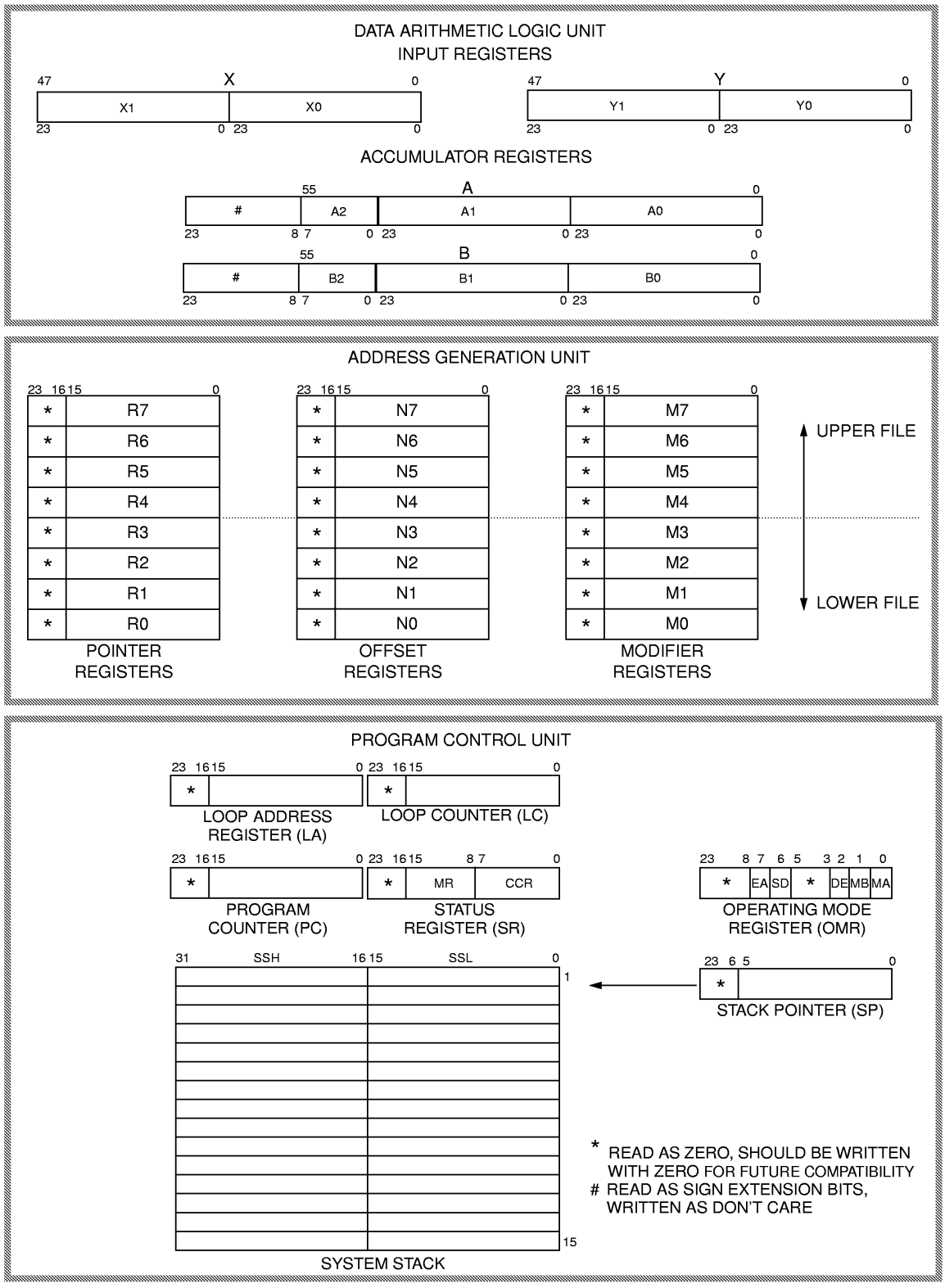



Bild 4.126: Das vollständige Programmiermodell des DSP5600x

# 5 Entwurf anwendungsspezifischer Mikrorechnersysteme

## 5.1 Grundbegriffe der Hardware-Beschreibungssprache VHDL

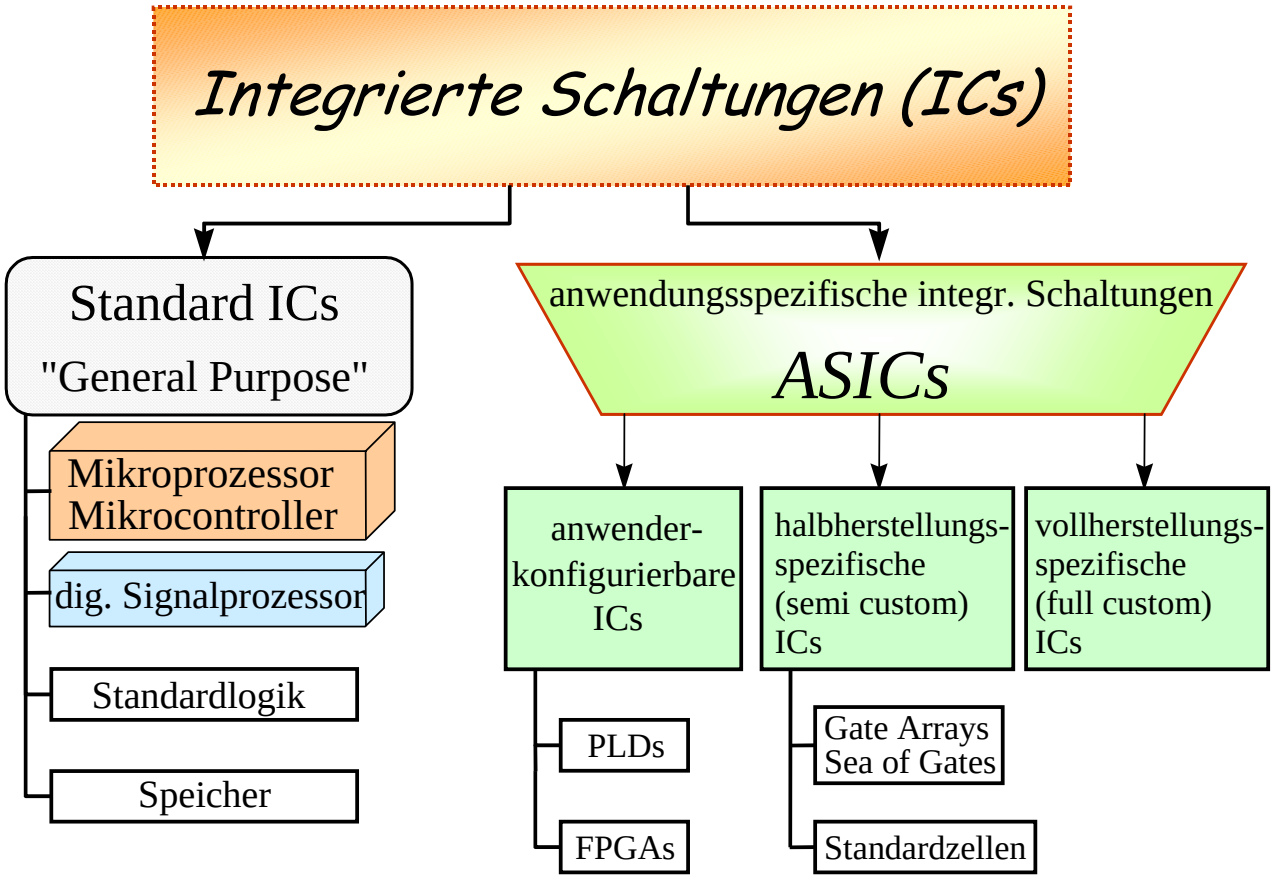



Bild 5.1: Übersicht über die Varianten digitaler integrierter Schaltungen

Für den Entwurf digitaler integrierter Schaltungen – wie sie im rechten Teil von Bild 5.1 kurz umrissen sind – können grundsätzlich zwei verschieden Vorgehensweisen angewandt werden.

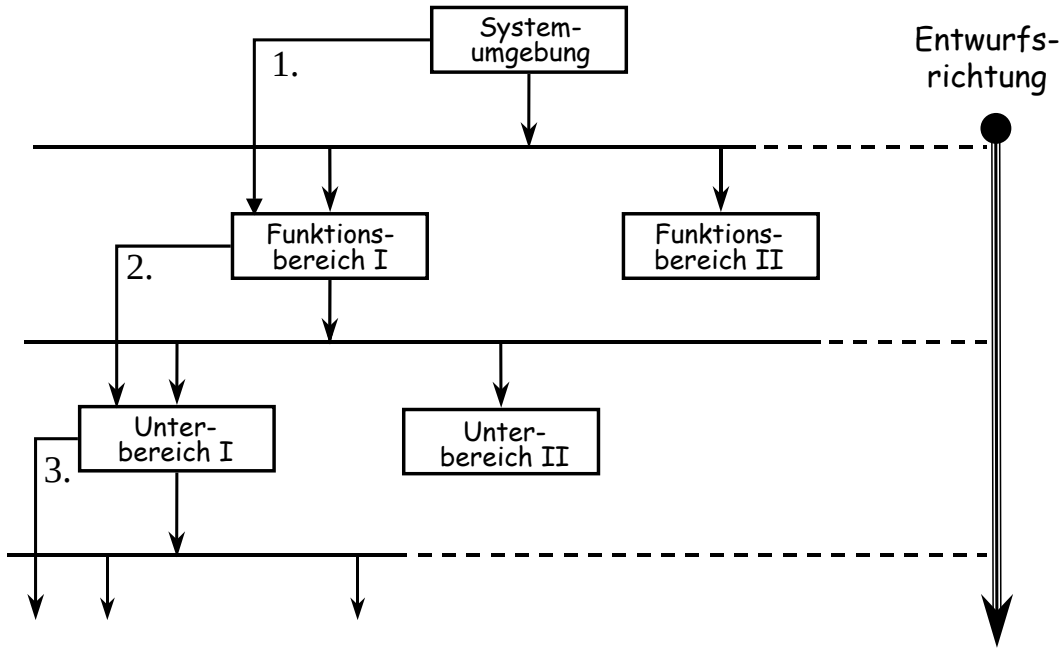



Bild 5.2: Schema des Top-Down-Entwurfs

Für große Projekte, die in einem Team bewältigt werden empfiehlt sich die in Bild 5.2 skizzierte Top-Down-Vorgehensweise. Sie geht von der Systemsicht aus und zerlegt die darunter liegenden Ebenen in die jeweils benötigte Detaildarstel-

lung. Es ergibt sich eine klare Gliederung, die das Endziel – das fertige, aus vielen aneinander angepaßten Einzelteilen bestehende System – stets im Auge behält. Eine „Verzettelung“ in Details wird bei der Top-Down-Methode vermieden.

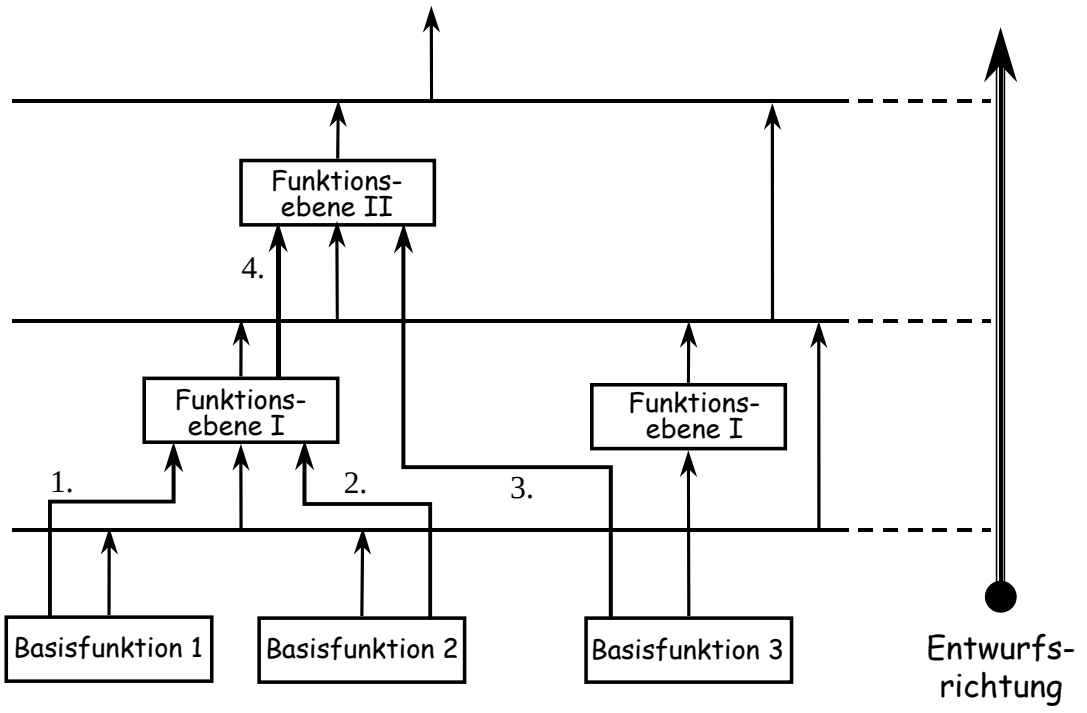



Bild 5.3: Schema des Bottom-Up-Entwurfs

Obwohl der in Bild 5.3 gezeigte Bottom-Up-Entwurfsstil diese Gefahr birgt, hat er dennoch seine Berechtigung und kann immer dann besonders erfolgreich praktiziert werden, wenn bereits Vorerfahrung in Form von realisierten Basisfunktionsblöcken vorhanden ist. Solche Blöcke können sehr schnell zu größeren übergeordneten Funktionseinheiten kombiniert werden, die dann auf den höheren Ebenen getestet werden können, ohne daß jede Basisfunktion nochmals verifiziert werden muß. Das Ziel – ein komplettes funktionsfähiges System – kann mit der Bottom-Up-Methode vom erfahrenen Entwickler (oder Entwicklungsteam) oft schneller erreicht werden, als mit der Top-Down-Methode. Allerdings darf nicht übersehen werden, daß insbesondere bei großen Projekten eine Bottom-Up-Vorgehensweise deutlich höherer Anforderung an das Projektmanagement stellt.

Elektronische Systeme aller Art – nicht nur digitale integrierte Schaltungen, mit denen wir uns hier schwerpunktmäßig befassen – sind seit den 80-er Jahren derart komplex geworden, daß eine Übersicht anhand von Standard-Dokumenten (Handbücher, Betriebs- und Reparaturanleitungen) nicht mehr zu gewinnen ist. Vor allem ein Vergleich zwischen verschiedenen Herstellern ist – nicht zuletzt durch die verschiedenartigsten Schutzmaßnahmen der Hersteller für ihr Know-How und ihr „geistiges Eigentum“ (engl.: Intellectual Property ≡ IP) – praktisch unmöglich geworden. Aus dieser unbefriedigenden Situation, die natürlich nicht nur den zivilen Handel und die Industrie betraf sondern auch Behörden und vor allem das Militär entstand Anfang der 80-er Jahre in den USA eine Initiative die die Grundlage für die heute weltweit am weitesten verbreitete Hardwarebeschreibungssprache VHDL bildete.

### 5.1.1 Historische Entwicklung von VHDL

Die Anfänge von VHDL[24] reichen – wie erwähnt - bis in die frühen achtziger Jahre zurück. Im Rahmen eines sogenannten **VHSIC**[25]-Programms wurde in den USA vom Verteidigungsministerium (Department of Defense, **DoD**) nach einer Sprache zur Dokumentation elektronischer Systeme gesucht. Dadurch sollten die enormen Kosten für **Wartung und Nachbesserung** bei militärischen Systemen, die etwa die Hälfte der Gesamtkosten ausmachten, reduziert werden. Von besonderer Wichtigkeit war klare Beschreibung von komplexen Schaltungen sowie die **Austauschbarkeit** von Modellen zwischen verschiedenen Entwicklungsgruppen. Das **VHSIC**-Programm startete mit einem Workshop zur Festlegung der Ziele. Im Juli 1983 bekamen schließlich die Firmen Intermetrics, IBM und Texas Instruments den Auftrag, die neue Sprache zu entwickeln. Sie sollte sich an die im militärischen Bereich weit verbreitete Programmiersprache **ADA** anlehnen. Im August 1985 konnte eine erste Version - VHDL V7.2 - vorgestellt werden, die dann im Februar 1986 zur Standardisierung an **IEEE**[26] übergeben wurde. Unter Beteiligung zahlreicher Software- und Hardwarehersteller aus der **CAE**[27]-Industrie gewann VHDL zunehmend Anhänger und wurde schließlich im Dezember 1987 als **IEEE l076-1987** zum ersten und bislang einzigen Standard für Hardwarebeschreibungssprachen. Der Standard definiert allerdings nur die **Syntax** und **Semantik** der Sprache selbst, nicht jedoch ihre Anwendung bzw. ein einheitliches Vorgehen bei der Anwendung.

Seit September 1988 müssen alle Elektronik-Zulieferer des **DoD** VHDL-Beschreibungen ihrer Komponenten und Subkomponenten bereitstellen. Ebenso sind VHDL-Modelle von Testumgebungen mitzuliefern.

Nach den **IEEE**-Richtlinien muß ein Standard alle **fünf Jahre** überarbeitet werden, um nicht zu verfallen. Aus diesem Grund wurde im Zeitraum 1992-1993 die Version **IEEE 1076-1993** definiert. Die Dokumentation des neuen Standards, im „Language Reference Manual“ (**LRM**) erschien Mitte 1994. Seit Beginn der neunziger Jahre hat sich VHDL weltweit etabliert. Der **1076-l993**-Standard entstand bereits weitgehend losgelöst vom **DoD** unter internationaler Beteiligung. Europa wirkte unter anderem über **ESPRIT** daran mit. Auch Asien - vor allem Japan - hat VHDL akzeptiert. Konkurrenz besteht heute praktisch nur noch durch „**Verilog HDL**“ in den USA, weil der führende **CAE**-Hersteller Cadence diese HDL in seine Werkzeuge eingebaut hat. Während **Verilog** über lange Zeit als „Cadence-proprietär“ anzusehen war, ist auch diese HDL mittlerweile ein offener Standard, der in den meisten **CAE**-Werkzeugen für den Entwurf integrierter Schaltungen gleichberechtigt neben VHDL zur Verfügung steht.

---

[24] **V**HSIC **H**ardware **D**escription **L**anguage
[25] **V**ery **H**igh **S**peed **I**ntegrated **C**ircuit
[26] **I**nstitute of **E**lectrical and **E**lectronics **E**ngineers
[27] **C**omputer **A**ided **E**ngineering

### 5.1.2 Entwurfsebenen und „Sichtweisen" integrierter Schaltungen

Sobald man sich mit dem Entwurf integrierter Schaltungen befaßt, wird man zwangsläufig mit einem Modell konfrontiert, daß die verschiedenen Entwurfsebenen aus den drei Sichten Verhalten, Struktur und Geometrie wiedergibt. Die Darstellung ist unter der Bezeichnung Gajski-Walker-Modell bekannt.

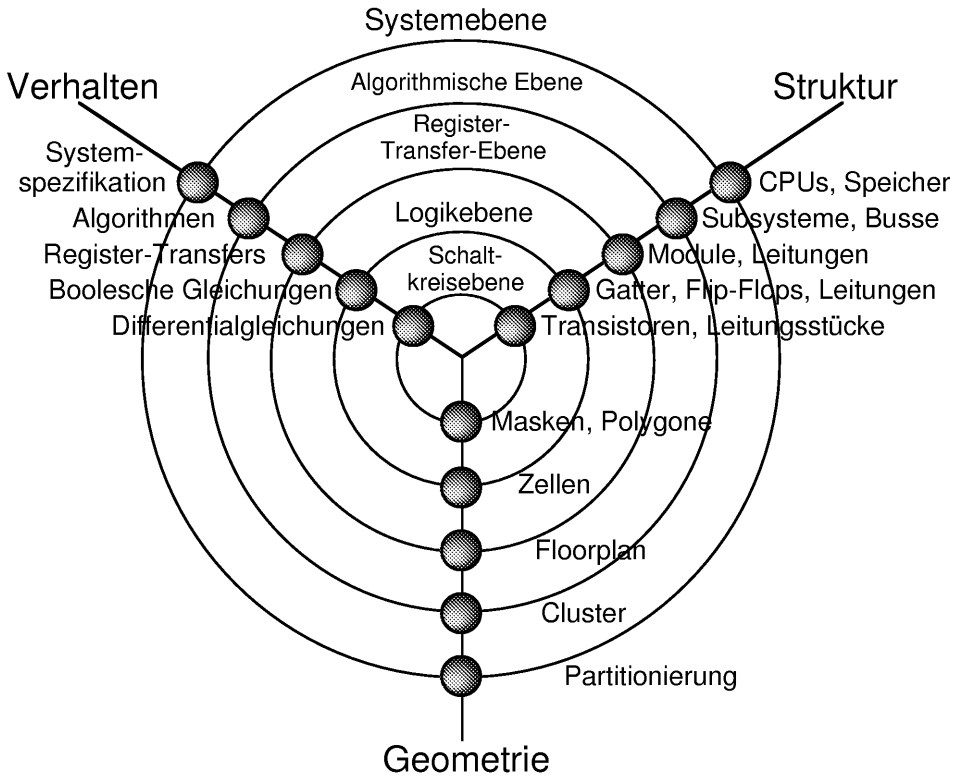



Bild 5.4: Das Gajski-Walker-Modell (auch Y-Modell genannt)

Die einzelnen Ebenen werden im folgenden kurz aus den drei Sichtweisen betrachtet, und ihre Rolle beim Entwurf digitaler integrierter Schaltungen wird erläutert.

#### Systemebene

Die Systemebene beschreibt die grundlegenden Charakteristika einer Schaltung.

In der Beschreibung werden typische Blöcke, wie Speicher, Prozessoren oder Interface-Einheiten verwendet, die jeweils durch ihre Funktionalität (z.B. Befehlssatz, Protokoll) charakterisiert werden.

Auf dieser Ebene dienen „natürliche" Sprache und Skizzen als Beschreibungsmittel. Die Systemebene liefert noch keine Aussagen über Signale, deren Zeitverhalten und weitere funktionale Details. Vielmehr dient sie einer groben Aufteilung der gesamten Schaltungsfunktion. Aus geometrischer Sicht beinhaltete die Systemebene beispielsweise eine rudimentäre Aufteilung der Chipfläche.

## Algorithmische Ebene

Hier kann eine Schaltungsbeschreibung durch nebenläufige (d.h. parallel ablaufende) Algorithmen erfolgen. Typische Elemente sind dabei Funktionen, Prozeduren, Prozesse und Kontrollstrukturen.

Aus strukturaler Sicht wird hier ein elektronisches System durch Blöcke beschrieben, die über Signale miteinander kommunizieren.

Die Verhaltenssicht enthält z.B. eine algorithmische Darstellung mit Variablen und Operatoren. Es wird noch kein Bezug zur Struktur der späteren Realisierung hergestellt. So fehlen z.B. auch noch alle zeitlichen Vorgaben bezgl. Takt- oder Rücksetzsignalen.

## Register-Transfer-Ebene

Auf der Register-Transfer-Ebene (engl.: **R**egister **T**ransfer **L**evel, **RTL**) werden die Eigenschaften einer Schaltung z.B. durch mathematische oder logische Operationen und durch die Transfermöglichkeiten der zu verarbeitenden Daten zwischen Registern spezifiziert – vgl. Mikrosequencer.

In die Beschreibung fließen in der Regel bereits Takt- und Rücksetzsignale ein. Die einzelnen Operationen können dann den Zuständen oder Flanken dieser Signale zugeordnet werden, so daß zeitliche Eigenschaften schon weitgehend definiert sind. Aus strukturaler Sicht werden Elemente wie Register, Zähler, Adreßdecoder, Multiplexer oder Addierer durch Signale miteinander verknüpft. Aus Verhaltenssicht ergeben sich meist Beschreibungen in Form endlicher Automaten. In geometrischer Sicht wird die Grobeinteilung der Chipfläche zu einem „Floorplan“ verfeinert.

Es gibt eine Reihe von CAE-Werkzeugen für die automatische Umsetzung einer Verhaltensbeschreibung in eine strukturale Beschreibung auf der Logikebene, d.h. RTL-Beschreibungen sind in der Regel synthetisierbar.

## Logikebene

Hier werden die Eigenschaften eines elektronischen Systems durch logische Verknüpfungen und deren zeitliche Eigenschaften (Verzögerungszeiten) beschrieben.

Der Verlauf der Ausgangssignale ergibt sich dabei durch die Anwendung dieser Verknüpfungen auf die Eingangssignale. Die Signalverläufe sind wertdiskret, z.B. '**low**', '**high**', '**tristate**'.

In strukturaler Sicht wird ein Design durch eine Zusammenschaltung von Grundelementen (AND-, OR-, EXOR-Gatter, Flip-Flops) dargestellt. Diese Grundelemente werden dabei von einer Bibliothek zur Verfügung gestellt. Innerhalb dieser Bibliothek sind die Eigenschaften der Grundelemente definiert. Hier sind die Charakteristika der einzelnen Zellen der Zieltechnologie in vereinfachter Form abgelegt, z.B. 190nm-Standardzellen-Prozeß der Fa. ATMEL, FPGA vom Typ ACEX EP 1K 100 Fa. ALTERA. Zur Erstellung der strukturalen Beschreibung (Gatternetzliste) werden meistens graphische Eingabewerkzeuge verwendet („Schematic Entry“).

Aus Verhaltensicht erfolgt vornehmlich eine Darstellung durch Boolesche Gleichungen, z.B. Y=(A AND B) XOR C oder durch Werte- bzw. Funktionstabellen. Der Übergang von der Verhaltenssicht auf die strukturale und geometrische Sicht erfolgt mit Hilfe von Synthesesoftware.

## Schaltkreisebene

Hier liegt struktural eine „verfeinerte“ Netzliste vor, d.h. es sind keine logischen Gatter oder noch komplexere Funktionseinheiten zusammengeschaltet, sondern Bauelemente wie Transistoren, Kondensatoren und Widerstände.

Einzelne Module werden nicht mehr durch logische Funktionen mit einfachen Verzögerungen beschrieben, sondern im Detail durch ihren physikalischen Aufbau.

Aus geometrischer Sicht hat man Polygonzüge vor sich, die z.B. unterschiedliche Dotierungsbereiche auf der Halbleiteroberfläche definieren.

Aus Verhaltenssicht benötigt man vornehmlich Differentialgleichungen zur Schaltungsbeschreibung, so daß Simulationen auf dieser Ebene entsprechend aufwendig und rechenintensiv sind.

Signale auf Schaltkreisebene können im Gegensatz zur Logikebene beliebige Werte zu beliebigen Zeitpunkten annehmen, d.h. sie sind zeit- und wertkontinuierlich.

Jede der vorgestellten Entwurfsebenen erfüllt einen bestimmten Zweck. Während auf den oberen Ebenen hohe Komplexitäten beherrschbar sind, bieten die unteren Ebenen Details und Genauigkeit. Systemebene und algorithmische Ebene eignen sich daher für die Dokumentation und Simulation des Gesamtsystems, während die Register-Transfer-Ebene für die Blocksimulation und synthesegerechte Modellierung geeignet ist. Bild 5.5 vermittelt einen anschaulichen Eindruck der Beschreibungen auf den verschiedenen Ebenen, die vorstehend behandelt wurden, während Bild 5.6 schematisch im Überblick die hierarchische Struktur hervorhebt.

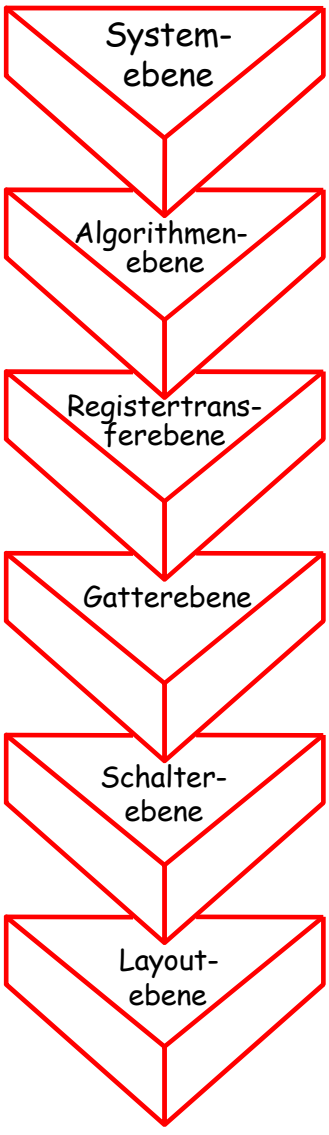



Bild 5.6: Symbolische Darstellung der Entwurfsebenen

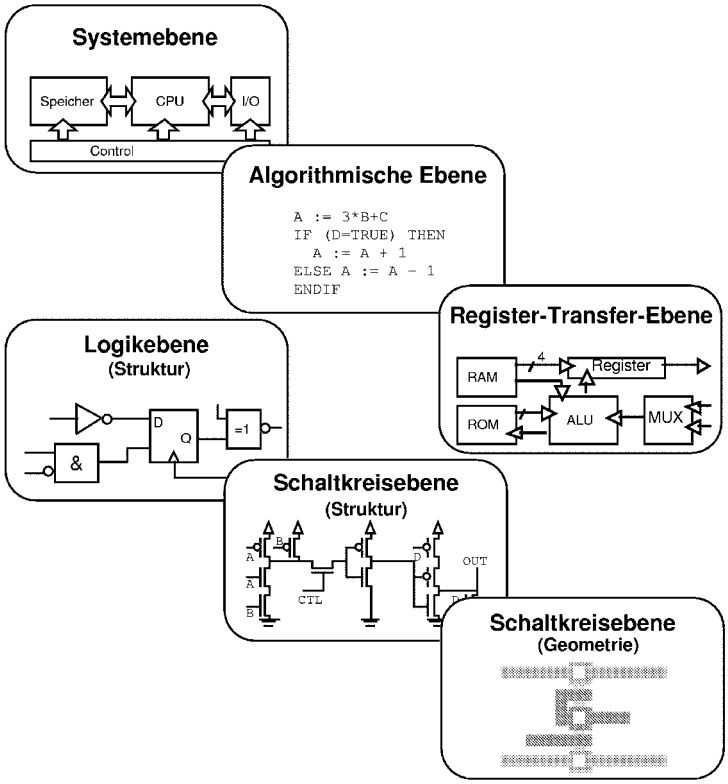



Bild 5.5: Anschauliche Übersicht über die Entwurfsebenen

Auf jeder Ebene wird nur die **benötigte Genauigkeit** geboten; unwichtige Details sind nicht sichtbar. Eine eindeutige Einordnung einer Beschreibung in eine bestimmten Ebene gelingt nicht immer, es gibt somit auch „Multi-Level“-Beschreibungen.

### 5.1.3 Wozu benötigt man Hardwarebeschreibungssprachen

In zahlreichen Anwendungen digitaler Technik trifft man den Mikrorechner immer weniger als alleinstehendes Element an. In den meisten Fällen ist er in ein mehr oder weniger umfangreiches **System**, das in der Regel auch analoge Bestandteile hat, eingebaut (oder eingebettet, engl.: embedded). Oft ist dabei die Komplexität – z.B. gemessen in der Zahl der Transistoren oder der benötigten Siliziumfläche - der den eigentlichen Mikrorechner umgebenden Schaltung weitaus größer als der Rechnerkern. Derartige Systeme sind keineswegs etwas völlig Neues, sondern werden seit Jahrzehnten auf Leiterplatten aufgebaut. Aufgrund der enormen Fortschritte der Halbleitertechnologie in den letzten Jahren ist es jedoch heute möglich, neben speziellen anwendungsspezifischen Schaltungsteilen auf einem Stück Silizium auch mehrere Standard-Rechnerkerne – d.h. sowohl Mikroprozessoren und Mikrocontroller als auch digitale Signalprozessoren - zu implementieren. Ein vollständiges „Embedded System“ schließt darüber hinaus auch analoge Schaltungsteile ein. Dazu zählen sowohl Analog/Digital- und Digital/Analogwandler als auch Operationsverstärker, mit denen z.B. analoge Filter oder digital einstellbare Verstärker realisiert sind. Letztere spielen sowohl in der Meß- und Automatisierungstechnik als auch in der Kommunikationstechnik eine wichtige Rolle, d.h. immer dort, wo man es mit Signalen, die einen sehr großen Dynamikbereich umfassen, zu tun hat. Im Zusammenspiel mit den digitalen Schaltungsteilen lassen sich auch adaptive Konzepte realisieren.

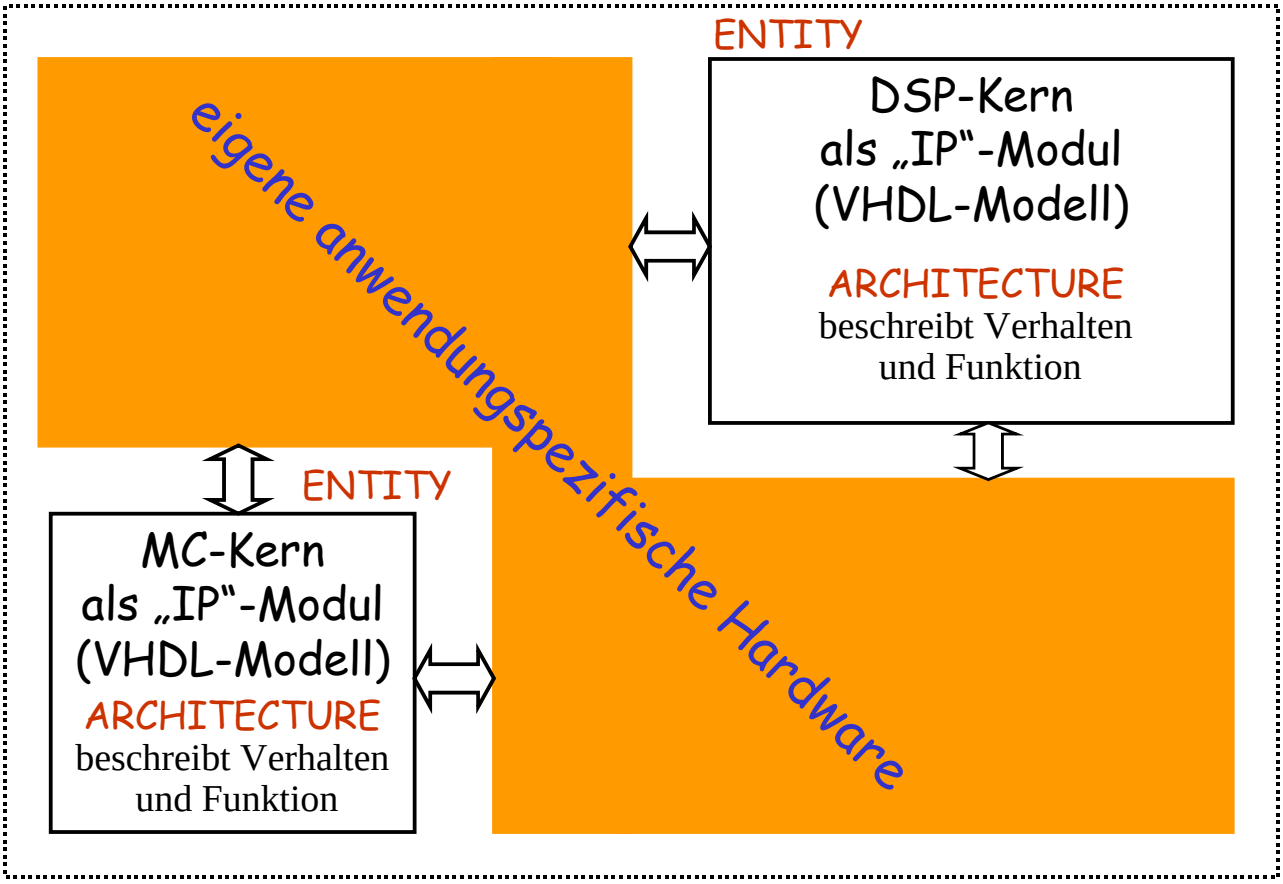



Bild 5.7: Konzept zum Aufbau komplexer Systeme auf Silizium

Die Vorteile, die sich durch eingebettete Systeme ergeben, sind so gewaltig, daß kein Unternehmen sie heute mehr außer Acht lassen kann. Auch mit noch so kostengünstiger Leiterplattentechnik gelingt es nicht mehr Produkte zu schaffen die gegenüber der Integration konkurrenzfähig wären. Bild 5.7 illustriert die Grundgedanken die zum (eingebetteten) System auf Silizium (engl.: **S**ystem **o**n **S**ilicon **SoS**) führen.

### 5.1.4 Wie entsteht eine integrierte Schaltung unter Verwendung von VHDL

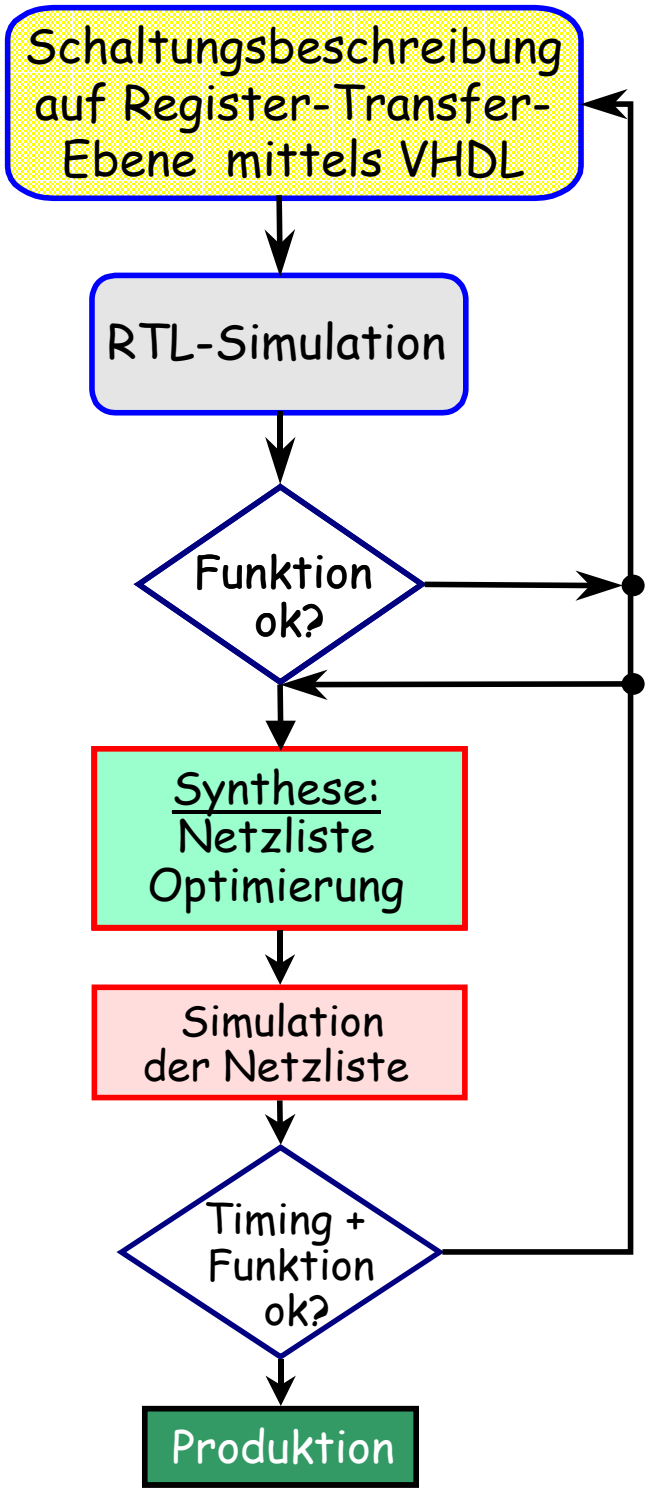



Bild 5.8: Prinzipieller Ablauf von der RTL-Beschreibung bis zur Produktion

Wie bereits angedeutet, können nicht alle VHDL-Modelle in Schaltungen umgesetzt werden. Ein „sicherer“ Ausgangspunkt ist jedoch immer eine Beschreibung auf der Register-Transfer-Ebene, die im Zusammenhang mit dem Mikrosequencer-Entwurf ausführlich diskutiert wurde. Die Schaltungsbeschreibung auf dieser Ebene ist dem Ingenieur vertraut, so daß er sich vom grundlegenden Verständnis einer Schaltung hier sehr rasch zurechtfindet. Die Umsetzung in den VHDL-Text erfordert allerdings einige Einarbeitung, nicht nur zum Erlernen der Sprachsyntax sondern auch zum Entwickeln eines sogenannten Modellierungsstils. Dies ist unbedingt nötig, da VHDL keine Vorgaben in dieser Richtung macht, d.h. ein Sachverhalt kann auf vielen verschieden Wegen beschrieben werden, die im Ergebnis natürlich auch zu verschieden aufgebauten Schaltungen führen. Da die Eigenschaften der Ergebnisse sich signifikant hinsichtlich Aufwand und Geschwindigkeit unterscheiden können, erhebt sich für den Ingenieur immer die Frage der Optimierung. Dies ist beim Einsatz von VHDL für den Anfänger eine äußerst komplexe, schwer überschaubare Angelegenheit, d.h. nur durch Sammeln von Erfahrung im Rahmen zahlreicher Entwürfe kann die nötige Sicherheit erworben werden.

Bild 5.8 gibt einen allgemeinen, grundlegenden Überblick über die Schritte vom VHDL-Text bis hin zur IC-Produktion.

An erster Stelle steht nach Fertigstellung der Texteingabe die VHDL-Simulation. Hier wird – noch ganz unabhängig von der Realisierungstechnologie - die Funktion der Schaltungsidee überprüft. Auftretende Fehler und/oder nötige Verbesserungen und Erweiterungen führen solange zurück zur VHDL-Textedition, bis ein zufriedenstellendes Ergebnis erreicht ist. Dann wird der nächste Schritt, die Synthese eingeleitet, mit der man die Ebene der Technologieunabhängigkeit verläßt - vgl. Bild 5.9. Es entsteht dabei die sogenannte Netzliste, ein Schaltplan, der aus „logischen Primitiven“ aufgebaut ist, d.h. z.B. aus Elementargattern und Flip-Flops. Die Netzliste wird je nach Synthesewerkzeug noch optimiert, bevor eine Simulation der Schaltung auf dieser Ebene erfolgt. Der wesentliche Unterschied zur beschriebenen VHDL-Simulation ist, daß jetzt das Zeitverhalten der Elemente – so wie es der Hersteller spezifiziert hat – mit einbezogen wird, d.h. typische Verzögerungszeiten von

Ein- zu Ausgängen sowie Setup- und Hold-Zeiten. Entdeckte Fehler oder nötige Verbesserungen auf dieser Ebene erfordern Verzweigungen zurück auf den VHDL-Text oder zum Beginn des Syntheseschritts, wo ggf. Vorgabeparameter für die Synthese angepaßt werden müssen.

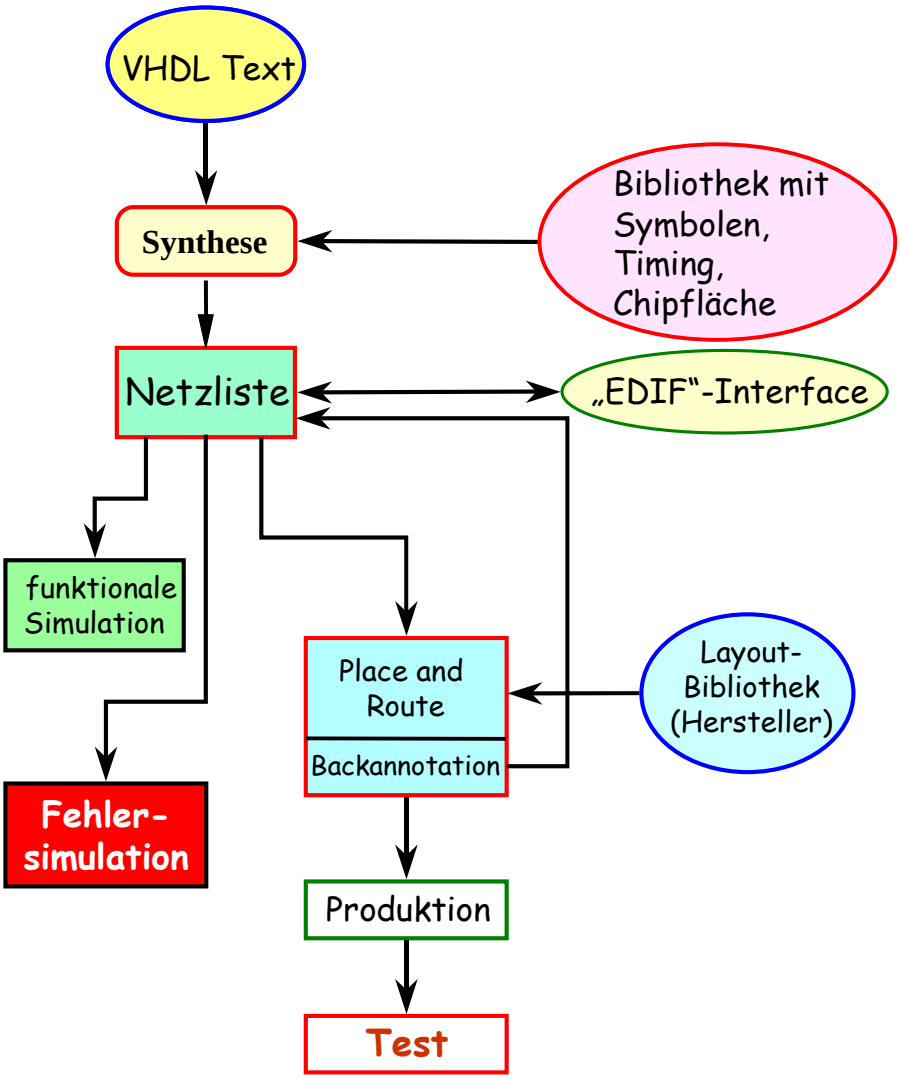



Bild 5.9: Vom VHDL-Text zum IC unter Einbezug der Technologie

In Bild 5.9 sind die bislang anhand von Bild 5.8 skizzierten Schritte ab der Synthese detaillierter und mit Bezug zur Technologie dargestellt. Für die Synthese muß demnach eine Bibliothek des Herstellers zur Verfügung stehen, in der - ähnlich wie in einem Datenbuch - die Symbole der verwendbaren Elemente (z.B. Gatter, Transfergates. Flip-Flops) und ihre Eigenschaften (z.B. Signalverzögerung vom Eingang zum Ausgang, Setup- und Hold-Zeiten) enthalten sind. Abhängig von der Zieltechnologie (Full-Custom IC, Gate-Array, Cell-Array oder FPGA)[28], enthält die Bibliothek auch Angaben über den Ressourcenverbrauch wie z.B. Chipfläche oder Anzahl der Logikzellen bei FPGAs.

Die mit der Synthese entstehende Netzliste enthält demnach bereits grundlegende Information über das Zeitverhalten des Entwurfs. Was aber hier noch fehlt, ist der Einfluß der Verbindungsleitungen, denn die Netzliste ist ein Schaltplan, in dem zwar alle benutzen Bauelemente genau mit ihren Eigenschaften spezifiziert sind, aber noch keine Layoutinformation. Diese wird erst im nächsten Schritt – Place and Route – gewonnen. Es ist trotzdem sinnvoll, mit der Netzliste eine funktionale Simulation durchzuführen. Damit können über die in Bild 5.8 angesprochene VHDL-Simulation hinaus zwei Aspekte verifiziert werden:

- Ist die Übersetzung des VHDL-Textmodells in die Schaltung fehlerfrei erfolgt?
- Hat das Zeitverhalten der Bauelemente keinen nachteiligen Einfluß auf die Funktion?

Wie Bild 5.9 des weiteren zeigt, gibt es zur Netzliste ein sogenanntes EDIF[29]-Interface. Es handelt sich hierbei um ein sehr leistungsfähiges Standardformat, das von den EDA-Werkzeugen aller führenden Hersteller erzeugt und gelesen werden kann. EDIF schließt alle drei “Sichten“ auf allen Ebenen des Y-Modells nach Bild 5.4 ein und wurde 1989 vom IEEE zum Industriestandard erhoben.

---

[28] diese ASIC-Varianten werden im nächsten Abschnitt im Überblick vorgestellt

[29] **E**lectronic **D**esign **I**nterchange **F**ormat

Je nach Technologie können für den durchgehenden Entwurf – wie er in Bild 5.9 – dargestellt ist, nicht alle benötigen Werkzeuge in einer geschlossenen Umgebung zur Verfügung gestellt werden. Das trifft z.B. auf das Placement und Routing für Full-Custom ICs oder komplexe Standardzellenentwürfe zu. Diese Aufgabe ist so komplex, daß oft eine Auslagerung auf sehr leistungsfähige Workstations erfolgen muß. Dazu kann das EDIF-Format verwendet werden. Für das Placement und Routing sind – wie in Bild 5.9 – angedeutet weitergehende Bibliotheksinformationen des Herstellers bezüglich der Zieltechnologie erforderlich. Bei Full-Custom ICs oder Standardzellenentwürfen muß neben der Geometrie der einzelnen Elemente auf der Siliziumoberfläche auch ein Satz von Entwurfsregeln (engl.: Design Rules) zur Verfügung gestellt werden. Denn auf dem Silizium sind, ähnlich wie auf einer Leiterplatte, z.B. Abstandregeln einzuhalten, die Toleranzen bei der Herstellung berücksichtigen, sowie Vorgaben über die Mindestbreite von Leitungen.

Wenn FPGAs als Zieltechnologie gewählt werden, kann in der Regel in einer PC-basierten vollständigen Entwurfsumgebung gearbeitet werden. Da hier die Zahl der Freiheitsgrade bei Plazierung und Verdrahtung – wie im folgenden Anschnitt noch näher ausgeführt - drastisch eingeschränkt ist, wird weitaus weniger Rechenleistung benötigt. Alle führenden FPGA-Hersteller liefern mittlerweile kostengünstige und leistungsfähige PC-basierte Entwurfsumgebungen, mit denen alle Schritte vom VHDL-Text bis zum Test der fertigen Schaltungen durchgeführt werden können. Als Beispiel wir im folgenden Abschnitt kurz auf die von der Fa. ALTERA bereitgestellte Software QUARTUS II eingegangen.

Nach dem Plazieren und Verdrahten geht der Entwurf – wie in Bild 5.9 zu sehen ist – noch nicht in Produktion, sondern der wichtige Schritt „Backannotation" wird vorher für alle Schaltungsarten ausgeführt. Hierbei wird aus der fertig verdrahteten Schaltung die Netzliste zurückgewonnen, wobei die Verdrahtungsinformation mit einfließt. Das wichtigste sind hierbei die zusätzlich entstandenen Verzögerungen. Da bei der heutigen CMOS-Technologie – wie in Kapitel 4 beschrieben – die Schaltgeschwindigkeit der Transistoren bis in den Picosekundenbereich vorgedrungen ist wird der Hauptteil der Verzögerungen durch die Verdrahtung verursacht. Es ist deshalb unabdingbar die Funktion der Schaltungen nach der Verdrahtung nochmals zu verifizieren. Erst wenn hierbei keine Fehler auftreten, kann die Produktion gestartet werden, die zunächst Muster für den physikalischen Test in der realen Anwendungsumgebung bereitstellt. Die Unterschiede zwischen den verschiednen ASIC-Arten sind an dieser Stelle naturgemäß groß. Beim FPGA genügt – wie im nächsten Abschnitt noch näher beschrieben - ein „Download" über die Druckerschnittstelle des PC, während z.B. bei einem Full-Custom IC oder einer Standardzelle jeweils ein vollständiger Maskensatz für die Halbleiterdotierung und die Metallisierung – siehe Kapitel 4 – erforderlich ist. Während die beim FPGA benötigte Schnittstellen Hardware für rund 5 € hergestellt werden kann, kostet ein Maskensatz zwischen 50.000 ... 250.000 €. Zu den hohen Kosten kommt hinzu, daß Fehler nicht „reparierbar" sind und in der Regel die Herstellung eines neuen Maskensatzes erfordern.

FPGAs hingegen sind je nach Technologie beliebig oft „reprogrammierbar", so daß Fehler in Sekundenschnelle und praktisch ohne Kosten repariert werden können. Nicht zuletzt dieser Aspekt hat in jüngster Zeit zu einem enormen Wachstum FPGA-basierte Schaltungsentwürfe geführt, und es ist davon auszugehen daß dieser Trend sich mit noch steigender Tendenz fortsetzt. Für FPGAs spricht auch noch der Wegfall der in Bild 5.9 eingetragenen „Fehlersimulation". Diese ist für alle ASICs, die die Mitwirkung des Halbleiterherstellers erfordern unabdingbar. Denn es geht hierbei darum, einen Satz sogenannter „Testvektoren" zu generieren, die an Eingangspins der Schaltung anzulegen sind, während die Reaktionen der Schaltung auf diese - auch „Stimuli" genannten Signale - mit einer Sollwerttabelle verglichen werden, die aus einer „Gutsimulation", d.h. der Simulation einer fehlerfreien Schaltung stammt. Es ist wichtig sich darüber klar zu sein, daß an dieser Stelle nicht die Funktion der Schaltung im Vordergrund steht, die ja im Verlauf des Entwurfs schon mehrfach si-

mulativ verifiziert wurde, sondern es geht darum, Fehler aller Art aufzudecken, die bei der Herstellung auftreten können. Dazu zählen z.B. defekte Transistoren durch Kristallfehler, fehlerhafte Dotierung etwa durch eine verunreinigte Maske, Leitungsunterbrechungen oder Kurzschlüsse sowie fehlerhafte Durchkontaktierungen zwischen verschiedenen Metallisierungsebenen.

Da jede einzelne Schaltung aus der Produktion dem Test unterzogen werden muß, ist die Zahl der Testvektoren begrenzt – der Durchlauf pro Chip sollte im Millisekundenbereich liegen.

Auf den ersten Blick scheint die Aufgabe einerseits aufgrund der hohen Anzahl von Fehlerquellen und anderseits aufgrund der Tatsache, daß viele interne Knoten der Schaltung von außen – d.h. über die Pins nicht direkt erreichbar sind – praktisch unlösbar. In langjähriger Praxis hat sich jedoch gezeigt, daß mit einem sehr einfachen Fehlermodell, nämlich dem Haftfehlermodell, bei dem immer dann die Entdeckung eines Fehlers angenommen wird, wenn ein Knoten sich nicht „bewegt“, d.h. daß er entweder auf einem „0“- oder auf einem „1“-Pegel festhängt, alle Fälle hinreichend gut erfaßt werden können. Jetzt besteht „nur“ noch die Aufgaben, alle Knoten von außen beobachtbar zu machen. Aufgrund der enormen Anzahl wäre Anfügen zusätzlicher Pins nur zu Testzwecken absolut nicht machbar. Die heute meist praktizierte Lösung ist dagegen der Einbau – d.h. die Integration - eines sehr langen parallel ladbaren Schieberegisters, über das mit einem einzigen Pin auch sehr lange Testvektoren an die zu untersuchenden Knoten gebracht und über einen weiteren Pin die Reaktionen seriell ausgelesen werden können. Diese Vorgehensweise ist als SCAN-Path-Technik bekannt. Andere Begriffe dafür sind z.B. Boundary Scan Technik oder die Abkürzung JTAG[30]-Interface. Die JTAG Standardisierung wird im nächsten Abschnitt noch im Zusammenhang mit der FPGA-Programmierung angesprochen. Sie ist nicht ohne Grund in viele Bereiche der Digitaltechnik eingeflossen. Denn durch die einfache Erreichbarkeit praktisch aller internen Knoten einer Schaltung über nur zwei Pins ergeben sich vielfältige Möglichkeiten, die über den Test hinaus auch zur Emulation genutzt werden können. Die Scan-Path-Technik wird wegen ihrer weitreichenden allgemeinen Bedeutung in der integrierten Schaltungstechnik – d.h. über die Mikrorechnertechnik hinausgehend - in weiterführenden Modell-Vorlesungen z.B. „Integrierte Signalverarbeitungssysteme“ detailliert behandelt. Hier werden auch Teststrategien vertieft, die zum Ziel haben, mit einer möglichst kleine Anzahl von Testvektoren eine entworfene Schaltung zu einem „hohen Prozentsatz“ testbar zu machen. Die Komplexität heutiger integrierter Digitalschaltungen macht einen vollständigen Test, bei dem nach den obigen Ausführungen jeder Knoten einmal „bewegt“ werden müßte, um festzustellen ob er „hängt“, unmöglich. Um die Problematik zu verdeutlichen, vergegenwärtige man sich, daß integrierte Digitalschaltungen, selbst wenn sie keine Mikrorechnerkerne einschließen, über mehrere hundert Flip-Flops verfügen. Diese können insgesamt $2^{Anzahl}$ Zustände speichern, die in einem vollständigen Test alle eingenommen werden müßten. Wenn die Anzahl der FFs nur 200 wäre, müßten somit über $1{,}6 \cdot 10^{60}$ Zustände überprüft werden. Die Fehlersimulation hat deswegen die Aufgabe mit Rechnerunterstützung und „Intuition“ den richtigen Kompromiß zu finden. Die Rechnerhilfe besteht dabei im wesentlichen darin, daß das entsprechende Programm an zufällig gewählten Stellen der Schaltung einzelne Haftfehler in die Schaltung einbaut, und dann prüft ob jeder dieser Fehler mit dem Testvektorsatz gefunden werden kann. Falls nicht ist der Designer am Zug und muß auf der Basis der Kenntnis seines Entwurfs gezielt weiter Testvektoren hinzufügen und natürlich den Erfolg überprüfen. Es ist leicht vorstellbar daß diese Prozedur auch bei hoher Rechenleistung sehr lange, d.h. Tage oder gar Wochen dauern kann, bis schließlich ein statistisch verläßliche Fehlerentdeckungsgrad, der nahe an 90% liegen sollte erreicht ist.
Obwohl Fehlersimulation und Testvektorerzeugung mit dem Entwurf direkt nichts zu tun haben, fallen diese Aufgaben dennoch dem Schaltungsentwickler zu, weil kein anderer sie besser lösen kann. In der Praxis kann das bedeuten, daß bei komplexen Schaltungen mehr als die Hälfte der

[30] **J**oint **T**est **A**ction **G**roup

Entwicklungszeit allein für den Test aufgewendet werden muß. Nicht zuletzt dieser hohe Aufwand hat zu wachsender Beliebtheit von FPGAs geführt, für die der Halbleiterhersteller den Test ja bereits durchgeführt hat und somit garantiert fehlerfreie Hardware liefert.

An dieser Stelle sei aber auf eine Falle hingewiesen, in die man als Entwickler schnell geraten kann, wenn man sich von vornherein alle Überlegungen hinsichtlich Test und Testbarkeit einer Schaltung erspart. Der Begriff „Design for Testability“ besagt in diesem Zusammenhang, daß es wichtig ist, von Anfang einen möglicherweise später benötigten Test im Auge zu behalten. Trifft man diese Vorsorge nicht, kann eine Schaltung entstehen, die gar nicht testbar ist. Wird dann überraschend der Test trotzdem benötigt, weil man aufgrund wachsender Stückzahlen vom FPGA auf z.B. eine Standardzelle umschwenken muß, dann fallen meist grundlegende Änderungen an, die den Entwickler wieder in eine frühe Phase des Designs zurückversetzen. In der Regel wird er dabei weit mehr Zeit verlieren, als wenn er von Beginn an – auch wenn die erste Zieltechnologie ein FPGA war – stets die Testbarkeit bei jedem Entwurfsschritt sichergestellt hätte.

### 5.1.5 ASIC Technologien und „Entwurfsstile“

In diesem Abschnitt wird ein kurzer anschaulicher Abriß über die in Bild 5.1 schon vorgestellten ASIC-Technologien und die dazugehörigen Entwurfsstile gegeben. Der Schwerpunkt wird dabei auf die FPGAs gesetzt, da sie aufgrund ihrer hohen Leistungsfähigkeit und der schnellen und einfachen Konfigurierbarkeit für die verschiedenen Arbeitsfelder des Elektroingenieurs die größte Bedeutung haben. Aus heutiger Sicht ist davon auszugehen, daß diese Bedeutung wächst und daß der Ingenieur für sein spezielles Arbeitsgebiet den Entwurf anwendungsspezifischer Lösungen beherrschen oder zumindest verstehen muß. D.h. nicht nur der Digitaltechniker und Rechnerspezialist, sondern auch der Kommunikations- Automatisierungs- und der Energietechniker wird sich der neuen Möglichkeiten bedienen, die es ihm gestatten werden, besser problemangepaßte, leistungsfähigere und preisgünstigere Lösungen zu schaffen, als es mit Standardbausteinen auf mehr oder weniger umfangreichen Leiterplatten jemals möglich wäre.

#### 5.1.5.1 Vollkundenschaltung (Full Custom IC)

Der Entwurf einer Vollkundenschaltung stellt zum einen hohe Ansprüche an den Entwickler bzw. das Entwicklungsteam und ist zum anderen der teuerste Weg zur anwendungsspezifischen Schaltung. Der Vorteil ist ein flächen- und leistungsoptimiertes Ergebnis, das in hohen Stückzahlen produziert werden kann und mögliche Konkurrenz lange Zeit aus dem Feld schlägt.

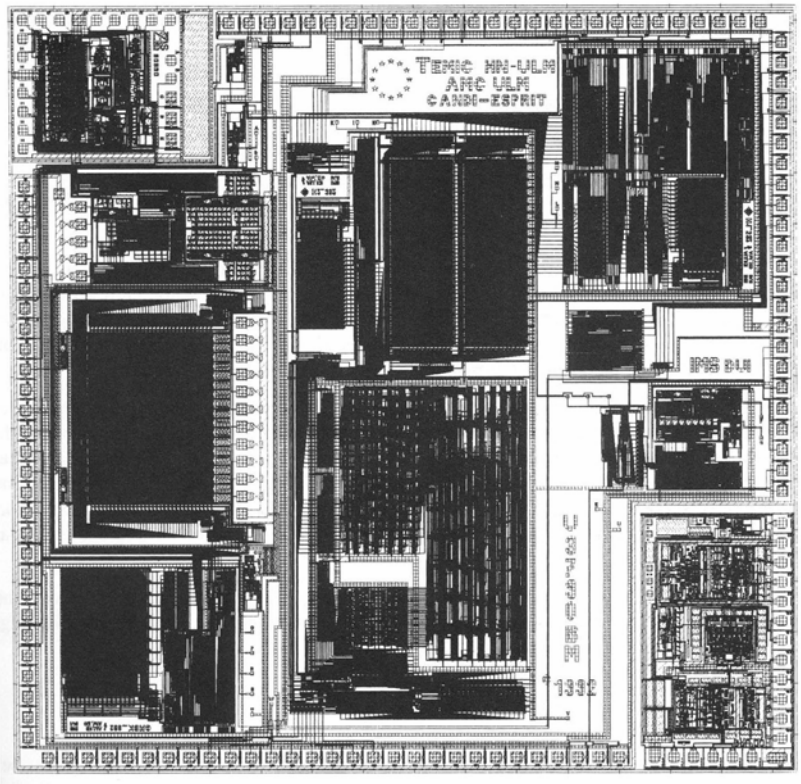


Bild 5.10: Typisches Chipbild einer Vollkundenschaltung

Der vollkundenspezifische Entwurf bietet die meisten Freiheitsgrade. Alle Strukturen werden einzeln ausgelegt und können bezüglich der gegebenen Randbedingungen optimiert werden. D.h. es wird die Dimensionierung und Anordnung der

einzelnen Transistoren sowie der Leiterbahnen im Detail durchgeführt. Dabei ist die Verwendung beliebiger geometrischer Formen erlaubt, also auch nicht geradlinig begrenzter Strukturen. Diese Freiheitsgrade erlauben die Ausnutzung aller Möglichkeiten, die eine Technologie.

### Die Charakteristika vollkundenspezifischen Entwurfs im Überblick

#### hohes Optimierungspotential

Da jede einzelne geometrische Struktur explizit zu entwerfen ist, kann sie den jeweils erforderlichen Gegebenheiten optimal angepaßt werden. Insbesondere im Hinblick auf höchste Pakkungsdichte, d.h. optimale Siliziumausnutzung, ist dies von Interesse. Voraussetzung ist die Verfügbarkeit von bezüglich der Technologie geschultem Personal.

#### hohe Technologietreue

Die detaillierte Realisierung der einzelnen Strukturen ermöglicht die Berücksichtigung komplexer technologischer Randbedingungen. In der Praxis können so einzelne „Design Rules“ gezielt ignoriert werden, was bei standardisierten (worst-case-orientierten) Verfahren unmöglich wäre.

#### hohes Fehlerrisiko

Wegen der Vielzahl der zu behandelnden Strukturen, die eine Vielfalt von Randbedingungen, mit teilweise komplizierten Wechselwirkungseffekten, einzuhalten haben, ist klar, daß damit ein hohes Fehlerrisiko verbunden ist. Eine rein manuelle Sicherstellung eines korrekten Entwurfs wird generell unmöglich sein, schon gar nicht durch visuelle Überprüfung eines fertigen Layouts, z.B. auf Einhaltung aller geometrischen Randbedingungen.

#### hoher Erstellungs- und Korrekturaufwand

Die Behandlung jeder einzelnen Struktur eines Entwurfs führt dazu, daß bei größeren Objekten ein hoher Zeitaufwand für Anordnung und Optimierung erforderlich wird. Hinzu kommt, daß beim Auftreten von Fehlern, weiterer Zeitaufwand für Korrekturen anfällt. Gewöhnlich führen Fehler zu teuren „Redesigns“, weil man oft bis in sehr frühe Entwurfsstadien zurückgehen muß.

Die obigen Betrachtungen zeigen, daß vollkundenspezifischer Entwurf nur in wenigen Anwendungsbereichen sinnvoll ist, z.B. da wo analoge oder gemischt analog/digitale Schaltungen aufgrund ihrer Zielspezifikation keine standardisierten Verfahren zulassen, oder wo höchste Packungsdichte zur Realisierung der Funktionalität erforderlich ist. Des weiteren ist vollkundenspezifischer Entwurf vertretbar, wenn die zu erwartenden Stückzahlen (ein Grenze kann heute bei etwa mehreren Millionen Stück pro Jahr gezogen werden) hohe Entwurfskosten eine lange Entwicklungszeit rechtfertigen; das wäre z.B. bei universell einsetzbaren programmierbaren Standardschaltungen wie Mikroprozessoren oder Mikrocontrollern und auch DSPs gegeben.

#### 5.1.5.2 Standardzellenentwurf (Cell Arrays)

Beim Standardzellenentwurf müssen ebenso wie bei einer Vollkundenschaltung alle Schritte der Halbeiterherstellung durchlaufen werden. D.h. es entstehen – in erster Linie aufgrund der Maskenfertigung hohe Kosten, bevor ICs zum Testen zur Verfügung stehen. Das Charakteristikum des Standardzellenentwurfs sind Einschränkungen der Freiheitsgrade durch Vorgabe standardisierter Funktionseinheiten, wie sie häufig bei digitalen Schaltungen vorkommen. Man kann auf vorentwor-

fene Grundlayouts zurückgreifen, die in „Zellbibliotheken“ bereitgestellt werden. Die Layouts dieser Zellen sind hinsichtlich funktionaler Korrektheit und der elektrischen Eigenschaften verifiziert. Sie können aus den Bibliotheken wie aus einem Baukasten entnommen und auf dem Silizium nach einfachen Regeln zusammengesetzt werden.

Anhand von Bild 5.11 lassen sich die wesentlichen Merkmale des Standardzellenentwurfes be-

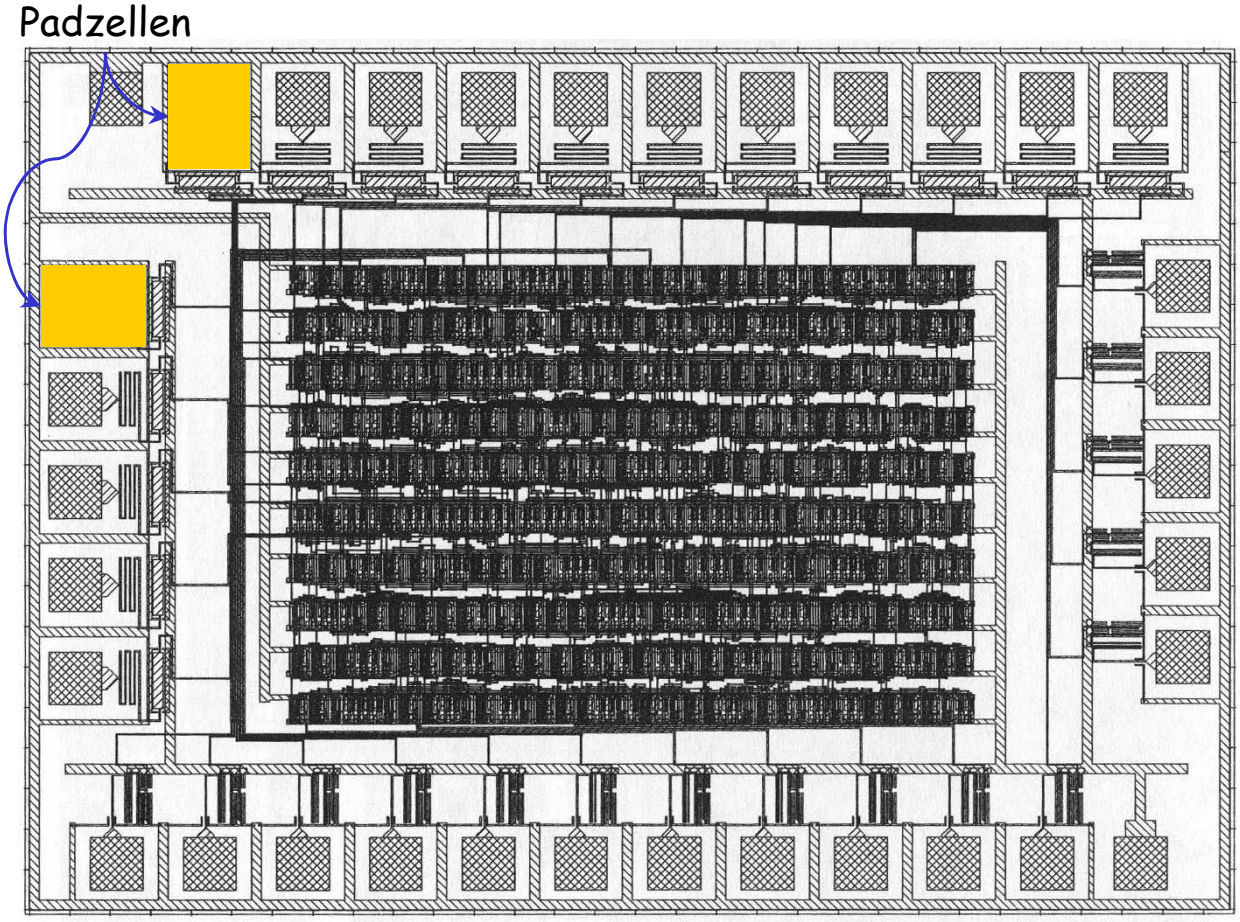



Bild 5.11: Typisches Chipbild eines Standardzellen ICs

schreiben. Man erkennt eine ausgeprägte Zeilenstruktur, die sich dadurch ergibt, daß für das Layout aller Bibliothekszellen immer rechteckige Zellen identischer Höhe verwendet werden, die an zwei gegenüberliegenden Seiten über festgelegte Positionen für Versorgungsleitungen und Ein- bzw. Ausgangsanschlüsse verfügen. An den beiden verbleibenden Zellseiten ist eine freie Anordnung von Anschlüssen zugelassen.

Diese Art der Standardisierung der Zellen ermöglicht es, komplexe Funktionen aufzubauen, indem Reihen von Zellen durch Aneinanderfügen ohne zusätzliche Verbindungsstrukturen gebildet werden. Der Entwurf von Schaltungen aus Standardzellen besteht somit aus der Auswahl von Zellen, die in Reihen angeordnet werden, der Plazierung der einzelnen Reihen in vorgegebenen Rasterabständen auf der Chipoberfläche und relativ der Verdrahtung.

## Der Standardzellentwurf ist folgendermaßen charakterisiert:

### einfache Zellanordnung

Die der Zahl „Design Rules“ ist gegenüber dem vollkundenspezifischen Entwurf drastisch reduziert. Es sind praktisch nur noch Regeln zu beachten, die den Zellrand und die Verdrahtung betreffen. Aufgrund der Vorgabe, daß immer Zellen fester Höhe – wie Bild 5.12 verdeutlicht -

direkt aneinander zu fügen sind, entsteht die charakteristische - in Bild 5.11 gut erkennbare - Zellreihenstruktur, die geometrisch nur wenige Randbedingungen einhalten muß, wodurch eine hoher Automatisierungsgrad möglich wird. Die stark reduzierte Entwurfsaufgabe führt zu schnellen Ergebnissen. Das betrifft sowohl die Zeit für die Ersterstellung als auch für gegebenenfalls erforderliche Korrekturen.

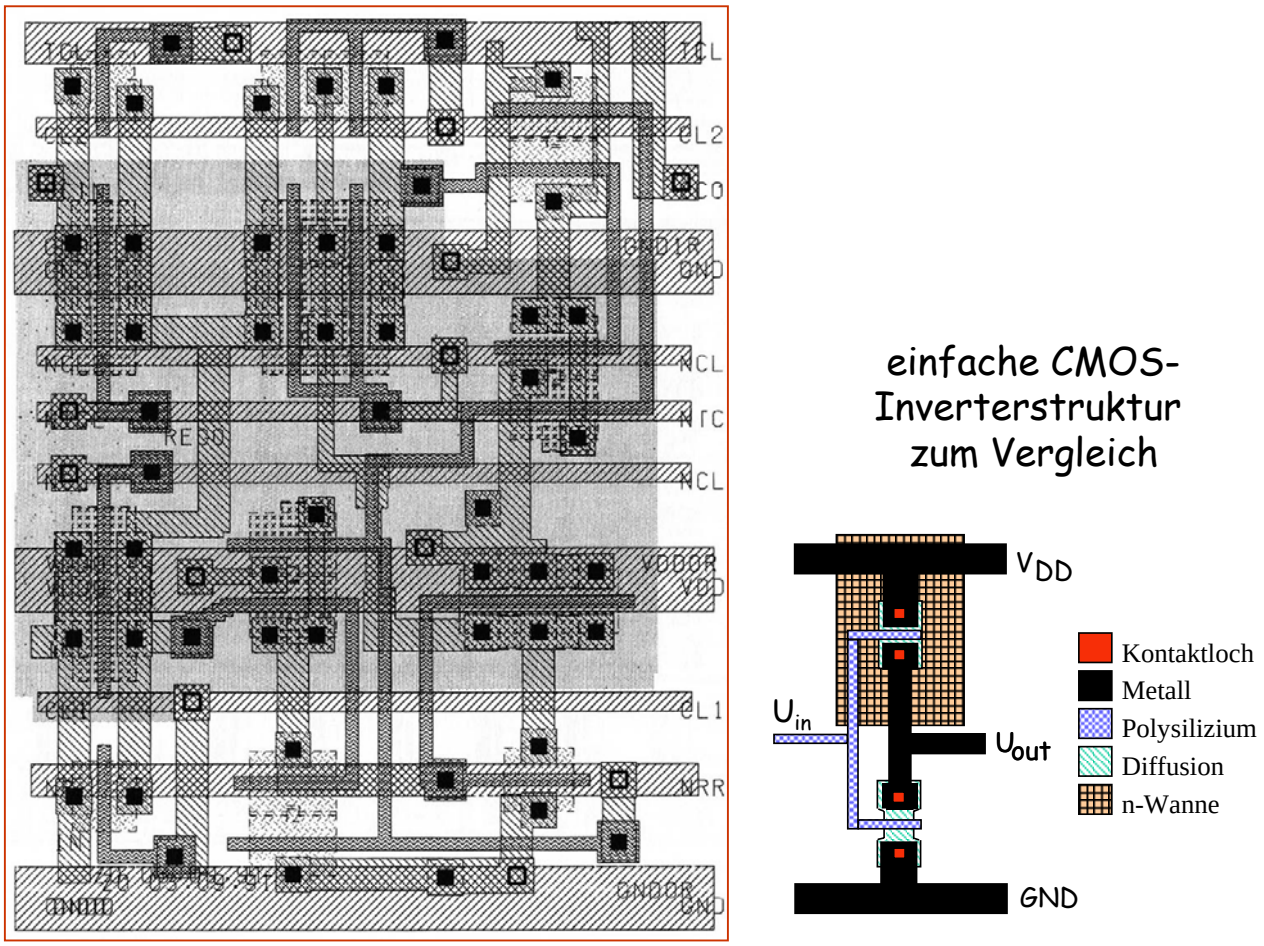



Bild 5.12: Einzelheiten einer Standardzellenstruktur im Vergleich mit einem CMOS-Inverter

## geringes Fehlerrisiko durch verifizierte Zellen

Geht man davon aus, daß sowohl die Zellen in der Bibliothek, als auch die Regeln, die zwischen den Zellen einzuhalten sind, verifiziert sind und ein korrektes Schaltungsverhalten garantieren, ist das Fehlerrisiko im Vergleich zum vollkundenspezifischen Entwurf erheblich reduziert. Was als Restrisiko verbleibt, ist die Sicherstellung der korrekten Schaltungsfunktion unter Einbeziehung der Laufzeit- und Verzögerungseffekte durch die erzeugten Verbindungsstrukturen.

## wenig Optimierungsmöglichkeiten

Da die Zellen auf Gatterebene festliegen, fehlt eine funktionale Strukturierung. Dies hat zur Folge, daß eine Optimierung der Zellanordnung, mit dem Ziel einer möglichst günstigen Verdrahtung, schwierig ist. EDA-Werkzeuge sind gewöhnlich nicht in der Lage, eine wirklich gute Anordnung zu finden. Auch beim manuellen Eingriff kann aufgrund des starren Anordnungsschemas immer noch eine ungünstige Zellanordnung verbleiben.

## Verschwendung von Verdrahtungsfläche

Wenn die Anordnung der Zellen in den Reihen nicht geschickt gelöst werden kann, wirkt sich das nachteilig auf den Verdrahtungsaufwand aus. Bei einfachen Werkzeugen für Standardzellentwurf ist ein Verdrahtungsflächenanteil von 60-70 % der gesamten Chipfläche nicht ungewöhnlich.

## Der Chiprand (Padzellen) muß separat behandelt werden

Die Anforderungen an die elektrischen Eigenschaften der Peripheriezellen, die die Anschlüsse nach Außen bilden, und die daraus resultierende Flächenaufteilung sind mit dem Standardzellentwurfsstil nicht verträglich. Daher ist in jedem Falle eine gesonderte Behandlung des Chiprandes und der Verdrahtung der Reihen mit den Padzellen erforderlich.

## einfacher, automatisierter Entwurf

Standardzellen sind im Hinblick auf eine einfache Automatisierbarkeit entstanden. Bei kommerziellen Entwurfssystemen für Standardzellen gibt es immer noch gravierende Nachteile, wie z.B.

- es gibt keine Garantie für vollständige Verdrahtung
- der Verdrahtungsflächenanteil kann unvertretbar hoch werden
- die Chipflächenausnutzung kann gering sein

Der Einsatz von Standardzellen ist immer dann empfehlenswert, wenn man einer kurzen Entwicklungszeit der Vorrang gegenüber einer hohen Technologieausnutzung geben kann. In der Praxis des Entwurfs digitaler mikroelektronischer Schaltungen wird der Standardzellentwurf heute am häufigsten verwendet, wenn sich die Stückzahlen im Bereich einige Hunderttausend bis eine Million pro Jahr bewegen.

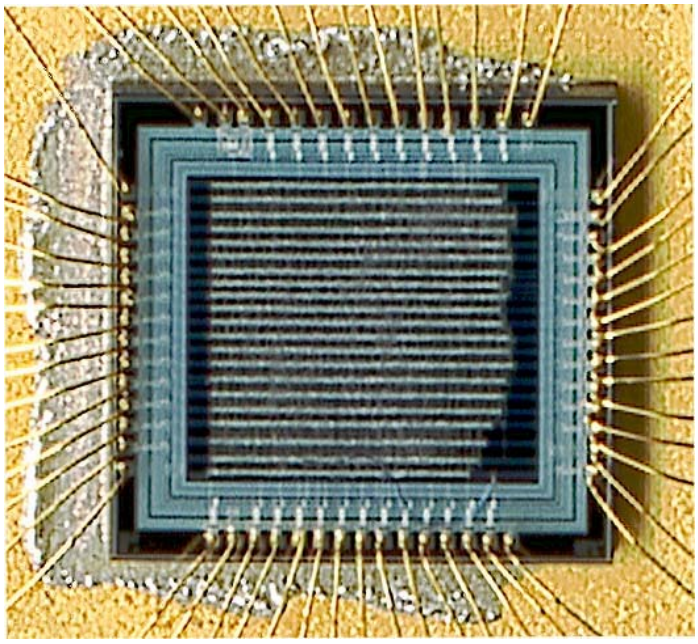


Bild 5.13: FSK-Modem für die Powerline-Kommunikation in Form einer Standardzelle (realisiert im Rahmen von „Europractice“)

Bild 5.13 zeigt das Beispiel eines Standardzellenchips, der Mitte der 90-er Jahre am IIIT realisiert wurde. Nähere Informationen zu den implementierten Funktionen sind im Anhang zu finden. Aus heutiger Sicht ist die Komplexität von ca. 25.000 Transistoren gering.

Neben sogenannten „Primitiven“ – wie z.B. Gattern, Flip-Flops - enthalten die heutigen Standardzellen-Bibliotheken der führenden Hersteller auch „Makrozellen“. Diese sind praktisch ausschließlich VHDL-Modelle auf der Register-Transfer-Ebene. Man findet z.B. „generische[31]“ Beschreibungen von ALUs, Registerbänken, Steuerwerken oder Multiplexern bis hin zu kompletten Mikrorechnerkernen (MP, MC und DSP). Mit heutigen Zellbibliotheken können vollständige, komplexe Systeme auf Silizium in kurzer Zeit erstellt werden. Dabei können auch analoge Schaltungsteile einbezogen werden, wenn sie als Bibliotheks-

[31] hierbei wird mit Parameterübergabe erst bei der Instantiierung z.B. die Bitbreite festgelegt

elemente verfügbar sind. Das ist z.B. der Fall für A/D- und D/A-Wandler sowie Operationsverstärker. Voraussetzung ist natürlich, daß alle Analogfunktion mit demselben CMOS-Halbleiterprozeß hergestellt werden können auf dem die Digitalfunktionen basieren. Eine Nutzung dieser Möglichkeit erfolgte 1997 im Rahmen eines Forschungsprojekts des IIIT mit der Fa. ABB. Es wurde ein sogenanntes „Mixed-Signal-ASIC“ realisiert, daß ein komplettes Powerline-Modem auf einem Chip darstellt.

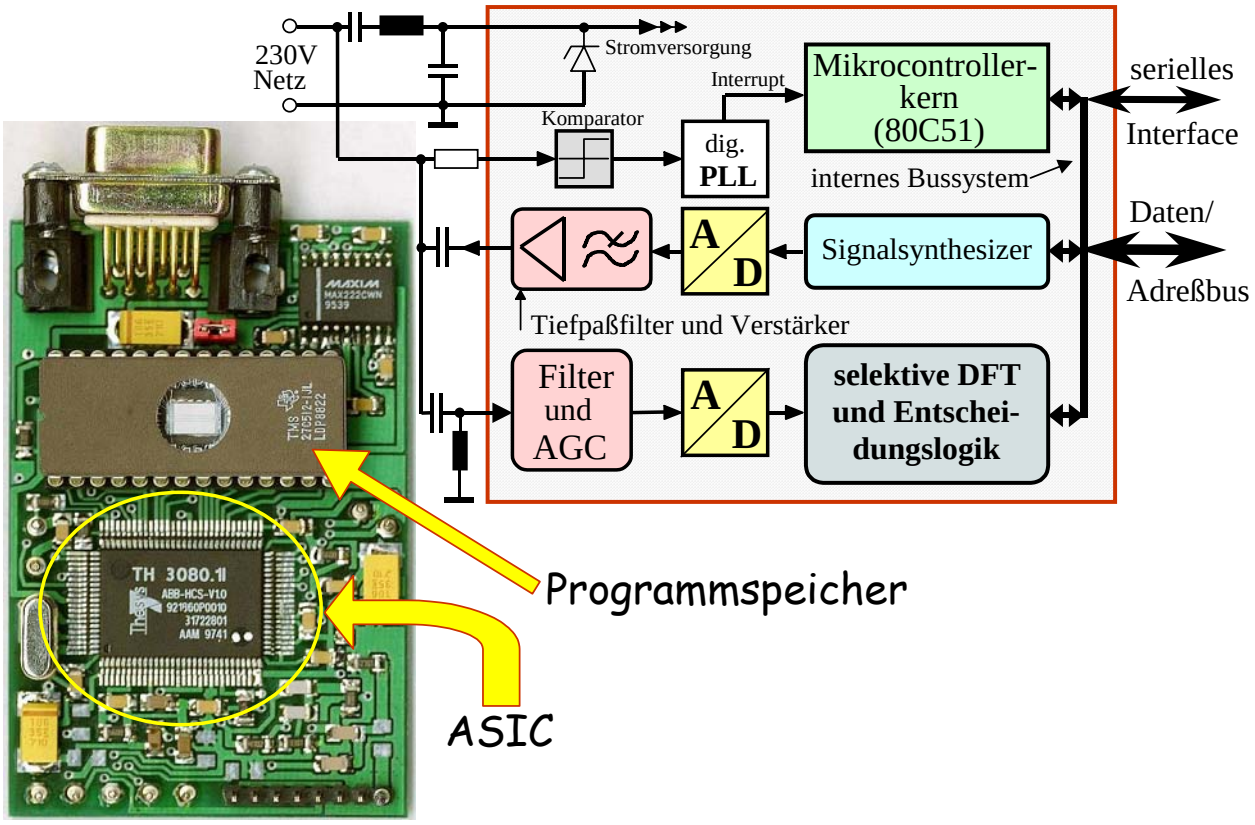



Bild 5.14: Mixed-Signal-ASIC, realisiert als Standardzelle

Anhand des Blockschaltbildes in Bild 5.14 sieht man daß die analogen Schaltungsfunktion im wesentlichen Signalfilterung und Verstärkung sowie A/D- und D/A-Wandlung umfassen. Der mit „selektive DFT und Entscheidungslogik bezeichnete Block beinhaltet unter anderem eine MAC-Einheit mit 8 Akkumulatoren (8-facher digitaler Korrelator). Der Signalsynthesizer besteht im wesentliche aus Adreßzählern und Abtastwertespeichern, wo die zu erzeugenden Signalformen abgelegt sind. Von hoher Bedeutung ist darüber hinaus die Integration des 8051-Mikrocontrollerkerns, der neben Aufgaben der Systemsteuerung und des Datentransfers mittels seiner seriellen Schnittstelle die einfache Bedienung der gesamten Hardware über seinen Befehlssatz gestattet.

Dabei kommt das bewährte Prinzip des Memory–Mapping zur Anwendung, d.h. der Spezialfunktionsregistersatz des 8051 wird entsprechend erweitet. Wegen der großen Zahl der zusätzlich benötigten Register reicht der im Standard 8051 vorgesehene Platz im internen Datenspeicher nicht mehr aus. Das Problem wurde hier durch Auslagerung und den externen Adreßbereich (XDATA) gelöst, wobei natürlich die entsprechende Hardware in Form eine RAM mitintegriert wurde.

Tabelle 5.1 zeigt exemplarisch einen Ausschnitt der sogenannten „Include“–Datei, die z.B. im „Header“ eines Assemblerprogramms aufgeführt werden muß, damit die über den 8051-Kern hinausgehende Hardware angesprochen werden kann. Man sieht, daß in einfacher Weise sowohl analoge als auch digitale integrierte Hardware mit dem Memory-Mapping-Konzept bedient werden kann.

```
;Mixed-Signal-ASIC mit 8051 Prozessor und integrierter Peripherie
;Exemplarischer Auszug aus der INCLUDE-DATEI
;Zugriff auf Akkumulator-Werte für die Symbole 0...3 (Read Only)
Symbol0_L       XDATA   0FF80H
Symbol0_H       XDATA   0FF81H
Symbol1_L       XDATA   0FF82H
Symbol1_H       XDATA   0FF83H
Symbol2_L       XDATA   0FF84H
Symbol2_H       XDATA   0FF85H
Symbol3_L       XDATA   0FF86H
Symbol3_H       XDATA   0FF87H
;Betriebsarten-Auswahl für das Modem
Modem_Mode      XDATA   0FF02H
;Steuerregister der Filter und der automatischen Verstärkungsregelung AGC
OpAmp_Ctrl      XDATA   0FF08H
;Verstärkungseinstellung der Operationsverstärker
AGC_Gain_slow   XDATA   0FF07H
AGC_Gain_fast   XDATA   0FF0BH
;Signalformspeicher
WAV_F0B         XDATA   0F000H
WAV_F0E         XDATA   0F1F3H
WAV_F1B         XDATA   0F200H
WAV_F1E         XDATA   0F3F3H
WAV_F2B         XDATA   0F400H
WAV_F2E         XDATA   0F5F3H
WAV_F3B         XDATA   0F600H
WAV_F3E         XDATA   0F7F3H
WAV_QUAB        XDATA   0F800H
WAV_QUAE        XDATA   0F9F3H
```

Tabelle 5.1: Ausschnitt zur SFR-Definition des Mixed–Signal–ASICs

### 5.1.5.3 Gate–Arrays

Über die Standardisierung von Zellen hinaus, ist auch eine Standardisierung der Chipfertigung selbst denkbar. Das wird bei Gate–Arrays praktiziert, wobei eine Vorfertigung der Siliziumscheiben, die alle Dotierungsschritte beinhaltet, realisiert ist. Dabei werden reguläre Anordnungen (Arrays) mit „Bauteilrümpfen“ vorgefertigt, die dann mittels weniger Maskenschritte, die die Verdrahtung durchführen, für die gewünschte Schaltungsfunktion „personalisiert“ werden. Die vorgefertigten Chips nennt man „Master“. Beim Gate–Array besteht der Master aus Feldern mit Transistoren, die noch unverbunden sind, und aus Verdrahtungskanälen, die zwischen den Feldern angeordnet sind. Die typische Topologie ist dabei eine zeilenweise Anordnung, in der sich Transistorfelder und Verdrahtungskanäle abwechseln, oder eine matrixartige Anordnung von Transistorfeldern, die horizontal und vertikal von Verdrahtungskanälen getrennt werden.

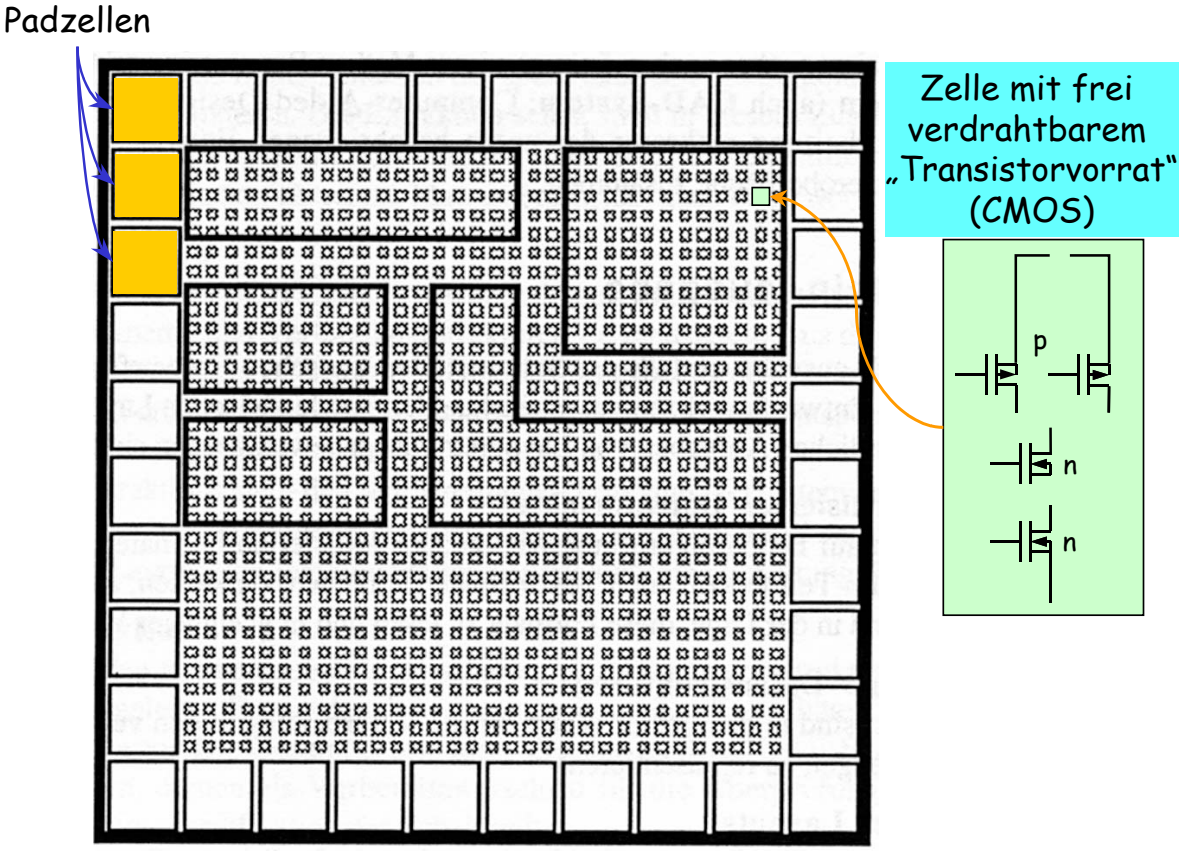



Bild 5.15: Gate-Array mit Sea-of-Gates–Struktur

Bild 5.15 zeigt einen Gate–Array–Master, der die „Sea–of-Gates“–Philosophie verfolgt. Sie unterscheidet sich prinzipiell nur gering von der ursprünglichen Zeilenanordnung. Die Sea-of-Gates–Struktur ist durch moderne Halbleitertechnologie für sehr hohe Integrationsdichten möglich geworden. Weil hierbei die Abmessungen der Transistoren drastisch reduziert werden konnten, hat sich eine Master-Topologie entwickelt, bei der der gesamte Chip mit einem dichten, zweidimensionalen Netz von Transistoren überzogen ist, zwischen denen relativ wenig Verdrahtungsraum bereitgehalten werden muß. Diese Topologie hat selbstverständlich Folgen für den Entwurf. Einerseits kann man nun Zellverdrahtungen mit recht hoher Komplexität erstellen, da nahezu beliebige zweidimensionale Verdrahtungsstrukturen möglich sind, insbesondere auch eine funktionale Orientierung. Bezüglich der Ausdehnung und der Form hat man lediglich durch das Transistorraster geringe Einschränkungen. Aufgrund der Regularität lassen sich Verdrahtungsstrukturen im Transistorraster verschieben, was insgesamt sehr flexible Layouts ermöglicht. Die Sea-of-Gates-Philosophie hat gegenüber der weitaus starreren Zeilenstruktur die besseren Zukunftsaussichten, weil sie die Vorteile des flexiblen Entwurfs mit kurzen Fertigungszeiten verbindet.

Gate-Arrays aller Art werden generell nur für digitale Anwendungen eingesetzt. Der Entwurf geht in der Regel von einer Schaltungsbeschreibung auf Register–Transfer- oder Logikebene aus und besteht letztlich aus der Erstellung von Verdrahtungsmustern, die zunächst die Transistoren innerhalb der Felder zu logischen Primitiven (Inverter, Gatter, Flip-Flops) verbinden, und anschließend die Gesamtfunktion der Schaltung durch Verbindung der Primitive realisieren. Es stehen leistungsfähige EDA-Werkzeuge zur Verfügung, die auf vielfältige, vorentworfene Verdrahtungsmuster aus Bibliotheken zurückgreifen können. Eine gewisse Verwandtschaft zum Standardzellentwurf ist somit gegeben.

Neben den Einschränkungen, die der Standardzellentwurf aufweist, kommen bei Gate-Arrays weitere hinzu, die sich im wesentlichen aus der starren Architektur der Master ergeben. Als Vorteile sind dagegen hoher Automatisierungsgrad, schneller Fertigungszyklus und relativ niedrige „Einmalkosten“ (für die Maskenherstellung) anzuführen.

### geringe Ausnutzung der „theoretischen Chipkapazität"

Die Vorfertigung der Master legt die theoretisch nutzbare Zahl von Transistoren auf dem Chip bereits fest. Diese Zahl kann jedoch bei Schaltungsrealisierungen praktisch nie erreicht werden. Die begrenzte Kapazität der Verdrahtungskanäle führt oft dazu, daß digitale Strukturen nicht in der wünschenswerten hohen Dichte angeordnet werden können. Man erhält relativ große Chips mit vergleichsweise geringer Komplexität.

### Mehrebenenverdrahtung verbessert die Chipausnutzung

Moderne Technologien, die bis zu 6 Metallisierungsebenen für die Interzellverdrahtung zur Verfügung stellen, können den oben genannten Nachteil mindern, da sie eine Erhöhung der Verdrahtungskapazität bringen. Bei modernen Gate-Arrays kann so eine Chipausnutzung bis über 90 % erreicht werden, allerdings zu erhöhten Maskenkosten wegen der hohen Zahl der Metallebenen.

### automatisierter Entwurf ist nötig und möglich

Da jegliche funktionale Orientierung beim Entwurfsprozeß fehlt, ist eine manuelle Durchführung praktisch ausgeschlossen. In der Tat ist ein entscheidendes Motiv für den Gate-Array–Einsatz durch die nahezu vollständig automatischen Abläufe gegeben.

### verkürzter Fertigungszyklus

Da die Herstellung der Master universell, d.h. unabhängig von jedem Entwurf erfolgt, ist die Durchlaufzeit für eine Gate–Array–Produktion signifikant geringer als z.B. für Vollkundenschaltungen oder Standardzellen, die jeweils einen vollständigen Halbleiterherstellungszyklus durchlaufen müssen. Gate–Arrays erlauben einen ausgesprochen schnellen Durchlauf von der Fertigstellung des Entwurfs bis zur Verfügbarkeit der ASICs für den praktischen Einsatz (Time-To-Market-Vorteil).

Trotz der genannten Vorteile haftet auch den Gate–Arrays der Nachteil an, daß der Gang zum Halbleiterhersteller erforderlich ist, um zum einsetzbaren Chip zu kommen. Obwohl Kosten und Durchlaufzeiten im Vergleich zu Vollkundenschaltungen oder Standardzellen erheblich geringer sind, führen auch hier Fehler in der Regel zu Redesigns, die sich zeitraubend und kostenintensiv auswirken. Eine elegante Umgehung all dieser Probleme bieten heute FPGAs, deren Komplexität durchaus mit Gate–Arrays vergleichbar ist. Der entscheidende Vorteil der FPGAs ist aber, daß sie am Arbeitsplatz des Ingenieurs – wo im Grunde nur ein „mäßig“ ausgestatteter PC erforderlich ist - ohne Mitwirkung des Halbleiterherstellers „personalisiert“, d.h. mit der gewünschten Schaltungs-

funktion versehen werden können. Zudem läßt sich dieser Vorgang bei den bevorzugten FPGA–Technologien beliebig oft wiederholen, so daß im Falle von Fehlern oder Verbesserungen so gut wie keine Zusatzkosten anfallen. Nicht zuletzt dieser Gesichtspunkt hat zu dem in jüngster Zeit beobachteten enormen Zuwachs FPGA–basierter Designs geführt. Aufgrund der raschen Weiterentwicklung der Technik in allen Bereichen entpuppt sich eine „festgenagelte“ Schaltung – selbst wenn sie programmierbare Bestandteile beinhaltet – nach kurzer Zeit als zu unflexibel oder zu langsam, um auf neue Herausforderungen zu reagieren. Bevor derart hohe Stückzahlen erreicht werden, die eine Vollkundenschaltung oder Standardzelle rechtfertigen würden, fordert der Kunde oft schon Erweiterungen, die ein Redesign erforderlich machen würden.

Mit FPGA–Lösungen kann auf solche Wünsche schnell und kostengünstig reagiert werden. Der folgende Abschnitt befaßt sich mit dieser sehr zukunftsträchtigen Thematik, mit der jeder Ingenieur der Elektrotechnik und Informationstechnik vertraut sein sollte.

#### 5.1.5.4 Vom PLD zum Field Programmable Gate Array (FPGA)

Alle heutigen FPGAs haben ihren Ursprung in einfachen programmierbaren Logikbausteinen, die unter der Bezeichnung PLD (**P**rogrammable **L**ogic **D**evice) seit über 25 Jahren auf dem Markt sind. Die ersten PLDs dienten dazu, die sogenannte Glue[32]-Logic, die eine oft sehr hohe Anzahl von Standard–TTL- oder CMOS–Schaltkreisen zwischen Mikrorechnern und ihrer Peripherie, vor allem externen Speichern erforderlich machte, zu vereinfachen. Bei Speichern wird – wie in Bild 5.37 angedeutet – oft die Decodierung von Adressen oder Adreßbereiche benötigt. Dazu sind große UND-Gatter (16...32 Eingänge) nötig. Dieser Anforderung wurde daher durch die ersten einfachen PLDs Rechnung getragen. Ein solcher Baustein wurde in Abschnitt 6.3 für den Aufbau der FSM des Mikrosequencers verwendet. In Bild 6.36 ist die gesamte resultierende Schaltung dargestellt, die auf dem Baustein GAL 16V8 in Bild 6.37 implementiert wurde. Bei der Schaltung in Bild 6.36 fällt auf, daß keine sehr breiten UND-Gatter benötigt werden – maximal 6 Eingänge genügen hier.

Der Gesamtaufbau des GAL 16V8 ist in Bild 5.16 dargestellt. Der Baustein gliedert sich in 8 Zellen, von denen jede eingangsseitig über 8 UND-Gatter der Breite 32 verfügt, deren Ausgänge auf ein 8-faches ODER–Gatter führen, das sich jeweils in den **OMLC** bezeichneten Blöcken befindet. **OLMC** bedeutet **O**utput **L**ogic **M**acro **C**ell, ein Begriff, der aus heutiger Sicht für die einfache Hardware ziemlich übertrieben erscheint – siehe Bild 5.17. Denn neben dem 8-fachen ODER enthält die OLMC in der Hauptsache ein vorderflankegetriggertes D-Flip-Flop, dem noch ein EXOR-Gatter (XOR) zur Invertierung des D-Eingangs vorgeschaltet ist. Die Konfiguration erfolgt über Multiplexer, die der Übersicht wegen in Bild 5.17 nicht eingezeichnet sind. Alle eingangsseitigen UND-Gatter können per Konfiguration mit bis zu 32 Eingangssignalen verbunden werden. So viele wären zum einen nur dann verfügbar, wenn auch alle Ausgänge des Bausteins als Eingänge konfiguriert wären und zum anderen macht ein Verbinden von mehr als 16 Signalen keinen Sinn, da dann Signale direkt und negiert UND-verknüpft würden, was immer zu dem trivialen Resultat „0“ führen würde.

Mit PLD–Strukturen nach Bild 5.16 lassen sich boolesche Ausdrücke in der disjunktiven Normalform (ODER–Verknüpfung von Mintermen) unmittelbar abbilden – vgl. Abschnitt 3.2. Mit Hilfe der Speicherelemente in den OLMCs können dann sehr effizient recht komplexe Automaten aufgebaut werden. Mit den 8 verfügbaren Flip-Flops könnten immerhin bis zu 256 Zustände kodiert werden. Für allgemeine Anwendungen (Aufbau beliebiger Digitalschaltungen) erweist sich aber die starre UND-Matrix als wenig nützlich. Es ist weitaus besser, eine hohe Zahl kleinerer „Einheiten“ z.B. in Form von Wertetabellen (**L**ook **U**p **T**ables ≡ LUT - vgl. Bild 5.20) an dieser Stelle vorzusehen.

[32] engl. Wort für Klebstoff

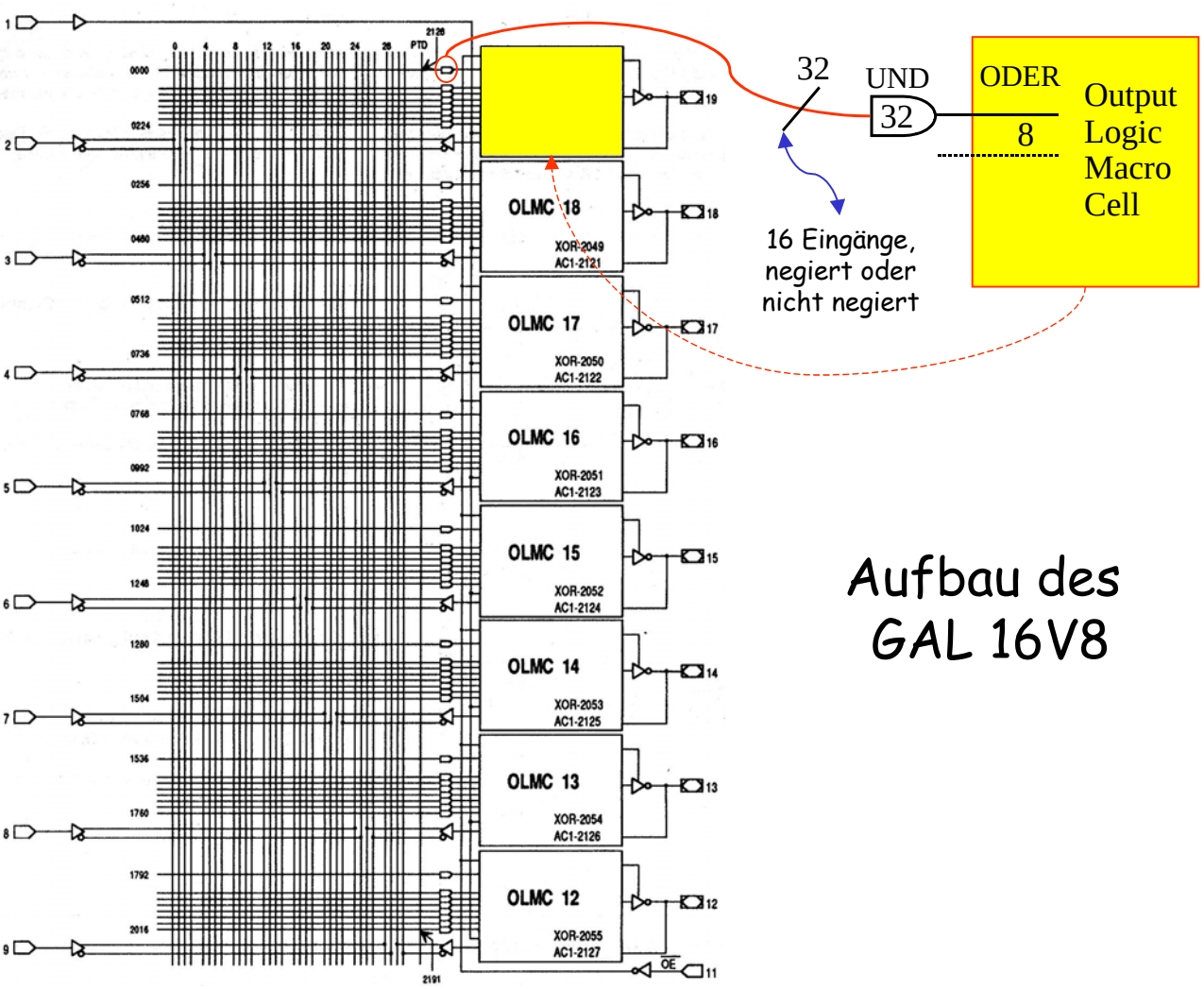



Bild 5.16: Gesamtaufbau des programmierbaren Logikbausteins GAL 16V8, bei dem die Konfigurierung mittels EEPROM-Zellen erfolgt

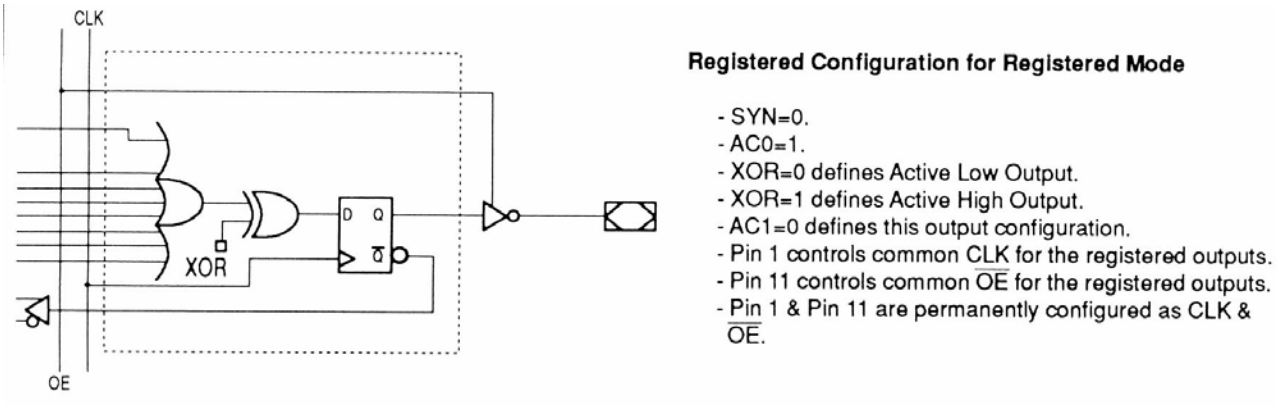



Bild 5.17: Konfiguration einer Output Logic Macro Cell

Gemäß der Philosophie – mehr und kleinere Einheiten – sind aus den sogenannten „Simple“-PLDs, zu denen das GAL 16V8 zählt, die komplexen PLDs (CPLD) entstanden. CPLDs werden von zahlreichen Herstellern in einer schier unübersehbaren Vielfalt angeboten. Bei genauerem Hinsehen lassen sich aber rasch gewisse bewährte Grundstrukturen ausmachen, die sich in mehr oder weniger modifizierter Ausgestaltung immer wieder finden. Es genügt daher für diese Einführung exemplarisch und stellvertretend für viele andere einen Baustein aus der MAX 3000-Reihe der Fa.

ALTERA zu betrachten – siehe Bild 5.17. Die Konfiguration erfolgt ebenso wie beim GAL 16V8 mittels EEPROM-Zellen, allerdings mit dem wesentlichen Unterschied, daß kein Programmiergerät benötigt wird. Ähnlich wie in den Bildern 7.35 und 7.36 für die Mikrocontrollerprogrammierung wird nur ein einfaches Interface zum Standard-Druckerport eines PC benötigt, das im übrigen auch zur FPGA-Programmierung dienen kann. Schaltung und Aufbau werden weiter unten vorgestellt.

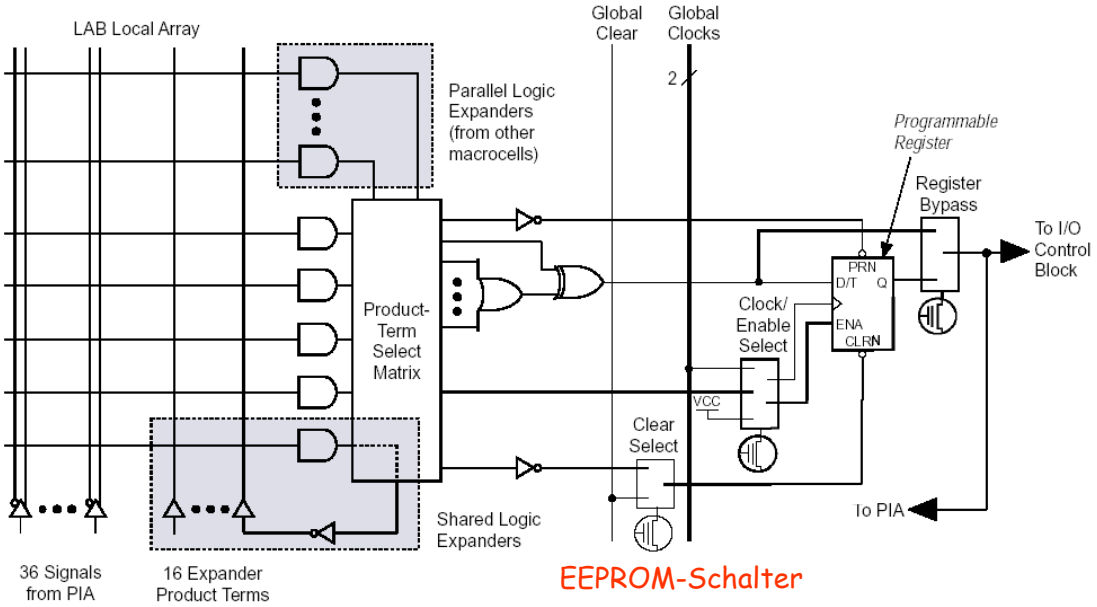



Bild 5.18: Typische Zellstruktur eines MAX 3000 CPLD-Bausteins

Eingangsseitig erkennt man die bewährte UND-Matrix, aus der mit dem Block „Produkt Term Select Matrix“ die gewünschte boolesche Funktion wiederum in disjunktiver Normalform erstellt werden kann. Ausgangsseitig hat man wieder ein Speicherelement, jedoch mit mehr Funktionen als beim einfachen PLD, die mittels mehrerer Multiplexer eingestellt werden können.

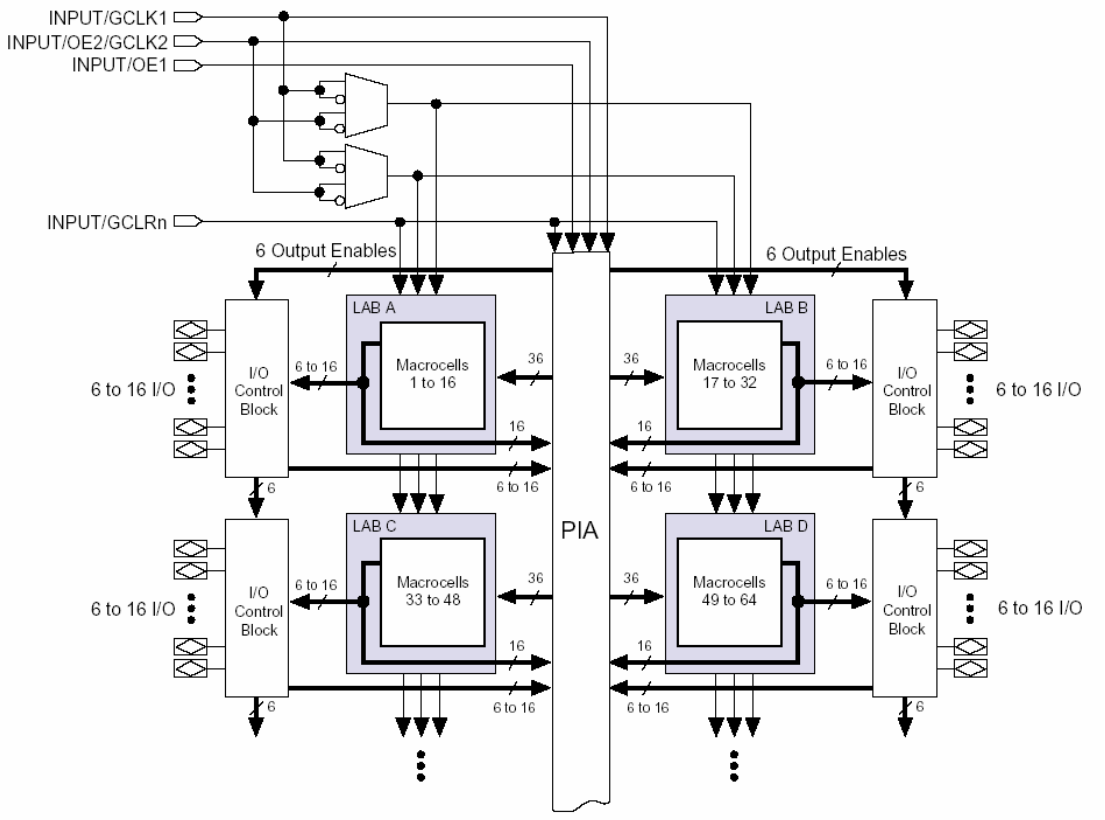



Bild 5.19: Gesamtaufbau eine MAX 3000 CPLDs

Der wesentliche Unterschied zum einfachen PLD wird in Bild 5.19 deutlich, wo der Gesamtaufbau des CPLDs zu sehen ist. Man sieht, daß hier jeweils 16 Zellen nach Bild 5.18 zu einem „**L**ogic **A**rray **B**lock“ (**LAB**) zusammengefaßt sind, von denen der hier dargestellte Baustein 4 Stück enthält. Man hat somit 64 Flip-Flops zur Verfügung, so daß sich schon sehr komplexe Automaten mit vielen Zuständen implementieren lassen. Neben den „I/O Control Blocks“, die jedem LAB zugeordnet sind, und in denen im wesentlichen festgelegt wird, ob ein Ausgang, ein Eingang oder eine Tristate-Funktion gewünscht ist, befindet sich im Zentrum eine komplexe Verdrahtungsmatrix, von deren Eigenschaften die erzielbare Geschwindigkeit der fertigen Schaltung wesentlich bestimmt wird. Hier kommt es darauf an, umständliches Routen zu vermeiden, d.h. die zahlreichen Signale, die hier zusammenkommen sind auf möglichst kurzen Wegen zu verschalten. In der Regel hängt die Leistungsfähigkeit einer implementierten Schaltung weit mehr von dieser Verdrahtungseffizienz ab als von Feinheiten des inneren Aufbaus der LABs oder deren Anzahl.

Der Schritt vom CPLD zum FPGA ist nun nicht mehr sehr groß, wie Bild 5.20 verdeutlicht.

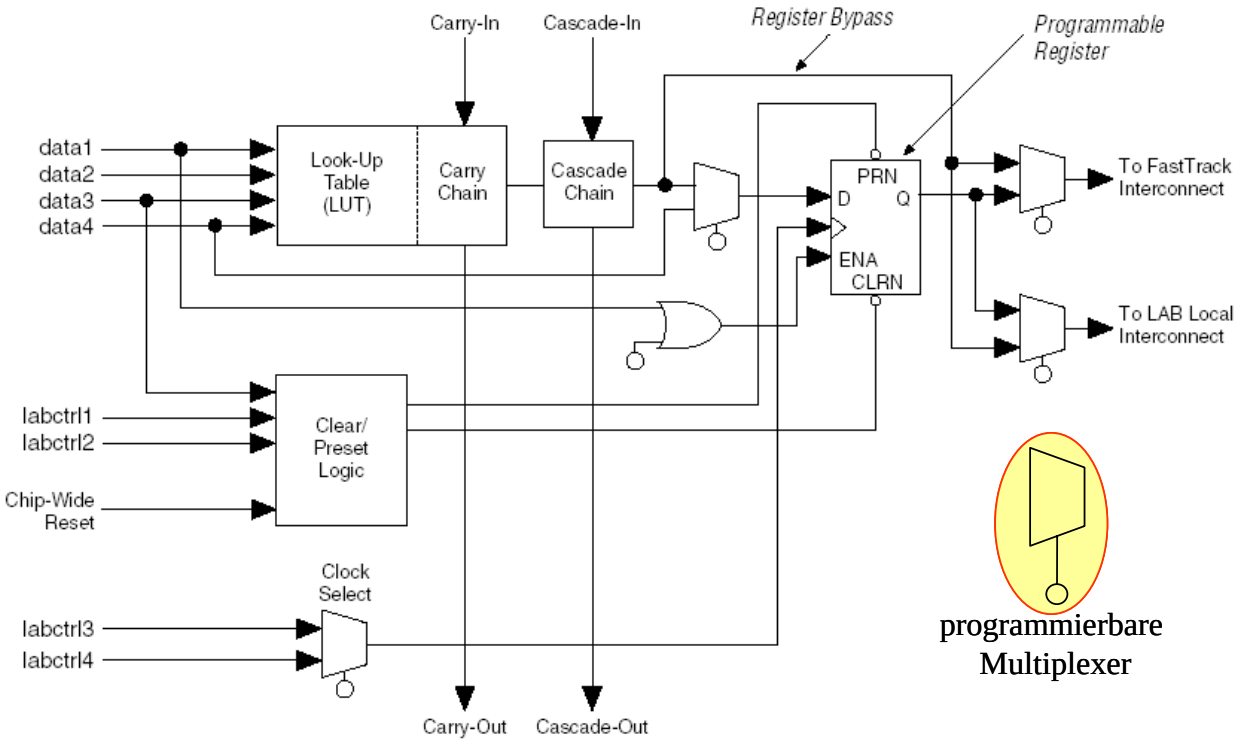



Bild 5.20: Logikelement eins FPGA aus der ACEX-Familie (ALTERA)

Auffällig ist allerdings, daß die breitgefächerte UND-Funktion am Eingang nicht mehr vorhanden ist. An ihre Stelle ist eine höchst flexible und viel allgemeiner einsetzbare Wertetabelle getreten. Der LUT–Block ermöglicht die Implementierung einer beliebigen booleschen Verknüpfung von 4 Eingangsvariabeln. Am Ausgang eines Logikelements findet man wieder das vorderflankengetriggerte D-Flip-Flop mit erweiterten Eigenschaften (Enable ENA, Preset PRN und Clear CLRN). Mit Hilfe zahlreicher programmierbarer Multiplexer läßt sich das Element sehr flexibel konfigurieren. Neben übergeordneten Takt- und Resetsignalen lassen sich die Eigenschaften des LE mit „labctrl“-Signalen, die aus anderen Blöcken stammen, beeinflussen. Die Verkettung von LEs zu komplexen Funktionen wird durch sogenannte Carry- und Cascade-Chains unterstützt.

In einer übergeordneten Hierarchiestufe werden aus den Logikelementen **L**ogic **A**rray **B**locks (**LAB**) gebildet – siehe Bild 5.21. Man sieht, daß hier 8 LE zusammengefaßt sind. Des weiteren sind in Bild 5.21 die Verdrahtungsmöglichkeiten angedeutet. Dabei werden zwei Hierarchieebenen unterschieden: LAB–Interconnect, sozusagen für den Nahbereich, und Row- bzw. Column-Interconnect für „Fernverbindungen“.

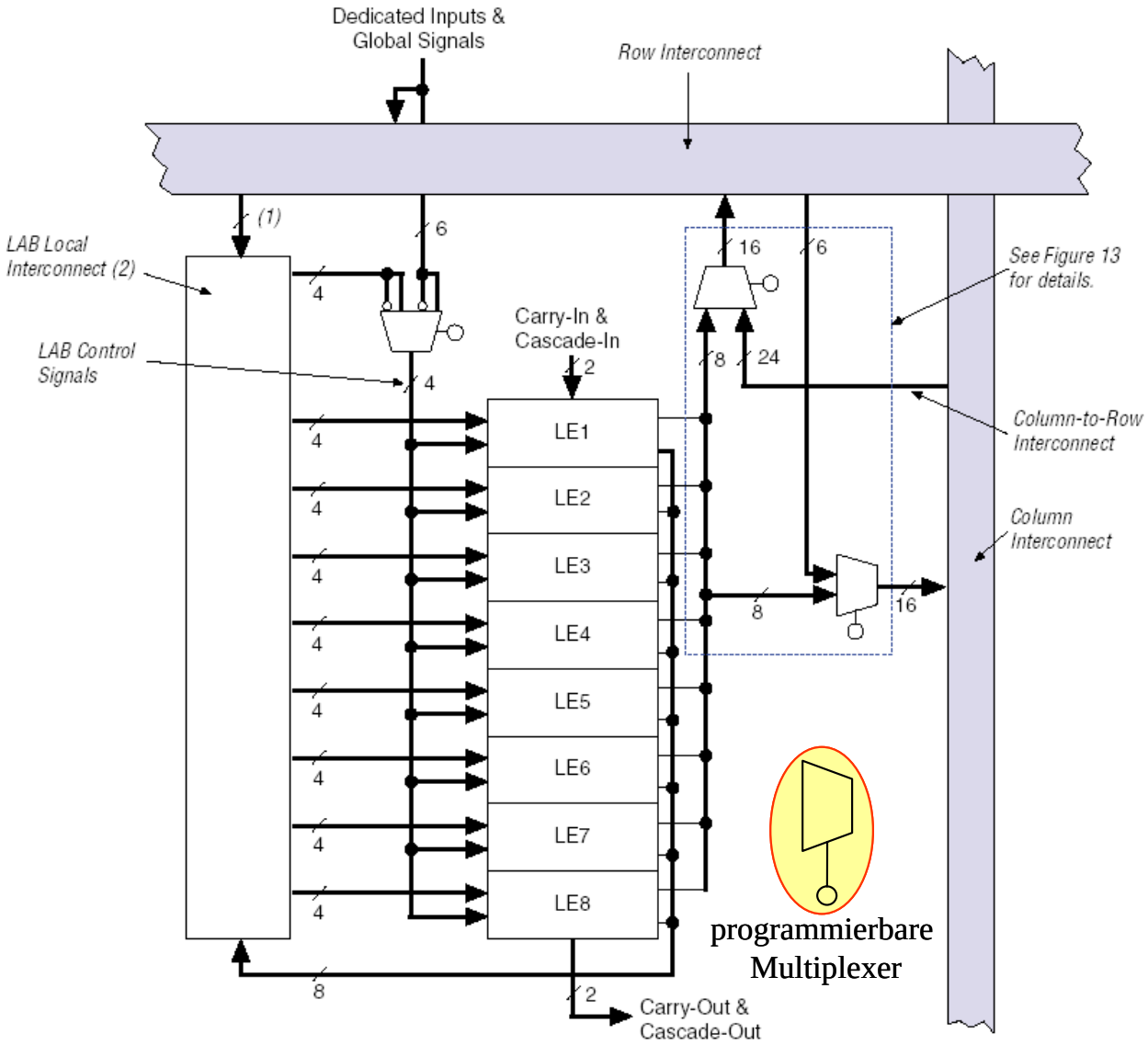



Bild 5.21: Aufbau eines LAB aus 8 Logikelementen bei ACEX FPGAs

Diesen Fernverbindungen kommt – wie schon beim CPLD angedeutet - eine hohe Bedeutung für die Leistungsfähigkeit und erzielbare Taktfrequenz der gesamten implementierten Schaltung zu. Das wird noch klarer beim Betrachten von Bild 5.23, wo der Gesamtaufbau eines ACEX-FPGAs dargestellt ist. Die matrixartige Verdrahtungsstruktur, die die Funktionseinheiten einrahmt, ist ganz augenfällig.

Rechts und links im Bild ist die nächsthöhere Hierarchiestufe erkennbar, in der jeweils 8 LABs nach Bild 5.21 zu einem **L**ogic **A**rray (**LA**) kombiniert sind. Im gezeigten Beispiel stehen 8 LAs mit insgesamt 64 LABs, d.h. mit 256 Logikelementen (LE) zur Verfügung.

Im Zentrum finden sich **E**mbedded **A**rray **B**locks (**EAB**), deren innere Struktur einen wesentlichen Beitrag dazu leistet, daß man komplette Mikrorechner - einschließlich DSPs – auf dem FPGA unterbringen kann. Hauptbestandteil eines EAB sind – wie Bild 5.22 zeigt – Speicherelemente in Form von RAM-Zellen, die sehr flexibel – besonders was die Wortbreite angeht – konfiguriert werden können. Diese Speicher wirken sich enorm ressourcensparend aus, das z.B. die integrierten Datenspeicher und Arbeitsregister eines Mikrorechners sehr schnell die relativ wenigen Flip-Flops in den Logikelementen aufbrauchen würden. In den EABs kann man darüber hinaus mit „verteilter Arithmetik“ auch schnelle Multiplizierer unterbringen.

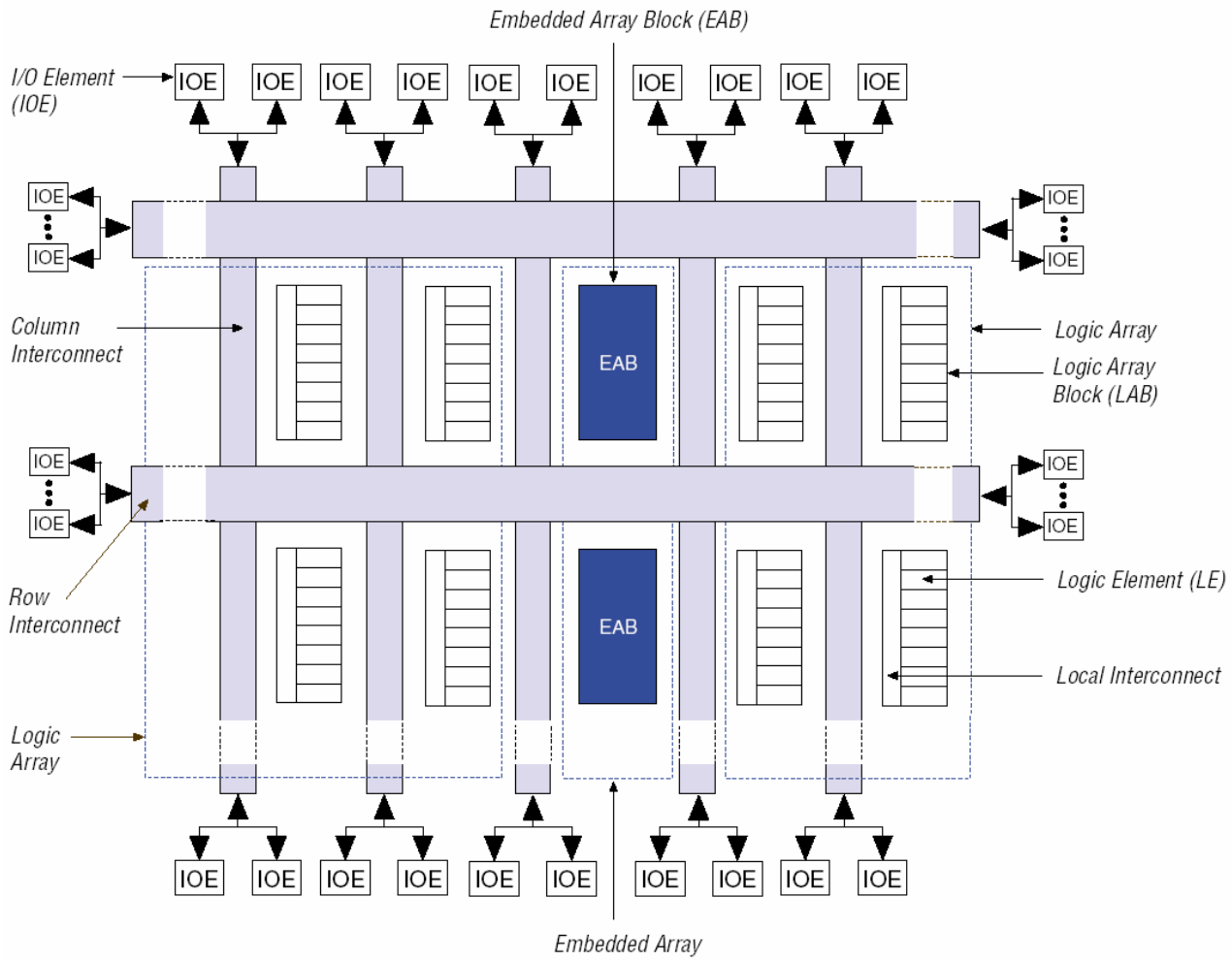



Bild 5.23: Gesamtarchitektur eines ACEX-FPGAs

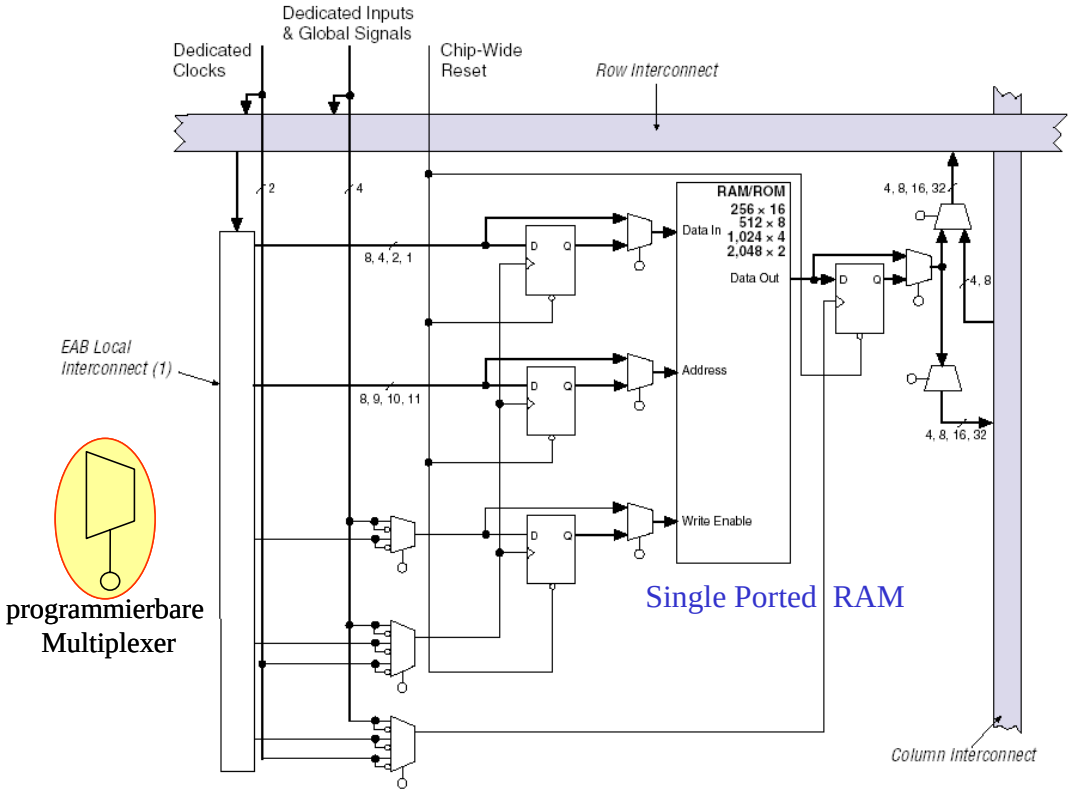



Bild 5.22: Embedded Array Block als „single-ported RAM" konfiguriert

Dazu gilt es, Algorithmen zu entwickeln, die das schon anhand von Bild 4.113 und Bild 4.114 vorgestellte speicherbasierte Multiplizierprinzip nutzen, wobei für lange Zahlen viele „Untereinheiten“ kaskadiert werden müssen. Aufgrund der hohen Zugriffsgeschwindigkeit der RAM-Zellen ist ein verteilt implementierter Multiplizierer trotz Kaskadierung immer noch sehr schnell. Eine detailliertere Beschreibung der Möglichkeiten, verteilte Arithmetik zu realisieren und weitere Eigenschaften der FPGA-Architekturen vorteilhaft zu nutzen, ist in Dokumenten, die über die Webseite http://www.altera.com erreichbar sind, zu finden.

Den Schluß dieses Abschnittes bildet ein Überblick über typische Eigenschaften einiger FPGAs.

Typ: Cyclone I

| Feature | EP1C3 | EP1C4 | EP1C6 | EP1C12 | EP1C20 |
|---|---|---|---|---|---|
| LEs | 2,910 | 4,000 | 5,980 | 12,060 | 20,060 |
| M4K RAM blocks (128 × 36 bits) | 13 | 17 | 20 | 52 | 64 |
| Total RAM bits | 59,904 | 78,336 | 92,160 | 239,616 | 294,912 |
| PLLs | 1 | 2 | 2 | 2 | 2 |
| Maximum user I/O pins *(1)* | 104 | 301 | 185 | 249 | 301 |

bislang

Einsatz in Forschungsprojekten des IIIT

ab jetzt

Typ: Cyclone II

| Feature | EP2C5 | EP2C8 | EP2C20 | EP2C35 | EP2C50 | EP2C70 |
|---|---|---|---|---|---|---|
| LEs | 4,608 | 8,256 | 18,752 | 33,216 | 50,528 | 68,416 |
| M4K RAM blocks (4 Kbits plus 512 parity bits | 26 | 36 | 52 | 105 | 129 | 250 |
| Total RAM bits | 119,808 | 165,888 | 239,616 | 483,840 | 594,432 | 1,152,000 |
| Embedded multipliers *(1)* | 13 | 18 | 26 | 35 | 86 | 150 |
| PLLs | 2 | 2 | 4 | 4 | 4 | 4 |
| Maximum user I/O pins | 158 | 182 | 315 | 475 | 450 | 622 |

Tabelle 5.2: Übersicht über die Eigenschaften von „low-cost“ und „low-power“ FPGAs

Man erkennt in Tabelle 5.2 die abgestufte Komplexität, die sich z.B. in der Zahl der „Typical Gates“ widerspiegelt. Die Diskrepanz zur Zahl der „Maximum System Gates“ soll an dieser Stelle nicht detailliert erklärt werden. Es sei nur darauf hingewiesen, daß bei den „Maximum System Gates“ alle Schaltungsteile, inklusive der umfangreichen Kette der „Boundary-Scan“-Flip-Flops, die für Test und Programmierung benötigt werden, mitgezählt sind. Des weiteren die gesamte Programmierlogik sowie die für das Konfigurieren benötigten Schalter und Speicher – siehe auch Bild 5.24. Somit sind hier erhebliche Schaltungsteile erfaßt, die zur Chip-Infrastruktur gehören, die aber dem Anwender nicht zur Implementierung seiner Funktionen zur Verfügung stehen.

Die Zahl der Logikelemente (LE), der EABs und der RAM-Bits gibt hingegen klare Anhaltspunkte für den Anwender. In den Forschungsarbeiten des IIIT wurde in den letzen Jahren der Typ EP1C12 eingesetzt. Seine Komplexität reichte aus, um z.B. schnelle digitale Filter zu realisieren, d.h. die Rechenleistung von mehreren Standard-DSPs zu implementieren. Derzeit werden größere und schnellere FPGAs vom Typ Cyclone II verwendet, um unter anderem schnelle digitale Kommunikationssysteme für Datenraten >>10 Mbit/s zu realisieren. Sie verfügen über „vorgefertigte“ (embedded) Multiplizierer. Einige aktuelle Forschungsprojekte, in denen FPGAs eingesetzt werden, werden in der Vorlesung vorgestellt.

### 5.1.5.5 FPGA und CPLD-Programmierung

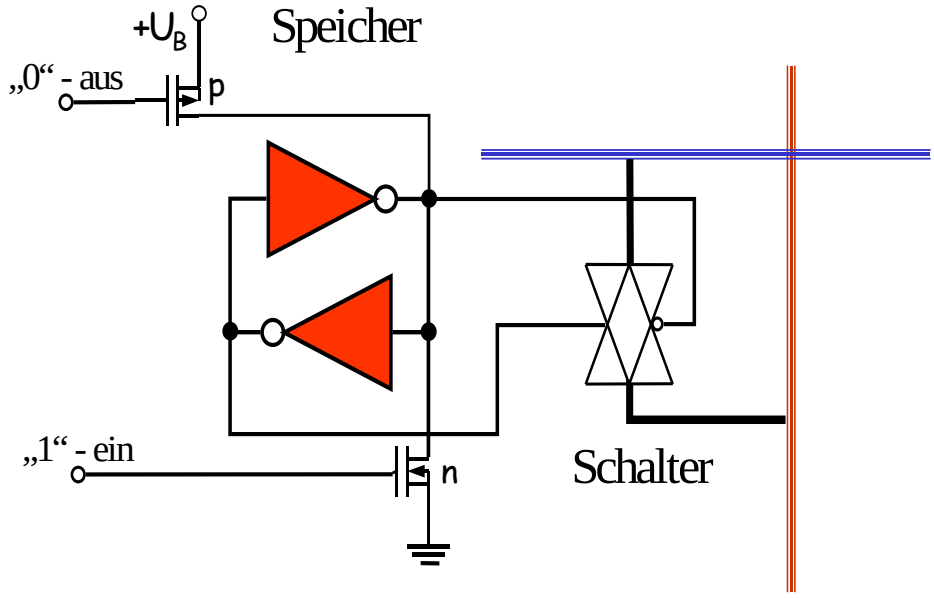



Bild 5.24: Prinzip der FPGA-Konfiguration mittels statischer RAM-Zelle als Speicher und Transfergate als bidirektionalem Schalter.

Von den vielfältigen Möglichkeiten PLDs, CPLDs und FPGAs zu konfigurieren wird hier exemplarisch die mit einem bidirektionalen Schalter gekoppelte statische RAM-Zelle betrachtet.

In der Entwicklungsphase eines neuen Produkts hat diese Variante den Vorteil, daß die Zahl der Programmierzyklen unbegrenzt ist, und daß der Programmiervorgang auch bei komplexen Bausteinen sehr schnell abläuft. Beim fertigen Produkt muß durch einen Festwertspeicher in Form eines Konfigurations-EEPROMs (mit 1bit breitem Ausgang) dafür gesorgt werden, daß die Funktion beim Einschalten der Stromversorgung geladen wird. Das geht sogar schneller als über den PC-Druckerport, d.h. in Bruchteilen einer Sekunde ist das System betriebsbereit.

Die Bestandteile der Schaltung in Bild 5.24 sind aus Kapitel 4, Bild 4.12 (Transfergate) und Kapitel 5, Bild 5.31 (statische RAM-Zelle) bekannt. Ein kurzes Anlegen eines „1“-Pegels an den n-Kanal-MOS-FET führt zum Einschalten des Schalters und dieser Zustand bleibt durch die Doppelinverterstruktur gespeichert, solange, bis ein „0“-Pegel an den p-MOS-FET angelegt wird, der den Speicher zum „Kippen“ in den anderen Zustand bringt. Im Ruhezustand sind beide Transistoren (n- und p-Kanal) ausgeschaltet, so daß das Speicherelement aus den beiden Invertern seine Information behält bis eine Stromabschaltung erfolgt. Da es in der Praxis wichtig ist, daß alle Schalter kurz nach Anlegen der Betriebsspannung den gleichen Zustand – nämlich AUS – einnehmen, ist ein globaler Reset-Impuls an alle p-Kanal-Transistoren anzulegen, bevor die Konfiguration erfolgt. Dazu muß dann für jeden Schalter, der eingeschaltet sein muß, ein „1“-Impuls an den zugehörigen n-Kanal-Transistor gelegt werden. Die Übertragung dieser Information, die während des Placement- und Routing-Prozesses im PC erzeugt wurde, erfolgt über den Druckerport mit Hilfe einer einfachen, in Bild 5.25 dargestellten Interfaceschaltung, die im wesentlichen aus einem CMOS-Treiberbaustein (74HC244) besteht. Während die 100Ω-Widerstände für ein rasches Abklingen von leitungsbedingten Überschwingern an den Signalflanken sorgen, dienen die 2,2kΩ-Widerstände zur Anpassung der verschiednen Betriebsspannungen (5V am PC und 3,3 V am FPGA). Bei dem ISP-Stecker handelt es sich um einen einfachen Flachkabelverbinder, der auf ein 10-adriges Flachkabel aufgepreßt wird. Geometrie und Pin-Numerierung sind gemäß Bild 4.90 genormt, während in Bild 5.27 die Pin-Funktionen kurz beschrieben sind. Hier wird die Verknüpfung mit der in Abschnitt 8.1.4 kurz angesprochenen Scan-Path oder Boundary-Scan Testmethodik klar – dort wurde auch bereits die Abkürzung JTAG genannt. Wie aus Bild 5.27 ersichtlich ist können die einzelnen Pins je nach Anwendung (Programmierung oder Schaltungstest in der Halbleiterherstellung) verschiedene Aufgaben haben. Die Norm gibt lediglich einen Rahmen vor, der neben der Steckergeometrie auch die Stromversorgungspins (GND: Pins 2 und 10, VCC: Pin 4), die Taktversorgung (Pin 1), sowie Eingangpin (9) und Ausgangspins (3) allgemein definiert.

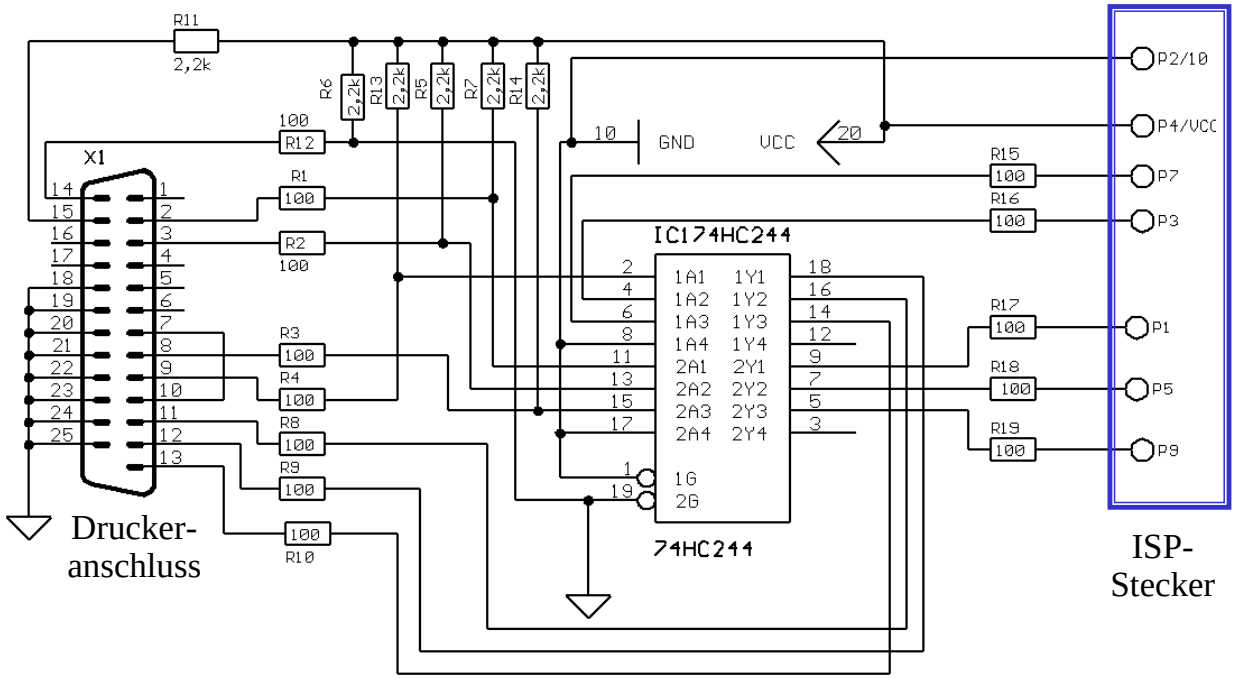



Bild 5.25: Interface zwischen PC-Druckerport und „In-System-Programmierstecker“

Bild 5.27 stellt die Programmieranwendung (PS-Mode), die uns hier interessiert der Testanwendung (JTAG-Mode) gegenüber. Man sieht, daß zum Programmieren neben dem Takt und dem einzuschreibenden Datenstrom (DATA0) noch ein Steuerbit (nCONFIG) und zwei Überwachungsbits (nSTATUS und CONF_DONE) vorgesehen sind.

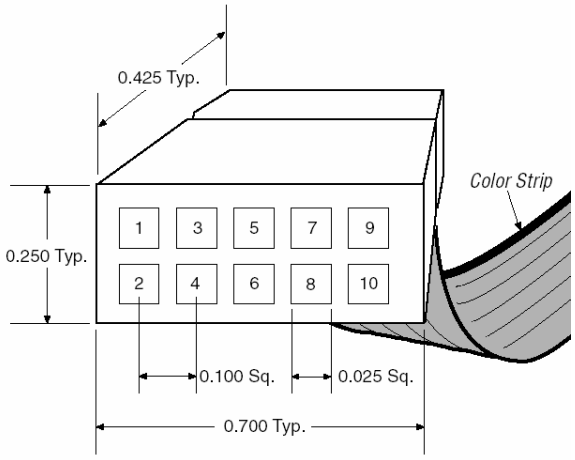



Bild 5.26: Aufbau und Belegung des 10-poligen ISP-Steckers

| Pin | PS Mode | | JTAG Mode | |
|---|---|---|---|---|
| | Signal Name | Description | Signal Name | Description |
| 1 | DCLK | Clock signal | TCK | Clock signal |
| 2 | GND | Signal ground | GND | Signal ground |
| 3 | CONF_DONE | Configuration control | TDO | Data from device |
| 4 | VCC | Power supply | VCC | Power supply |
| 5 | nCONFIG | Configuration control | TMS | JTAG state machine control |
| 6 | – | No connect | – | No connect |
| 7 | nSTATUS | Configuration status | – | No connect |
| 8 | – | No connect | – | No connect |
| 9 | DATA0 | Data to device | TDI | Data to device |
| 10 | GND | Signal ground | GND | Signal ground |

Bild 5.27: Pin-Funktionen am ISP-Stecker

Im JTAG–Testbetrieb müssen – wie in Abschnitt 8.1.4 kurz umrissen wurde – Testvektoren über Pin 9 (TDI) in die interne Flip-Flop-Kette eingeschrieben werden und die Testergebnisse über Pin 3 (TDO) ausgelesen werden.

Beim Arbeiten mit FPGAs muß man sich nicht um Details der Signalübertragung für die Programmierung kümmern. In der vollständigen PC-basierten FPGA–Entwicklungsumgebung **QUARTUS II** der Fa. Altera, die hier kurz exemplarisch betrachtet wird, steht die Menüfunktion „Programmieren“ zur Verfügung, die für die Übertragung der Konfigurationsinformation über Druckerport und ISP–Stecker ins Ziel–FPGA sorgt.

Der Vorgang ist in Bild 5.28 anschaulich zusammengefaßt.

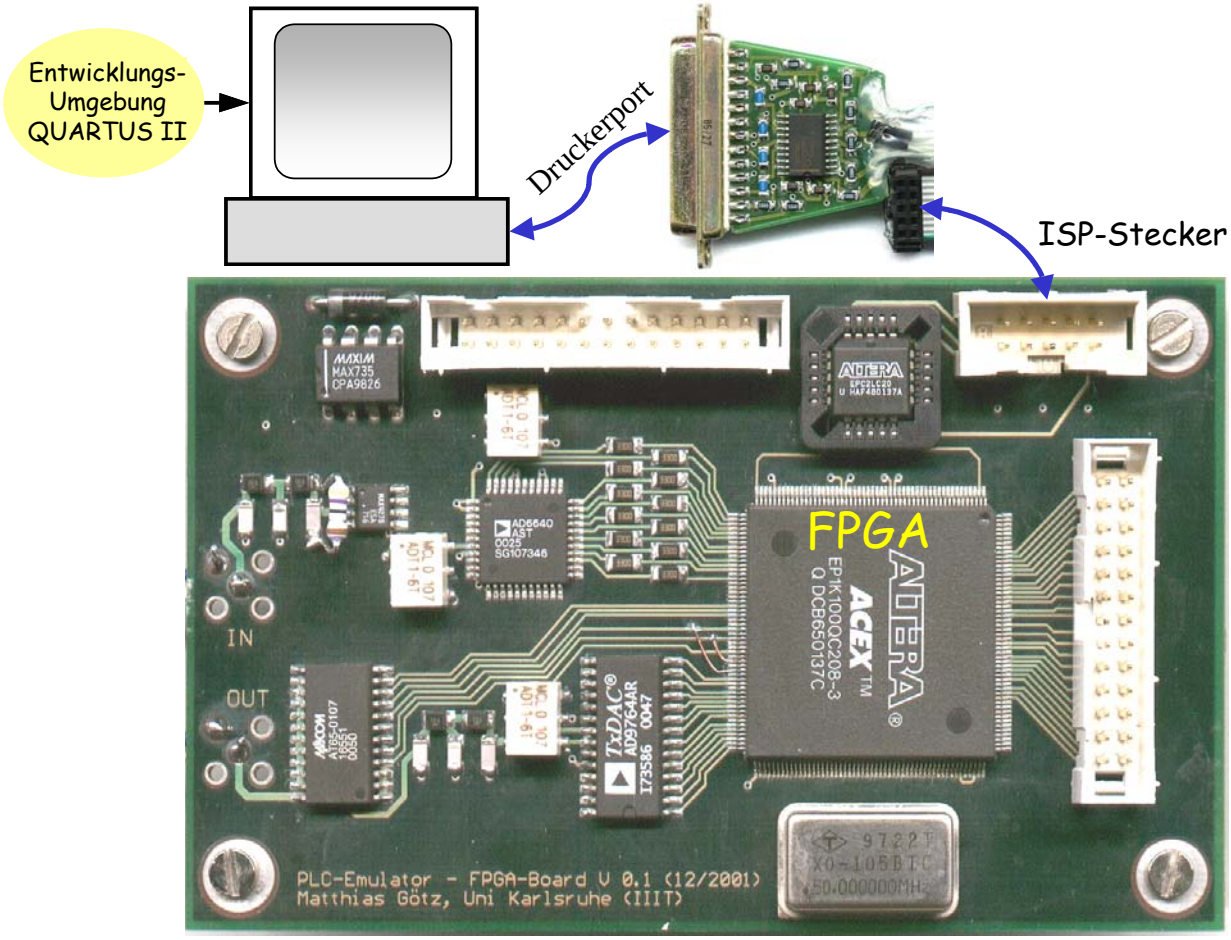



Bild 5.28: Illustration des Programmiervorganges bei FPGAs

Im oberen Bildteil ist neben dem PC die Platine der Schaltung nach Bild 5.25 zu sehen, die das Bindeglied zwischen PC und Anwenderhardware bildet. Mit dem FPGA wird hier ein komplexer Powerline–Kanalemulator realisiert, zu dem in Anhang einige weitergehende Informationen gegeben werden. Links neben dem ISP–Stecker erkennt man einen kleinen quadratischen Baustein in einem Sockel. Hierbei handelt es sich um das Konfigurations–EEPROM, aus dem das FPGA automatisch beim Anlegen der Stromversorgung programmiert wird, wenn die Entwicklung abgeschlossen und die PC-Verbindung entfernt ist.

Bild 5.29 gibt einen Einblick in die schon mehrfach erwähnte vollständige FPGA–Entwurfsumgebung QUARTUS II der Fa. Altera. Die Software belegt ca. 2GB Festplattenplatz und läuft auf PCs unter Windows oder Linux, kann aber auch unter UNIX auf Workstations eingesetzt werden. Um auch sehr große FPGAs mit z.B. 4.000.000 „Typical Gates“ bearbeiten zu können, sollte ein PC über mindestens 2GB Arbeitsspeicher verfügen.

In QUARTUS II steht ein komfortabler VHDL–Editor zur Verfügung. Des weiteren gibt es umfangreiche Bibliotheken mit logischen Primitiven und auch Makros – von denen viele allerdings extra gekauft, bzw. lizenziert werden müssen. Hiermit kann sowohl in VHDL als auch mit dem Grafikeditor – vgl. Bild 5.29 gearbeitet werden. Hinter jedem der Blöcke steht ein VHDL–Modell, das an die zu lösende Aufgabe angepaßt werden kann. Wenn die Design-Eingabe komplett ist kann die Synthese auf die Zielhardware hin erfolgen. Es entsteht eine Netzliste, auf deren Basis die Schaltung mit den Bauteileigenschaften simuliert werden kann. Nachdem Plazierung und Verdrahtung erfolgt sind, ist eine erneute Simulation, die den Einfluß der Verdrahtung berücksichtigt, erforderlich. Bestätigt diese Simulation die gewünschte Funktion der Schaltung, kann der Program-

mierer aufgerufen werden, der die Konfigurationsdaten - wie oben beschrieben - ins Ziel–FPGA bringt.

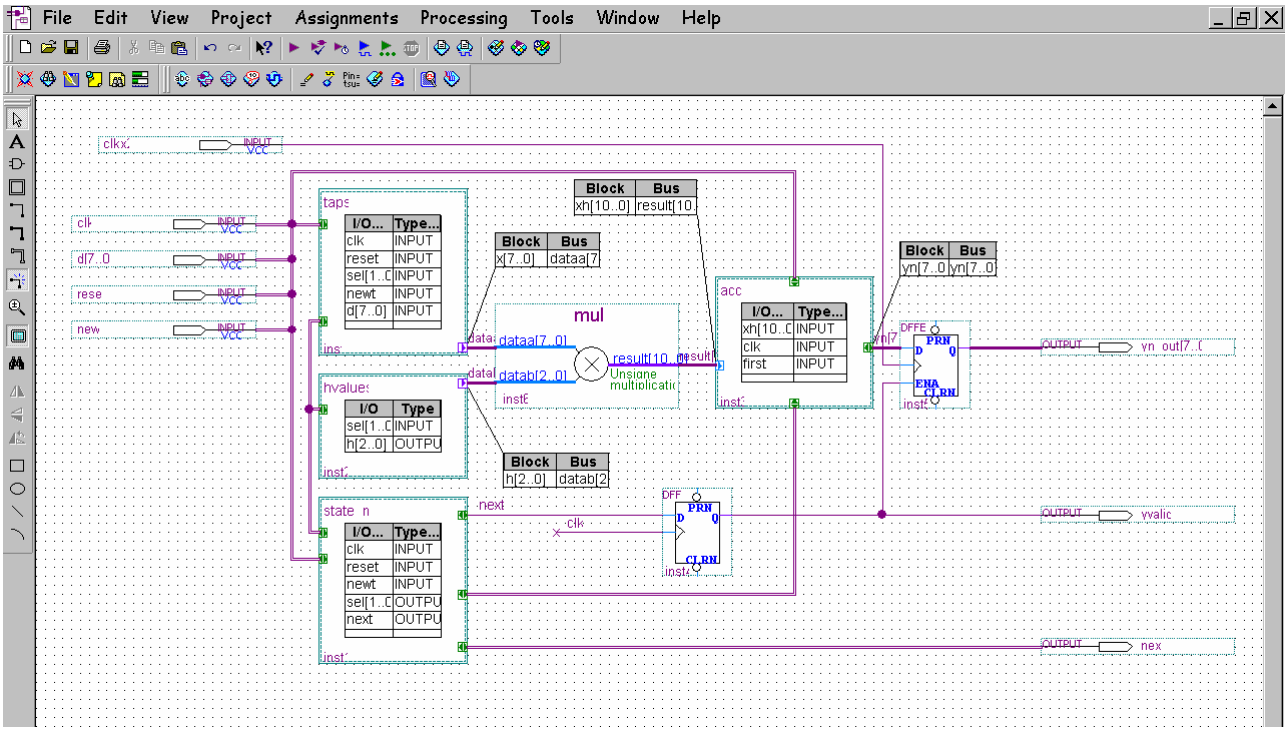



Bild 5.29: Einblick in die FPGA–Entwurfsumgebung QUARTUS II (Grafik Editor)

Die QUARTUS II Software wird Universitäten zu sehr günstigen Sonderkonditionen zur Verfügung gestellt. Am IIIT steht eine mehrfache „Floating License“ zur Verfügung, die jeder am Institutsnetz angeschlossene PC nutzen kann, d.h. auch die PCs in Diplomandenräumen, solange, bis die Zahl der Lizenzen ausgeschöpft ist. Falls sich hierbei künftig Engpässe ergeben sollten, kann zu geringen Kosten die Zahl der Lizenzen erhöht werden.

Neben der Vollversion bietet ALTERA auch eine kostenlose WEB-EDITION zur Entwicklung von CPLDs und FPGAs auf der WEB-SITE

https://www.altera.com/support/software/download/altera_design/quartus_we/dnl-quartus_we.jsp

an. Der Link führt zum „Download“–Bereich, von wo die

***Quartus II Web Edition Version 7.0***

im Umfang von ca. 500MB heruntergeladen werden kann. Zur Lizenzierung ist die Angabe der Netzwerkkartennummer des Zielrechners erforderlich. Ein File mit der Bezeichnung „license.dat“ wird per E-Mail zugesandt und ermöglicht nach Installation gemäß den Anweisungen auf obiger WEB-SITE die Nutzung der Software. Nach ca. 6 Monaten läuft die Lizenz ab und kann durch Anforderung eines neuen „license.dat“–Files wiederum für 6 Monate erneuert werden.

## 5.2 VHDL−Tutorial

### 5.2.1 Allgemeines über VHDL

Für die allgemeine Akzeptanz einer HDL ist ihre Effizienz im Rahmen einer Entwicklungsumgebung von entscheidender Bedeutung.

Die HDL muß dabei werkzeugunabhängig sein.

VHDL erfüllt diese Voraussetzungen und hat darüber hinaus den Status eines IEEE−Standards. Auf dieser soliden Grundlage hat VHDL einen hohen Grad der Durchdringung in Industrie und Forschung erreicht. Eine Vielzahl von Werkzeugen für Simulation und Synthese stehen heute auf der Basis von VHDL zur Verfügung. Im folgenden sind einige wichtige Aspekte, deren sich jeder Nutzer von VHDL bewußt sein sollte, zusammengestellt.

- ✓ VHDL ist sowohl von der Hardwareplattform als auch von der dort verwendeten Software (z.B. Betriebssystem) unabhängig. Es gibt mittlerweile zahlreiche Anbieter von Software, die für die meisten Entwurfsschritte digitaler integrierter Schaltungen VHDL−basierte Lösungen bereitstellen. Da VHDL das Einbinden systemabhängiger Aspekte in „Packages“ unterstützt, können VHDL−Modelle in der Regel problemlos portiert werden.
- ✓ VHDL ist technologieunabhängig. Die Entscheidung für eine bestimmte Technologie (FPGA, Gate Array, Standardzelle, Vollkundenschaltung) muß erst relativ spät getroffen werden. Ein eventuell nötiges Umschwenken auf eine andere Technologie erfordert nur Änderungen in den technologieorientierten Schritten, d.h. beginnend mit der Schaltungssynthese.
- ✓ VHDL hat das Entstehen und die starke Verbreitung leistungsfähiger Synthesewerkzeuge in Gang gesetzt. Auf diese Weise ist eine neue und produktivere Entwurfsmethodik für digitale integrierte Schaltungen heute praktisch jedem Ingenieur in an seinem Laborarbeitsplatz zugänglich.
- ✓ VHDL ist ohne Erläuterungen durch Kommentare les- und verstehbar. Die Syntax ist sehr ausführlich und hat selbsterklärende Befehle, so daß der Begriff Selbstdokumentation absolut berechtigt ist.
- ✓ Mittels VHDL läßt sich ein bedeutendes Merkmal anwendungsspezifischer Digitalschaltungen, nämlich das parallele Arbeiten von Funktionseinheiten, beschreiben, wohingegen nahezu alle heutigen Standard-Mikrorechner und die zugehörige Software immer noch sequentiell orientiert sind.
- ✓ Als Nachteil von VHDL ist die Komplexität zu nennen, die eine sehr lange Einarbeitungszeit erforderlich macht. Das Verhalten komplexer VHDL−Modelle in der Simulation ist für einen Neuling kaum nachvollziehbar.

### 5.2.2 VHDL-Sprachumfang

Die in Rahmen dieser Einführung beschriebenen Anweisungen sind eine Untermenge der Möglichkeiten, die VHDL bietet. So wird beispielsweise nicht auf Funktionen und Prozeduren eingegangen. Auch Befehle wie „**EXIT**“ oder „**ASSERT**“ sind für diese Einführung nicht von Bedeutung und wurden deshalb weggelassen.

#### 5.2.2.1 Port-Deklaration

In der Port-Deklaration werden jeweils die Ein- und Ausgänge eines Designs deklariert. Der Modus gibt dabei die Richtung des Signals an. IN bedeutet, daß es sich bei dem Signal um ein Eingangssignal handelt. Solche Signale sind daher nicht beschreibbar.

OUT dagegen zeigt an, daß das Signal ein Ausgangssignal ist. Solche Signale sind daher nur beschreibbar und nicht lesbar. Sie können auch nicht in logische Verknüpfungen einbezogen werden. Durch INOUT werden Busse deklariert.

„Typ“ gibt den Typ des Signals an. Dieser kann vordefiniert oder selbstdefiniert sein. Bei der Syntax ist zu beachten, daß die letzte Zeile nicht mit einem Semikolon abgeschlossen wird.

Syntax:

```
PORT (port_name: Modus Typ;
      port_name: Modus Typ
     );
```

Beispiel:

```
PORT (a: IN    BOOLEAN;
     b: IN    INTEGER RANGE 0 TO 31;
     c: INOUT BIT;
     e: OUT   STD_LOGIC;
     f: OUT   STD_LOGIC_VECTOR (0 DOWNTO 3)
     );
```

### 5.2.3 Konstantendeklaration

Eine Konstante weist einem Namen einen bestimmten Wert zu. Dieser Wert kann im gesamten Programm nur gelesen und nicht neu beschrieben werden.

Es ist vorteilhaft, ein Programm so weit wie möglich in Abhängigkeit von Konstanten zu entwikkeln (z.B. Länge eines Schieberegisters, Breite eines Addierers). Dies erleichtert spätere Änderungen erheblich.

Syntax:

```
CONSTANT Konstantenname: Typ := Wert;
```

Beispiel:

```
CONSTANT a: BOOLEAN   := TRUE;
CONSTANT b: INTEGER   := 31;
CONSTANT c: BIT_VECTOR (3 DOWNTO 0) := "0000";
CONSTANT d: STD_LOGIC := 'Z';
CONSTANT e: STD_LOGIC_VECTOR (3 DOWNTO 0) := "0-0-";
```

### 5.2.4 Variablendeklaration

Variablen haben im Gegensatz zu Signalen keine globale Gültigkeit. Sie können daher nur in Prozessen deklariert und verwendet werden. Es gibt zwar die Möglichkeit, globale Variablen in Form von „**SHARED VARIABLES**“ zu definieren. Davon sollte jedoch möglichst kein Gebrauch gemacht werden, da so Konstrukte entstehen können, deren Verhalten nicht vorhersagbar ist.

Auch hier können Defaultwerte zugewiesen werden. Ansonsten werden wie bei den Signalen die niedrigsten Werte des erlaubten Wertebereichs angenommen.

Syntax:

```
VARIABLE var_name: Typ;
```

Beispiel:

```
VARIABLE a: BOOLEAN;
VARIABLE b: INTEGER RANGE 0 TO 31;
VARIABLE c: BIT;
VARIABLE d: BIT_VECTOR (3 DOWNTO 0);
VARIABLE e: STD_LOGIC;
VARIABLE f: STD_LOGIC_VECTOR (0 TO 3);
```

### 5.2.5 Signaldeklaration

Neben Variablen und Konstanten, die auch von prozeduralen Programmiersprachen her bekannt sind, verfügt VHDL zusätzlich über Objekte der Klasse „Signal“. Diese Klasse wurde eingeführt, um spezielle Eigenschaften elektronischer Schaltungen modellieren zu können. Unter anderem können Änderungen von Signalwerten zeitlich verzögert zugewiesen werden. Damit lassen sich die Laufzeiten von Hardwarekomponenten nachbilden. Signale dienen im wesentlichen dazu, Daten zwischen parallel arbeitenden Modulen auszutauschen. Verschiedene Module können dabei mitunter auf ein und dasselbe Signal schreiben. Diese Eigenschaft gestattet die Modellierung von Bussen in VHDL.

Wird bei der Deklaration kein Defaultwert mit ‘**:=**‘ explizit zugewiesen, so wird der bei Typ am weitesten links stehende Wert als Defaultwert angenommen.

Syntax:

```
SIGNAL sig_name: Typ;
```

Beispiel:

```
SIGNAL a: BOOLEAN := true;          --Defaultwert: true
SIGNAL b: BOOLEAN;                  --Defaultwert: false
SIGNAL c: INTEGER RANGE 0 TO 31;    --Defaultwert: 0
SIGNAL d: BIT;                      --Defaultwert: ‘0‘
SIGNAL e: BIT_VECTOR (3 DOWNTO 0);  --Defaultwert:‘0‘,‘0‘,‘0‘,‘0‘
SIGNAL f: STD_LOGIC;                --Defaultwert: ‘0‘
```

### 5.2.6 Zuweisungen

Signalzuweisung:
```
sig_name <= Wert;
```

Variablenzuweisung:
```
var_name := Wert;
```

Bemerkung: Durch die Symbole := oder =: bzw. => oder <= werden Wertzuweisungen an Variablen und an Signale unterschieden.

### 5.2.7 Komponenten

Komponenten dienen der Realisierung strukturaler Modelle. Durch Einbinden getrennter Programmstücke als sogenannte „BLACK BOXES“ wird das Gesamtprogramm sehr übersichtlich. Man muß nur spezifizieren, wie Ein- und Ausgangssignale verdrahtet werden, ohne den Inhalt der entsprechenden Komponente anzuschauen. Ein Ändern von Funktionen ist dadurch selbst in späten Entwicklungsstadien einfach möglich.

#### 5.2.7.1 Komponentendeklaration

Eine Deklaration macht eine Komponente mit ihren Ein- und Ausgangsports bekannt. Dies geschieht im Deklarationsteil einer **ARCHITECTURE**, eines **BLOCKS** oder einer **PACKAGE**.

Syntax:
```
COMPONENT Komponentenname
  Portdeklarationen;
END COMPONENT;
```

#### 5.2.7.2 Komponenteninstantiierung

Bei der Instantiierung werden die bereits bekannt gemachten Ports der Komponente verdrahtet. Dies geschieht durch Zuweisen von Signalnamen an die Ports.

Syntax:
```
Name: Komponentenname
   PORT MAP (
    port_name => sig_name,
    port_name => sig_name
           );
```

### 5.2.8 Typdefinition

Bevor in einem VHDL–Modell mit einem Objekt gearbeitet werden kann, muß festgelegt werden, welche Werte das Objekt annehmen darf. Zu diesem Zweck werden Datentypen deklariert, mit deren Hilfe die Wertebereiche definiert werden. VHDL verfügt nur über sehr wenige, vordefinierte Datentypen, wie z.B. **real** oder **integer**. Es gibt jedoch umfangreiche Möglichkeiten, benutzerdefinierte Datentypen zu erzeugen. Man kann von vordeklarierten Typen sogenannte Untertypen ableiten. Untertypen sind im einfachsten Fall im Wertebereich eingeschränkte Grundtypen. Die weitere Ableitung von Untertypen aus einem Untertyp ist nicht möglich.

Beispiel:
```
TYPE Aufzaehlungstyp IS (a, b, c, d, e);
SUBTYPE S_Aufzaehlungstyp IS Aufzaehlungstyp RANGE b TO d;
```

### 5.2.9 Process

Während nebenläufige Anweisungen direkt im Programmteil der **Architecture** stehen, enthalten die ebenfalls dort angesiedelten Prozesse Vorgänge, die in Abhängigkeit von Signalereignissen ablaufen (z.B. Taktvorderflanke).

Hier ist zu beachten, daß ein Prozeß nur dann aufgerufen wird, wenn sich eines der in seiner **„Sensitivity–Liste“** aufgeführten Signale ändert.

Prozesse sind generell nebenläufig.

Während Komponenten in Prozessen nicht erlaubt sind, dürfen aber lokale Variablen dort deklariert und unmittelbar verwendet werden.

Syntax:

```
Name:   -- optional
  PROCESS (Sensitivity-List)
    Deklarationen
  BEGIN
    Anweisungen
  END PROCESS;
```

Beispiel:

```
PROCESS (clock)
BEGIN
  IF (clock'event AND clock='1') THEN -- Taktvorderflanke
    ...
  END IF;
END PROCESS;
```

### 5.2.10 If

Die **IF–**Anweisung ist ebenfalls aus höheren Programmiersprachen bekannt. Sie arbeitet Anweisungen in Abhängigkeit von einer Bedingung ab, sobald diese als Ergebnis den booleschen Wert „**true**“ liefert.

Syntax:

```
IF Bedingung THEN Anweisung;
   ELSIF Bedingung THEN Anweisung
   ELSE Anweisung
END IF;
```

### 5.2.11 Case

Die **CASE–**Anweisung ist eine Möglichkeit zur Abkürzung von **IF**–Befehlsketten bei bedingter Ausführung mehrerer Befehle, die alle abhängig von einem Ausdruck sind. „**WHEN OTHERS**” ist ein Synonym für alle nicht aufgeführten Bedingungen.

Syntax:

```
CASE Ausdruck IS
  WHEN wert   => Anweisung;
  WHEN wert   => Anweisung;
```

```
  WHEN OTHERS => Anweisung;
END CASE;
```

Beispiel:

```
CASE sel IS
  WHEN 0 | 1 | 2 => z <= b;
  WHEN 3 to 10   => z <= c;
  WHEN OTHERS    => z <= d;
END CASE;
```

### 5.2.12 Bedingte Signalzuweisung

Während die Case–Anweisung in Abhängigkeit von einem Ausdruck entscheidet, können mit der **bedingten Signalzuweisung** einem Signal in übersichtlicher Weise mehrere Werte in Abhängigkeit von mehreren Bedingungen zugewiesen werden.

**Fall 1:**

Syntax:

```
Name:    -- optional
  sig_name <= Wert WHEN Bedingung ELSE
              Wert WHEN Bedingung ELSE
              Wert;
```

Beispiel:

```
z <= a WHEN (x>10) ELSE
     b WHEN (x>5)  ELSE
     c;
```

**Fall 2:**

Syntax:

```
Name:    -- optional
  WITH Ausdruck SELECT
    sig_name <= Wert WHEN Bedingung,
                Wert WHEN Bedingung,
                 Wert WHEN OTHERS;
```

Beispiel:

```
WITH sel SELECT
  z <= a WHEN 0 | 1 | 2,
       b WHEN 3 TO 10,
       c WHEN OTHERS;
```

### 5.2.13 For–Schleife

Die For–Schleife wird [Obergrenze – Untergrenze + 1]-mal sequentiell abgearbeitet, d.h. der Ablauf entspricht dem äquivalenten Konstrukt einer höheren Programmiersprache. Die lokale Variable *i* muß dabei nicht vorher deklariert worden sein. Dies geschieht automatisch beim ersten Aufruf der Schleife.

Die Anzahl der Schleifendurchläufe hingegen muß für die Synthese bekannt sein.

Syntax:

```
Name:  -- optional
  FOR i IN Untergrenze TO Obergrenze LOOP
    Anweisungen
  END LOOP;
```

Beispiel:

```
FOR i IN 1 TO 10 LOOP
  a(i):=i*i;
END LOOP
```

### 5.2.14 While-Schleife

Auch die **WHILE–**Schleife ist aus herkömmlichen Programmiersprachen hinreichend bekannt. Die Schleifenbedingung wird dabei am Beginn der Schleife geprüft. Die **WHILE–**Schleife kann daher gegebenenfalls ohne Ausführung übersprungen werden.

Syntax:

```
WHILE Bedingung LOOP
      Anweisung
END LOOP;
```

Beispiel:

```
i:=0;
WHILE (i<10) LOOP
  s<=i;
  i:=i+1;
END LOOP;
```

### 5.2.15 Wait

Die **WAIT–**Instruktion bewirkt eine Pause bis zum Eintreten eines Ereignisses. Dies kann eine Bedingung sein, eine Signaländerung oder einfach der Ablauf einer bestimmten vorgegebenen Pausendauer. **WAIT** kann an jeder Stelle eines Programms auftreten. Es ist jedoch zu beachten, daß dieser Befehl **nicht synthetisiert** werden kann. **WAIT** wird daher vorwiegend beim Erstellen von Testbenches benutzt, wodurch ein sequentieller Ablauf mit klar definierter Geschwindigkeit erzielt wird. Das ist eine Grundvoraussetzung für jegliche Simulation.

Beispiel:

```
WAIT ON a, b;         -- (wartet bis sich a oder b ändert)
WAIT UNTIL x>10;    -- (wartet bis x>10 ist)
WAIT FOR 10 ns;     -- (stoppt das Programm für 10 ns)
WAIT;               -- (Endlosschleife)
```

### 5.2.16 Generate

Wird ein Hauptmodul aus mehreren gleichen Untermodulen aufgebaut, so lassen sich seine Strukturen mit Hilfe der **GENERATE**–Anweisung einfach und übersichtlich gestalten. Abhängig von einer

Bedingung oder einem über eine Schleife definierten Bereich wird eine Reihe von Anweisungen ein- oder mehrfach ausgeführt. Das untenstehende Beispiel verdeutlicht diesen Sachverhalt. Es wird ein Addierer aus mehreren Volladdierern und einem Halbaddierer aufgebaut. Dabei werden die Überträge übernommen und die Überläufe an den nächsten Block weitergegeben.

Syntax:

```
Name:
  FOR i IN Untergrenze TO Obergrenze GENERATE
    Anweisungen
  END GENERATE;

  IF Bedingung GENERATE
    Anweisungen
  END GENERATE;
```

Beispiel:

```
g0: FOR i IN 0 TO 3 GENERATE
      g1: IF i=0 GENERATE
            ha: half_adder PORT MAP (
                a => x(i), b => y(i),
                s => z(i), co => tmp(i+1)
                  );
          END GENERATE;
      g2: IF i> = 1 AND i<=3 GENERATE
            fa: full_adder PORT MAP (
                a => x(i), b => y(i), c => tmp(i),
                s => z(i), co => tmp(i+1)
                  );
          END GENERATE;
    END GENERATE;
```

### 5.2.17 Block

Eine Block-Anweisung enthält einen Satz nebenläufiger Programmteile und ist bei der Organisation eines Designs besonders nützlich. Blocks können geschachtelt werden, um eine Hierarchie zu erzeugen. Objekte, die in einem Block deklariert sind, sind in diesem und in allen untergeordneten Blöcken bekannt.

Syntax:

```
Blockname: BLOCK
BEGIN
  Anweisungen
END BLOCK;
```

## 5.3 Aufbau eines VHDL–Modells

Ein VHDL–Modell besteht mindestens aus einer Schnittstellenbeschreibung (**Entity**), einer oder mehreren Verhaltens- oder Strukturbeschreibungen (**Architecture**) und gegebenenfalls einer oder mehreren Konfigurationen (**Configuration**). Die Aufteilung der Einheiten auf eine oder mehrere Dateien ist willkürlich. Beim Compilieren muß stets die Reihenfolge

Entity => Architecture => Configuration

beachtet werden.

Werden bestimmte Objekttypen, Funktionen oder Prozeduren von mehreren Modellen benötigt, ist es vorteilhaft, sie in einer übergeordneten Einheit (**Package**) abzulegen.

### 5.3.1 Bibliotheken (Libraries)

Erfolgreich compilierte VHDL-Modelle werden in einer **Library** für die nachfolgende Simulation oder für die Weiterverwendung in anderen Modellen gespeichert.

Der Vorteil bei der Nutzung von Bibliotheken besteht darin, daß bereits verfügbare Schaltungsentwürfe nicht explizit von allen Anwendungsprogrammen eingebunden werden müssen. Ohne besondere Deklaration steht immer **Library `*.std`** zur Verfügung. Hier sind die allgemeinen Packages gespeichert. Nach korrektem Compilieren eines VHDL-Quellcodes wird das Ergebnis einer eigenen Bibliothek mit dem Default-Namen **`work`** hinzugefügt.

Syntax für eine Bibliothekdeklaration:

```
LIBRARY library_name_1 {,library_name_2....};
```

#### 5.3.1.1 Use–Anweisung

Mit Hilfe der **Use–**Anweisung werden in Bibliotheken vorhandene Elemente angesprochen und stehen zur Nutzung im aktuellen Design zur Verfügung. Die jeweilige **Library** und gegebenenfalls die **Package**, in der die gewünschten Funktionen und Modelle abgelegt, muß angegeben werden. Auch ohne **Use**–Anweisung sind alle Elemente von **`*.std`** immer sichtbar.

Syntax:

```
USE library_name.all;
USE library_name.element_name;
USE library_name.package_name.all;
USE library_name.package_name.element_name;
```

### 5.3.2 Package

Eine **Package** enthält Anweisungen wie Konstanten- und Typendeklarationen sowie die Beschreibung von Prozeduren und Funktionen, die in mehreren VHDL–Modellen gebraucht werden. Zum Beispiel kann in einer **Package** der zu verwendende Logiktyp (zwei- oder mehrwertige Logik) mit allen korrespondierenden Operatoren definiert werden. So können häufig benötigte Deklarationen durch einmalige Eingabe in verschiedene Projekte eingebunden werden. Packages eignen sich auch, um globale Informationen innerhalb eines komplexen Entwurfs oder innerhalb eines Projekts einmalig festzulegen und damit die Gefahr von widersprüchlichen Definitionen praktisch auszuschließen. Ein Ändern solcher globalen Informationen führt mit minimalem Aufwand zu einer Neukonfiguration eines kompletten Modells. Durch einfaches Ändern der Bitbreite z.B. von Bussen, Multiplexern, Addieren oder Multiplizierern läßt sich ein Entwurf in erheblichem Umfang verändern und somit leicht den unterschiedlichsten Anforderungen anpassen.

Syntax:

```
PACKAGE definitionen IS
  CONSTANT zeit: TIME := 1 ns;
  CONSTANT a: INTEGER:= 0;
  SUBTYPE int_subtype IS INTEGER RANGE -16 TO 15;
  TYPE int IS ARRAY (3 DOWNTO 0) OF int_subtype;
END definitionen;
```

### 5.3.3 Entity

Eine **ENTITY** definiert einen neuen Komponentennamen, Ein- und Ausgänge und die damit verbundenen Deklarationen sowie Unterprogramme und sonstige Vereinbarungen, die für alle dieser **ENTITY** zugeordneten Architekturen gelten sollen. Man kann sagen, daß die Teilmodelle eines komplexen Entwurfs über die **ENTITY** kommunizieren, d.h. über die dort vorgegebene Schnittstellenbeschreibung. Die deklarierten Ein- und Ausgänge sind die **Ports** eines Modells. Name, Datentyp und Signalflußrichtung werden in der **ENTITY** definiert. Außerdem werden hier Parameter deklariert, die dem Modell übergeben werden können, die sogenannten **GENERICS**. So kann beispielsweise die Bitbreite von Ports festgelegt werden.

Mit jeder Port-Deklaration der **ENTITY** werden **Signale** gleichen Typs und gleichen Namens deklariert, die unter Beachtung des jeweiligen **Modus** des Ports in der **ENTITY** und den zugehörigen **Architekturen** verwendet werden können:

- Modus **IN**: die Portleitungen werden als Eingänge deklariert (nur lesbar)
- Modus **OUT**: die Portleitungen werden als Ausgänge deklariert (nur schreibbar)
- Modus **INOUT**: die Portleitungen können als Ein- und als Ausgänge verwendet werden (les- und schreibbar)
- Modus **BUFFER**: die Portleitungen sind Ausgänge, die lesbar, aber nur von einer Quelle beschreibbar sind

Zusätzliche Signale müssen in den jeweiligen **ARCHITECTURES** deklariert werden.

Syntax:

```
ENTITY signale IS
PORT (clock, reset: IN  STD_LOGIC;
      x:   IN   INTEGER;
      y:   IN   STD_LOGIC_VECTOR (7 DOWNTO 0);
      z:   OUT   INTEGER RANGE -8 To 7
        -- kein Semikolon in der letzten Zeile,
        -- sondern erst nach der folgenden Klammer
     );
END signale;
```

### 5.3.4 Architecture

Die **ARCHITECTURE** beschreibt das Verhalten einer Komponente. Hierfür gibt es verschiedene Möglichkeiten, nämlich eine Struktur- oder eine Verhaltensbeschreibung. Beides ist auch kombinierbar.

Im Unterschied zu herkömmlichen Programmiersprachen dient VHDL der Hardwarebeschreibung. Hardwarestrukturen arbeiten sowohl nebenläufig als auch sequentiell. Um solche Eigenschaften zu erfassen, müssen nebenläufige Anweisungen zur Verfügung stehen, die parallel bearbeitet werden.

VHDL–Anweisungen können in parallele und sequentielle eingeteilt werden. Innerhalb des Anweisungsteils einer **ARCHITECTURE** sind Konstrukte wie z.B. die Instantiierung von Komponenten, **BLOCK**- und **GENERATE**–Anweisungen sowie sämtliche Prozesse nebenläufig. Sequentielle Anweisungen (z.B. zum Abarbeiten von Schleifen) hingegen funktionieren wie es von konventionellen Programmiersprachen bekannt ist. Sie dürfen nur innerhalb von Prozessen, Funktionen oder Prozeduren stehen.

Auf Basis einer **ENTITY** können mehrere Architekturen miteinander kombiniert werden, d.h. es können für eine Schnittstellendefinition mehrere Beschreibungen auf unterschiedlichen Abstraktionsebenen oder verschiedene Entwurfsalternativen existieren.

Die in **ARCHITECTURES** deklarierten Signale stellen die Verbindungen zwischen den einzelnen Strukturen her. Sie sind **nur** in der jeweiligen **ARCHITECTURE** bekannt. Von außen kann nicht darauf zugegriffen werden.

Die innerhalb von **ARCHITECTURES** spezifizierten Prozesse, auf die im folgenden eingegangen wird, sind grundsätzlich als nebenläufig zu betrachten.

Syntax:

```
ARCHITECTURE signale_arch OF signale IS
SIGNAL a:    INTEGER RANGE -4 TO 3;
SIGNAL b, c: INTEGER;
BEGIN
B <= a;    -- nebenläufige Anweisungen (nicht getaktet)
C <= a+1;
P1: PROCESS (clock, reset)
  VARIABLE d: INTEGER;
  BEGIN
  ...
END PROCESS
P2: PROCESS (clock, reset)
  VARIABLE e: INTEGER;
  BEGIN
  ...
  END PROCESS
END signale_arch
```

### 5.3.5 Prozesse

Während nebenläufige Anweisungen direkt im Programmteil der **ARCHITECTURE** stehen, enthalten die ebenfalls dort angesiedelten Prozesse Vorgänge, die in Abhängigkeit von Signaländerungen (d.h. ereignisgesteuert) gestartet werden (z.B. Taktflanke).

Ein Prozeß beschreibt das Modellverhalten algorithmisch. Nach dem Schlüsselwort „**PROCESS**" steht in Klammern die sogenannte Empfindlichkeitsliste (**Sensitivity List**). Sie beinhaltet diejenigen Signale auf deren Änderung der Prozeß wartet, d.h. er wird dann - und nur dann - aktiviert, wenn eine solche Änderung aufgetreten ist.

Prozesse sind nebenläufig, das heißt, sie können gleichzeitig ablaufen.

Während Komponentendeklarationen und Instantiierungen in Prozessen nicht erlaubt sind, dürfen hier lokale Variablen deklariert und unmittelbar anschließend verwendet werden.

Wird ein Taktsignal in die Empfindlichkeitsliste geschrieben, dann wird bei der Synthese ein Flipflop generiert, das mit diesem Signal getaktet ist. Durch eine derartige zeitdiskrete Ablaufsteuerung entsteht zwar ein gewisser Schaltungsaufwand, jedoch werden durch die nunmehr synchrone Arbeitsweise klar definierte Zustände erzeugt.

Im folgenden Beispiel ist ein getakteter Prozeß dargestellt.

Syntax:

```
PROCESS (signale)
-- Variablendeklarationen
BEGIN
-- Anweisungen
END PROCESS;
```

**Beispiel:**

```
PROCESS (clock, reset)
-- ggf. Variablendeklaration
BEGIN
  IF reset ='0' THEN
      -- Anweisungen (im Falle eines Resets)
  ELSIF (clock'event AND clock='1')
      -- nebenläufige/sequentielle Anweisungen (getaktet)
  END IF;
END PROCESS
```

### 5.3.6 Variablen

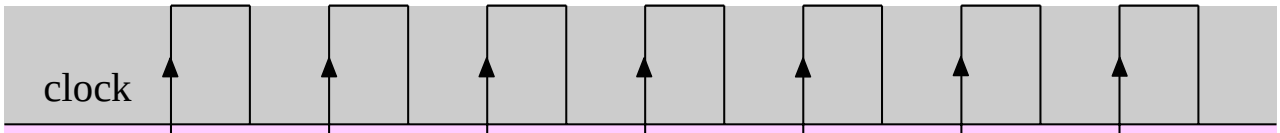
**VARIABLEN** sind Objekte, deren aktueller Wert gelesen und ohne Zeitverzögerung neu zugewiesen werden kann. Der Variablenwert kann sich also im Laufe der Simulation ändern. Variablen existieren nur lokal innerhalb von Prozessen.

In **ARCHITECTURES** können **keine lokalen Variablen** deklariert werden. Dies geschieht ausschließlich innerhalb von Prozessen. Auf diese Prozesse ist dann auch die **Gültigkeit** der lokalen Variablen **beschränkt**.

Das folgende Beispiel erzeugt den in Tabelle 5.3 dargestellten Zeitverlauf. Dabei sei da Signal *x* von außen vorgegeben.

Beispiel:

```
PROCESS (clock)
VARIABLE y1, y2, y3: INTEGER;
BEGIN
  IF (clock'event AND clock='1') THEN
    y1:=x;
    y2:=x+1;
    y3:=y2;
  END IF;
END PROCESS;
```




| | | | | | | | |
|---|---|---|---|---|---|---|---|
| x | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| y1 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| Y2 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| Y3 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

Tabelle 5.3: Zeitverlauf bei Verwendung von Variablen

### 5.3.7 Signale

Neben Variablen und Konstanten, die auch von prozeduralen Programmiersprachen her bekannt sind, verfügt VHDL zusätzlich über Objekte der Klasse **SIGNAL**. Diese Klasse wurde eingeführt, um spezielle Eigenschaften elektronischer Schaltungen modellieren zu können. Änderungen von Signalwerten können z.B. zeitverzögert zugewiesen werden (*a* <= '0' after 10 ns). Damit lassen sich die Laufzeiten von Hardwarekomponenten nachbilden. Signale dienen im wesentlichen dazu, Daten zwischen parallel arbeitenden Modulen auszutauschen. Verschiedene Module können dabei mitunter auf ein und dasselbe Signal schreiben. Diese Eigenschaft gestattet die Modellierung von Bussen.

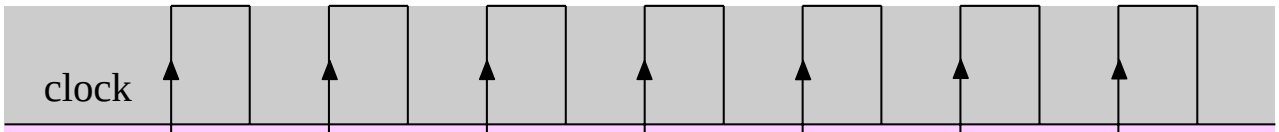
Signale übernehmen Werte **verzögert** nach ihrer Zuweisung. Diese Verzögerung wird auch „Delta–Delay" genannt. Sie entsteht durch die ereignisgesteuerte Simulation, die ein VHDL–Modell so oft abarbeitet, bis alle Signale stabil sind, d.h. alle Änderungen abgeklungen sind. Dieser Zustand wird dem nächsten Schritt übergeben. Verzögerungen kommen bei getakteten Prozessen dadurch zustande, daß die im Prozeß festgelegten Schritte nur einmal beim Auftreten eines gemäß „Sensitivity List" definierten Ereignisses durchlaufen werden.

Zu beachten ist, daß in **ARCHITECTURES** deklarierten Signale nur dort (d.h. lokal) bekannt sind. Externe Modellbestandteile können nicht darauf zugreifen.

Im folgenden Beispiel ergeben sich durch Verwendung von Signalen Unterschiede im Zeitverlauf im Vergleich mit Tabelle 5.3, wo Variablen eingesetzt waren.

Beispiel:

```
SIGNAL x, y1, y2, y3: INTEGER;
...
PROCESS (clock)
BEGIN
  IF (clock'event AND clock='1') THEN
    y1 <= x;
    y2 <= x+1;
    y3 <=y2 ;
  END IF;
END PROCESS
```




| x | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| y1 | 'U' | 1 | 2 | 3 | 4 | 5 | 6 |
| y2 | 'U' | 2 | 3 | 4 | 5 | 6 | 7 |
| y3 | 'U' | 'U' | 2 | 3 | 4 | 5 | 6 |

Tabelle 5.4: Verlauf bei Verwendung von Signalen
('U' steht für nicht initialisiert, d.h. unbekannt)

Man erkennt, daß eine Entscheidung, ob Variablen oder Signale verwendet werden sollen, im Einzelfall sorgfältig abzuwägen ist.

### 5.3.8 Nebenläufige Anweisungen

Nebenläufige Anweisungen verzweigen nicht weiter in Unterkomponenten. Am Beispiel eines Halbaddierers werden die Vorteile dieser Art der Programmierung aufgezeigt. Da VHDL direkt in Hardware umgesetzt wird, sind später alle Abläufe tatsächlich parallel.

Im folgenden realisiert ein komplettes VHDL–Modell einen Halbaddierer, der zwei Bit *a* und *b* addiert.

Beispiel:

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY halfadd IS
PORT(a, b  : IN BIT;
     erg, c: OUT BIT
    );
END halfadd;

ARCHITECTURE halfadd_arch OF halfadd IS
BEGIN
  c<=a AND b;
  erg<=a XOR b;
END halfadd_arch;
```

### 5.3.9 Sequentielle Anweisungen

In sequentiellen oder auch prozeduralen Beschreibungen werden Konstrukte wie Verzweigungen (*IF*, *ELSIF*, *ELSE*), Schleifen (*LOOP*s) und Unterprogrammaufrufe wie *FUNCTION* oder *PROCEDURE* verwendet. Die einzelnen Anweisungen werden dabei nacheinander (sequentiell) abgearbeitet. Solche Beschreibungen ähneln den Quellcodes höherer Programmiersprachen wie C oder Pascal.

Das folgende Beispiel stellt einen Addierer dar, der zwei 8-bit-Vektoren bitweise addiert und ständig das aktuelle Ergebnis ausgibt. An diesem Beispiel werden einige Schwierigkeiten deutlich, die eine Umsetzung sequentieller Strukturen in Hardware mit sich bringt. So kommen z.B. während des Rechenprozesses undefinierte Ausgaben vor. Erst nachdem der Addierer die Rechenoperation beendet hat, ist das Ergebnis stabil. Die Einführung eines Prozesses ist hier notwendig, weil die Schleife (*LOOP*) eine lokale Variable *i* deklariert – ein Vorgang, der nur in Prozessen erlaubt ist.

Beispiel:

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY addierer IS
PORT(a, b : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
     erg : OUT STD_LOGIC_VECTOR (8 DOWNTO 0)
    );
END addierer;

ARCHITECTURE addierer_arch OF addierer IS
```

```
SIGNAL c: STD_LOGIC:=´0´;

BEGIN

PROCESS (a, b)
 FOR i IN 0 TO 7 LOOP
  erg(i)<=a(i) XOR b(i) XOR c;
  c<=(a(i) AND b(i) AND NOT(c)) OR (NOT(a(i)) AND b(i) AND c)
  OR (a(i) AND NOT(b(i)) AND c);
 END LOOP;
 erg(8)<=c;
END PROCESS;

END addierer_arch;
```

### 5.3.10 Verhaltensmodellierung

Bei der Verhaltensmodellierung wird das Verhalten einer Komponente durch die Reaktion der Ausgangssignale auf die Eingangssignale unter Verwendung verschiedener Operatoren, Signalzuweisungen oder Bedingungen beschrieben.

Um in einer **ARCHITECTURE** Verhaltensmodellierung durchzuführen, sind Prozesse zu verwenden. Parallele Arbeitsweise kann durch mehrere Prozesse erreicht werden.

### 5.3.11 Strukturale Modellierung

Diese Modellierung bietet den Vorteil, daß in der Regel kein umfangreiches zusammenhängendes Programm entsteht, sondern ein Design auf mehrere kurze Einheiten aufspalten werden kann, die getrennt compiliert werden. Somit bleibt die Übersicht erhalten, und kleine Änderungen erfordern keinen hohen Zeitaufwand. In Top–Down–VHDL–Designs verwendet man gerne die strukturale Modellierung, weil bereits compilierte Designs damit leicht eingebunden werden können. Es entstehen gut überschaubare hierarchische Strukturen in Form von Bäumen.

Beispiel:

Der Einfachheit halber entsprechen im folgenden Beispiel die Modelle XOR2 und AND2 den Operatoren XOR und AND.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE WORK.ALL;

ENTITY  halfadd IS
PORT (a, b:   IN BIT;
      erg, c: OUT BIT
     );
END halfadd;

ARCHITECTURE halfadd_arch OF halfadd IS

COMPONENT XOR2
PORT(x, y: IN BIT; -- Komponentendeklaration
```

```
        z: OUT BIT
    );
END COMPONENT;

COMPONENT AND2
PORT (x, y: IN BIT;
         z: OUT BIT
     );
END COMPONENT;

BEGIN
U0: XOR2 PORT MAP (a, b, erg); -- verwendete Komponenten
U1: AND2 PORT MAP (a, b, c);

END halfadd_arch;
```

Bild 5.30 zeigt den Aufbau eines VHDL–Modells in strukturaler Beschreibung. Dabei sind drei Komponenten eingebunden, die Teil des Modells werden.

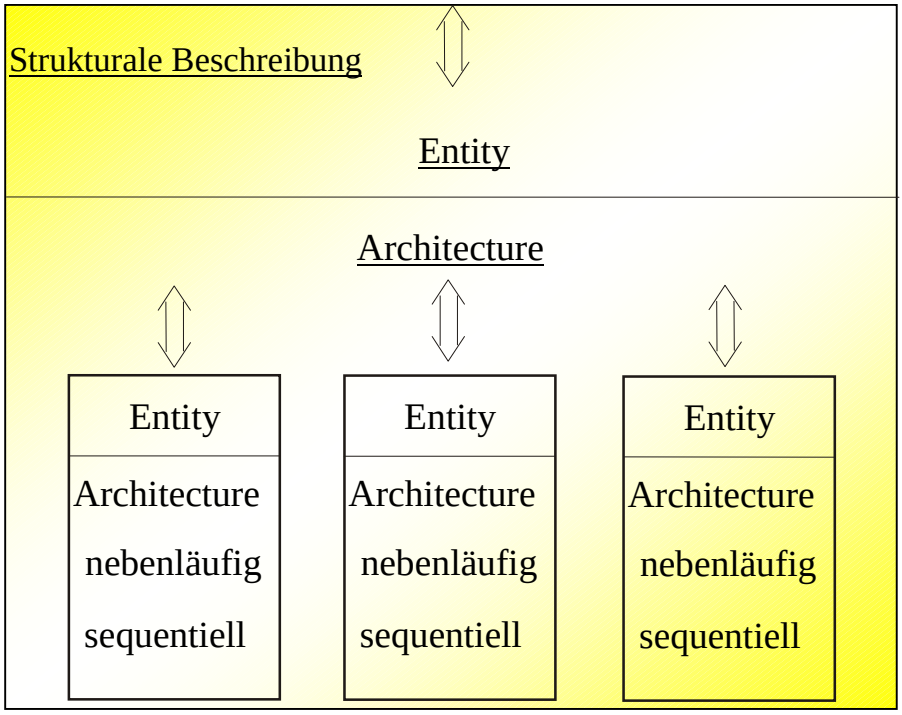



Bild 5.30: Aufbau eines VHDL–Modells aus struktureller Sicht

### 5.3.12 Configuration

In der **CONFIGURATION** wird zunächst festgelegt, welche **ARCHITECTURE** zu verwenden ist. Bei strukturalen Beschreibungen kann zusätzlich angegeben werden, aus welchen Bibliotheken die einzelnen Submodule entnommen werden, wie sie eingesetzt (verdrahtet) werden, und welche Parameter (**GENERICS**) übergeben werden sollen. Zu einer **ENTITY** können mehrere Konfigurationen gehören. Dadurch wird der Entwickler beim Experimentieren mit verschiedenen Varianten unterstützt, weil er unterschiedliche Architekturen und Komponenten in der Konfiguration zusammenstellen kann.

Beispiel:

```
CONFIGURATION signale_config OF signale IS
FOR simulate
  FOR prog: signale USE ENTITY work.signale(signale_arch);
       END FOR;
END FOR;
END signale_config;
```

### 5.3.13 Testbench

Zu einem vollständigen VHDL–Design gehört auch eine Testumgebung, „Testbench“ genannt. Sie dient der Überprüfung entwickelter Beschreibungen. Eine Testbench bindet den Entwurf als Komponente ein und weist den Eingangssignalen Werte zu. Anhand der Werte der Ausgangssignale kann die Funktion und die Korrektheit des Entwurfs untersucht werden.

Oft ist es erforderlich, die Instruktionen in einer Testbench sequentiell mit einer definierten Geschwindigkeit abzuarbeiten. Um dies zu erreichen, wird der Wait–Befehl verwendet. So können Testbenches entwickelt werden, die bei jedem Takt die Eingangssignale neu belegen.

Die Testbench muß stets als letzte Einheit compiliert werden, bevor sie in den Simulator geladen werden kann.

Im folgenden Beispiel ist eine Testbench dargestellt, die das Modell „testdesign“ als Komponente einbindet und sein Eingangssignal belegt.

Beispiel:

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY testbench IS   -- keine Portdeklaration
END testbench;

ARCHITECTURE simulate OF testbench IS

COMPONENT testdesign
PORT (x: IN  INTEGER;
      y: OUT INTEGER
     );
END COMPONENT;

SIGNAL eingang, ausgang: INTEGER;

BEGIN
  test : testdesign PORT MAP (eingang, ausgang);
PROCESS
BEGIN
  LOOP                  -- Endlosschleife
    eingang <= 1;
    WAIT FOR 20 ns;
    eingang <=0;
    WAIT FOR 20 ns;
```

```
  END LOOP;
END PROCESS;
END simulate;

CONFIGURATION test_simulate OF testbench IS
  FOR simulate
    FOR test: testdesign USE ENTITY
    work.testdesign(testdesign_arch);
    END FOR;
  END FOR;
END test_simulate;
```

### 5.3.14 Die Datentypen **std_ulogic** und **std_logic**

Im IEEE–Standard 1164-1993 wird eine **neunwertige** Logik eingeführt, die die in Tabelle 5.5 angegebenen Werte umfaßt. In der VHDL–Bibliothek
**ieee.std_logic_1164** werden diese Werte durch die Deklarationen

```
type std_ulogic IS ('U','X','0','l','Z','W','L','H','-');
type std_logic IS ('U','X','0','l','Z','W','L','H','-');
```

zu den Datentypen **std_ulogic** und **std_logic** zusammengefaßt.

| Wert | Bedeutung | Verwendung |
|---|---|---|
| 'U' | nicht initialisiert | nicht initialisiertes Signal im Simulator |
| 'X' | undefiniert | mehr als einen aktiven Signaltreiber ⇒ Buskonflikt liegt vor |
| '0' | starke logische '0' | entspricht einem auf Null liegenden Signal der Breite 1bit |
| '1' | starke logische 'l' | entspricht einem auf Eins liegenden Signal der Breite 1bit |
| 'Z' | hochohmig | Tri-State Ausgang |
| 'W' | schwach unbekannt | Simulator erkennt Buskonflikt zwischen 'L'- und 'H'-Pegel |
| 'L' | schwache logische '0' | z.B. Ausgang mit Pull-Down-Widerstand (Open Source) |
| 'H' | schwache logische '1' | z.B. Ausgang mit Pull-Up-Widerstand (Open Drain) |
| '-' | don't care | Zustand ohne Bedeutung; kann für Minimierung verwendet werden |

Tabelle 5.5: Wertevorrat der Standardlogik-Datentypen

Von diesem Datentyp wurde auch ein Datentyp X01 mit dreiwertiger Logik sowie ein Datentyp X01Z jeweils mit dem im Namen angegebenen Wertevorrat abgeleitet. Diese stellen jeweils Untermengen des **std_(u)logic**-Datentyps dar.

Worin besteht nun der Unterschied zwischen den Datentypen **std_logic** und **std_ulogic**? Wie bei den bisher bekannten Datentypen darf für Signale des Typs **std_ulogic** jeweils nur ein Treiber existieren, d.h. diese Signale dürfen nur in einem Prozeß bzw. einer nebenläufigen Anweisung eine Wertzuweisung erfahren, auch wenn die Signalzuweisung im VHDL–Code zu unterschiedlichen Zeitpunkten erfolgt. Zu berücksichtigen ist nämlich, daß alle Treiber ständig einen Signalwert liefern. Signalkonflikte, die in der Hardware zu Schaltungsfehlern führen würden, wer-

den bei diesem Datentyp also nicht aufgelöst (das „**u**“ in der Typbezeichnung drück dies aus, es steht für **unresolved data type**). Der Design-Compiler erkennt solche Zustände und bricht mit einer Fehlermeldung ab.

Beim Datentyp **`std_logic`** ist das anders. Hier dürfen durchaus mehrere Treiber für ein Signal existieren. Dieser Datentyp wird insbesondere bei bidirektionalen Bussen benötigt, auf denen die Daten von mehreren Sendern kommen können. Die Entscheidung, welcher Treiber sich durchsetzt, trifft der Simulator auf Basis der im **1164 IEEE–Standard** definierten Auflösungsfunktion (***Resolution Function***).

Für den Anwender ist es wichtig zu wissen, wie die Auflösungsfunktion des Datentyps **`std_logic`** wirkt. Sie ist in Form der in Tabelle 5.6 angegebenen Matrix im **IEEE 1164-Paket** abgelegt. Der resultierende Signalwert, der sich beim gleichzeitiger Aktivität zweier Signaltreiber ergibt, ist dort zu entnehmen. Die aktuellen Signalwerte der beiden gleichzeitig aktiven Treiber sind jeweils in der ersten Zeilen bzw. als Spalte einzusetzen. Das Resultat ist im Kreuzungspunkt abzulesen.

### Auflösungsfunktion für den Datentyp `std_logic`

(Aus Gründen der Übersicht wurden die Anführungszeichen in der Tabelle weggelassen)

Signal 1 ⟶ (horizontal) / Signal 2 ⟶ (vertical)

| Signal 2 \ Signal 1 | U | X | 0 | 1 | Z | W | L | H | - |
|---|---|---|---|---|---|---|---|---|---|
| U | U | U | U | U | U | U | U | U | U |
| X | U | X | X | X | X | X | X | X | X |
| 0 | U | X | 0 | X | 0 | 0 | 0 | 0 | X |
| 1 | U | X | X | 1 | 1 | 1 | 1 | 1 | X |
| Z | U | X | 0 | 1 | Z | W | L | H | X |
| W | U | X | 0 | 1 | W | W | W | W | X |
| L | U | X | 0 | 1 | L | W | L | W | X |
| H | U | X | 0 | 1 | H | W | W | H | X |
| - | U | X | X | X | X | X | X | X | X |

**jeweiliges „Auflösungsergebnis“ bei „Kollision“**

Tabelle 5.6: Auflösungsfunktion für den Datentyp **`std_logic`**

(Aus Gründen der Übersicht wurden die Anführungszeichen in der Tabelle weggelassen)

Die Auflösungsfunktion wird im Simulator automatisch aktiviert, sobald ein Signal durch mehr als einen Prozeß beeinflußt wird. Das bedeutet, daß versehentliche Mehrfachzuweisungen an ein Signal beim Datentyp **`std_logic`** vom Compiler während der Entwurfsphase nicht beanstandet werden. Tabelle 5.6 ist zu entnehmen, daß ein Signal, das nicht initialisiert (‘U‘) oder undefiniert (‘X‘) ist, beim Zusammenschalten durch kein anderes Signal verändert werden kann.

Da jede boolesche Verknüpfung von ‘U‘ bzw. ‘X‘ mit einem anderen Signalwert wie z.B. ‘1‘ oder ‘0‘ als Resultat stets wieder einen ‘U‘- bzw. ‘X‘–Zustand generiert, muß zunächst immer der unde-

finierte Zustand beseitigt werden, bevor in der weiteren Logik verwendbare ‘l‘ oder ‘0‘ Pegel zur Verfügung stehen. Ein einfaches Beispiel ergibt sich bei einen Flipflop, wo der D-Eingang mit einem Ausgang (in der Regel $\overline{Q}$) verbunden ist. Da alle Signale vom Typ **`std_(u)logic`** im Simulator mit ‘U‘ initialisiert werden, sind die Flipflop-Ausgänge solange undefiniert, bis ein **Reset** erfolgt ist.

Der undefinierte Zustand ‘X‘ entsteht immer bei einem Buskonflikt, z.B. wenn gleichzeitig ein Sender eine ‘0‘ und ein anderer eine ‘l‘ auf den Bus legt. Der hochohmige Zustand ‘Z‘ muß durch einen Tri–State-Treiber explizit definiert werden. ‘Z‘ wird durch jeden anderen Signalwert überschrieben.

Im VHDL–Quellcode können die Datentypen **`std_ulogic`** und **`std_logic`** verwendet werden, wenn vor der **ENTITY**–Deklaration die IEEE–Bibliothek mit Hilfe einer **LIBRARY**-Anweisung deklariert wurde. Zusätzlich muß durch eine **USE–**Anweisung angeben werden, daß alle Komponenten des **`std_logic_1164`**-Pakets verwendet werden sollen. Die beiden folgenden Zeilen müssen sich demnach vor jeder Entity befinden, in der ein **`std_(u)logic`**-Datentyp benutzt werden soll:

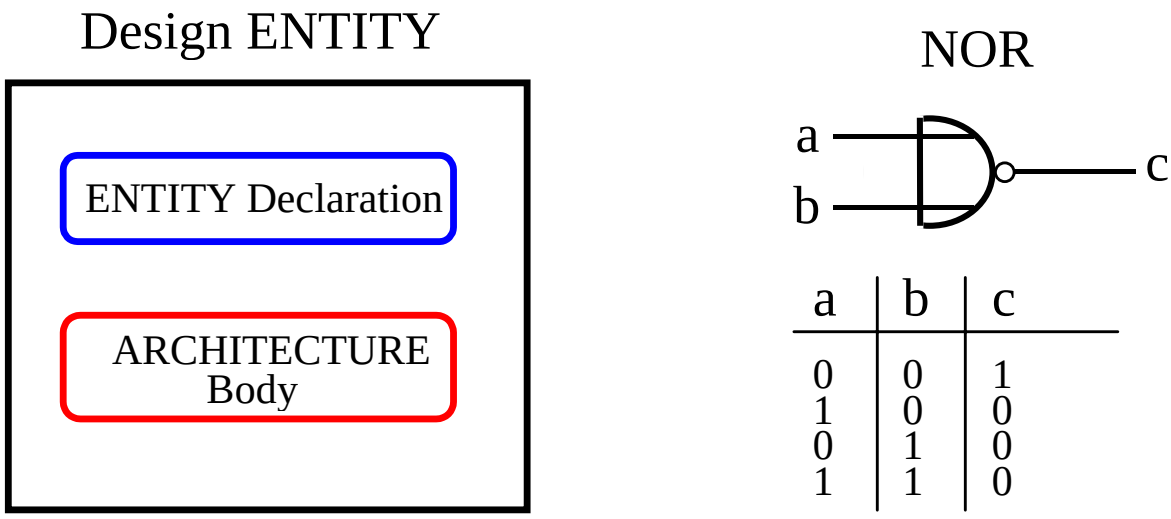
```
library ieee;
use ieee.std_logic_1164.all;
```

Während sich die **LIBRARY**-Anweisung auf alle Einheiten einer VHDL–Datei bezieht, ist bei der **USE**–Anweisung zu beachten, daß diese vor jeder einzelnen **ENTITY**–Deklaration wiederholt werden muß, die entsprechende Datentypen verwendet. Falls in einer VHDL–Datei eine **ARCHITECTURE** angegeben wird, zu der die zugehörige **ENTITY** in einer anderen Datei abgelegt ist, dann müssen vor dieser **ARCHITECTURE LIBRARY**- und **USE–**Anweisung plaziert werden.

## 5.4 Beschreibung und Synthese einfacher Digitalschaltungen - VHDL-Simulation

### 5.4.1 VHDL–Modellbeispiele

- **Ein einfaches Einführungsbeispiel: NOR–Gatter**




| a | b | c |
|---|---|---|
| 0 | 0 | 1 |
| 1 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 1 | 0 |

Bild 5.31: Entity und Architecture als Grundelemente eines VHDL–Modell

```
ENTITY NOR_Gate IS
PORT (a, b:IN bit;
            c:     OUT bit)
END NOR_Gate;


ARCHITECTURE Verknuepfung OF NOR_Gate IS
BEGIN
c<=  '1' WHEN a = '0' AND b = '0' ELSE
      '0' WHEN a = '0' AND b = '1' ELSE
      '0' WHEN a = '1' AND b = '0' ELSE
      '0' WHEN a = '1' AND b = '1' ELSE
      '0'
END Verknuepfung
```

```
BEGIN
c <= a NOR b
END Verknuepfung
```

← alternativ

- **Ein 4bit-Addierer**

```
ENTITY Adder IS

PORT (a, b:      IN INTEGER RANGE 0 TO 15;
            c:     OUT INTEGER RANGE 0 TO 15)

END Adder;

ARCHITECTURE Addition OF Adder IS
BEGIN
c <= a + b;
END Addition;
```

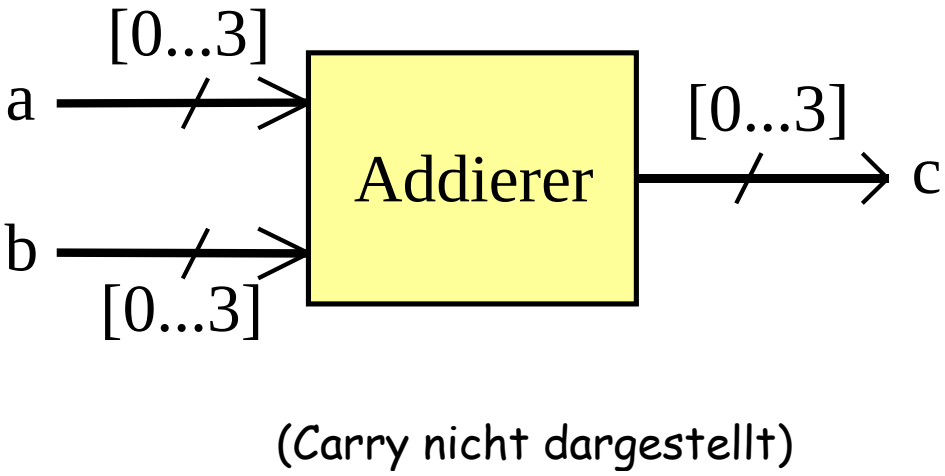



Bild 5.32: Die Entity des Addieres als Blockschaltbild

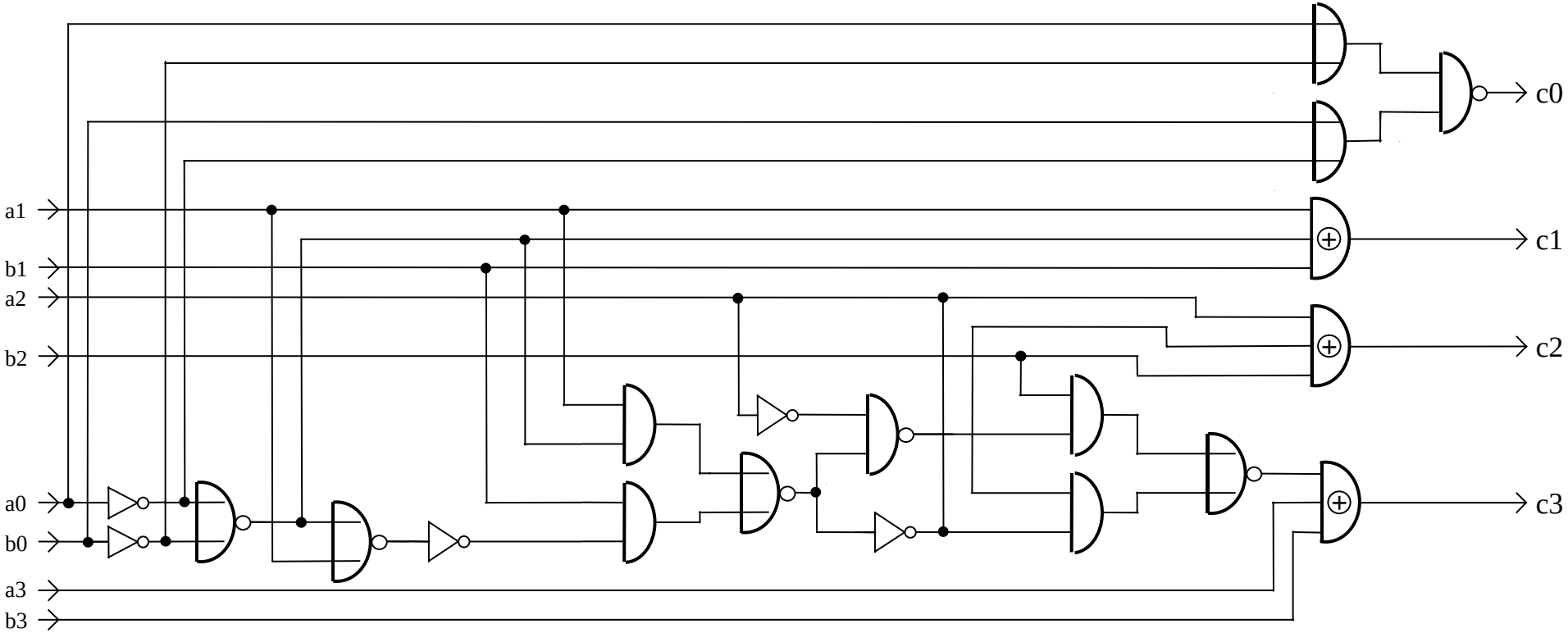


Bild 5.33: Ergebnis der VHDL–Synthese für den 4bit-Addierer

- **Probleme von Beschreibungen auf algorithmischer Ebene**

Der folgende Ausschnitt aus der Architektur eines hier nicht näher spezifizierten Schnittstellenbausteins ist eine Beschreibung auf algorithmischer Ebene. Der Baustein soll immer dann, wenn er von einem Controller eine Aufforderung `adr_request` erhält, eine interne Adresse `int_adr` frühestens nach 10 ns auf den Bus, d.h. auf `bus_adr` legen. Die Beschreibung enthält keine Angaben über die Schaltungsstruktur und weder Takt- oder Rücksetzsignale.

```
ARCHITECTURE algo_ebene OF io_ctrl IS
BEGIN
    ....
    schreib_dat_alg: PROCESS
    BEGIN
        WAIT UNTIL adr_request = '1';
        WAIT FOR 10ns;
        bus_adr <= int_adr;
    END PROCESS schreib_dat_alg;
END algo_ebene;
```

**Register-Transfer-Ebene**

```
ARCHITECTURE rt_ebene OF io_ctrl IS
BEGIN
     .....
      SR_DAT: PROCESS (clk)
          VARIABLE zwischen: boolean;
      BEGIN
          IF rising_edge(clk) THEN
             IF ((adr_request = '1') AND (zwischen = false)) THEN
             zwischen:= true;
             ELSIF (zwischen = true) THEN
                  bus adr <= int_adr;
                  tmp := false;
             END IF;
          END IF;
      END PROCESS SR_DAT;
END rt_ebene;
```

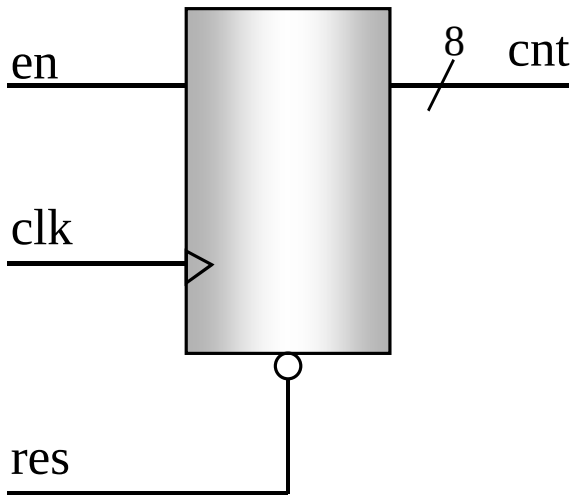
Um das Beispiel auf der Register-Transfer-Ebene darzustellen, wurde ein Taktsignal clk hinzugefügt, so daß die Abläufe in Abhängigkeit von diesem Signal beschrieben werden können

Wenn bei einer **ersten** aktiven Taktflanke clk ein gesetztes adr_request-Signal entdeckt wird, wird zunächst die temporäre Variable zwischen gesetzt, damit bei der **nächsten** aktiven Taktflanke – **Wartezeit** - die Adresse auf den Bus geschrieben werden kann. Durch geeignete Wahl der **Taktperiode** ist sicherzustellen, daß die Wartezeit von mindestens 10ns eingehalten wird. Im Gegensatz zur algorithmischen Ebene wird hier ein zeitliches Schema für den Ablauf vorgegeben und implizit eine Schaltungsstruktur beschrieben.

- **8bit-Vorwärts-Binärzähler mit asynchronem Reset und Enable**

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use IEEE.std_logic_arith.all;
entity Zaehl is
port      (clk, res, en : in std_logic;
cnt: out std_logic_vector(7 downto 0);
end Zaehl;

architecture COUNT of Zaehl is
signal icnt: std_logic_vector(7 downto 0);
begin
process (clk, en, icnt, res)
      begin
      if (res='0') then
                  icnt <= (others => '0');
      elsif (clk'event and clk='1') then
      if (en = '1') then
      icnt <= icnt+'1';
        end if;
end process;
      cnt <= icnt
end COUNT
```



- **Finite State Machine**

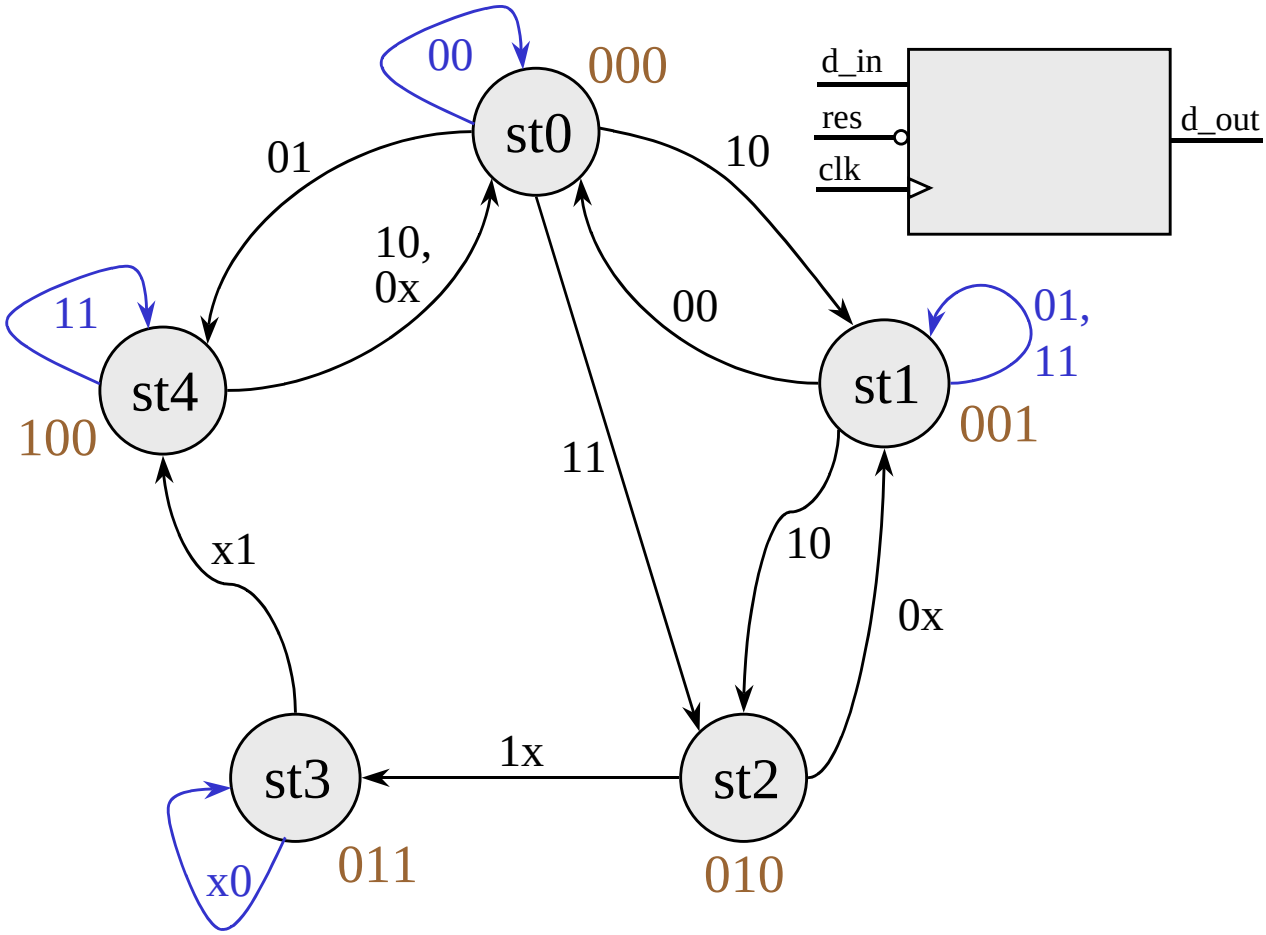



Bild 5.34: Beispiel des FSM-Modells eines Automaten

## VHDL-Beschreibung

```
library IEEE;
use ieee.std_logic_1164.all;

entity M_AUT is
port (clk, res: in std_logic;
d_in: in std_logic_vector (1 downto 0);
d_out: out std_logic_vector (2 downto 0);
end M_AUT;

architecture FUNCT of M_AUT is
        type zustands_werte is (st0, st1, st2, st3, st4);
        signal akt_zust, folg_zust: zustands_werte;
        begin

-- Definition der FSM-Register
Zust_Reg: process (clk, res)
        begin

        if (reset = '0') then
        akt_zust <= st0;
                elsif (clk ='1' and clock'event) then
                akt_zust <= folg_zust;
        end if;
```

```
end process Zust_Reg;

-- Kombinatorik der FSM
FSM: process (akt_zust, d_in)
begin
case akt_zust is

        when st0 =>
        case d_in is
            when"00"  => folg_zust <= st0;
            when"01"  => folg_zust <= st4;
            when"10"  => folg_zust <= st1;
            when"11"  => folg_zust <= st2;
            whenothers      => null;
end case;

        when st1 =>
        case d_in is
            when     "00"          => folg_zust <= st0;
            when     "10"          => folg_zust <= st2;
            when     others => folg_zust <= st1;
end case;
        when st2 =>
        case data_in is
            when     "00"          => folg_zust <= st1;
            when     "01"          => folg_zust <= st1;
            when     "10"          => folg_zust <= st3;
            when     "11"          => folg_zust <= st3;
            when     others        => null;
end case;
        when st3 =>
      case d_in is
            when     "01"          => folg_zust <= st4;
            when     "11"          => folg_zust <= st4;
            when     others        => folg_zust <= st3;
end case;

        when st4 =>
        case d_in is
            when     "11"            => folg_zust <= st4;
            when     others          => folg_zust <= st0;
end case;

        when others => folg_zust <= st0;
end case;
end process FSM;

-- Moore-Ausgabewerte (hängen nur von akt_zust ab)

AUSGABE: process (akt_zust)
        begin
        case akt_zust is
        when st0 => d_out <= "000";
        when st1 => d_out <= "001";
        when st2 => d_out <= "010";
```

```
        when st3 => d_out <= ”011”;
        when st4 => d_out <= ”100”;
        when others => d_out <=”000”;
end case;
end process AUSGABE;
end FUNCT;
```

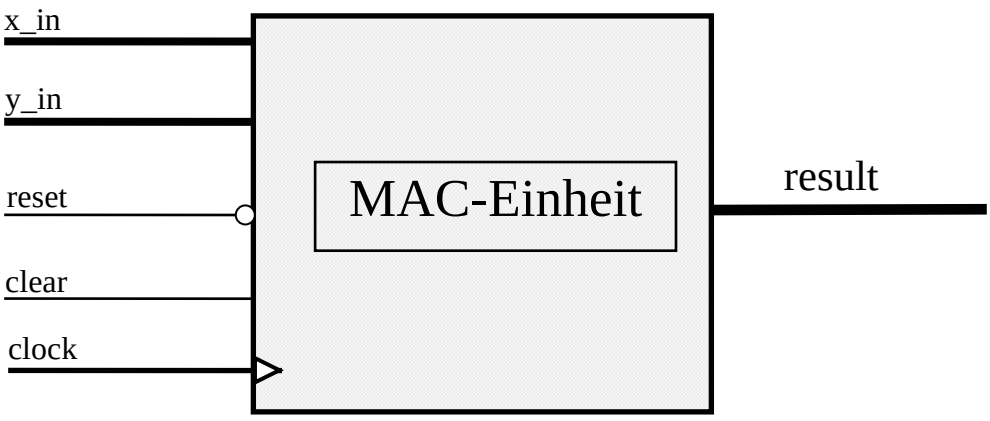
- **VHDL-Beschreibung einer MAC-Unit**

Entity

```
ENTITY mac_unit IS
PORT (clock, reset, clear : IN bit;
x_in, y_in        : IN integer range -8 to 7; -- 4bits
result            : OUT integer range -128 to 127); -- 8bits
END mac_unit;
```

Architecture

```
ARCHITECTURE mac_calc OF mac_unit IS
SIGNAL  int_bus: integer range -128 to 127;
BEGIN
        PROCESS (clock, reset)
    BEGIN
            IF reset = ‘0’ THEN
                int_bus <= ‘0’;
        ELSEIF clock= ‘1’ AND clock’event THEN
        IF clear = ‘1’                                  THEN
                    int_bus <= int_bus + x_in * y_in;
            ELSE
                    int_bus <= x_in * y_in;
            END IF;
        END IF;
        END PROCESS;
    result <= int_bus;
    END mac_calc;
```



### 5.4.2 Simulation von VHDL-Entwürfen

Es wird eine einfache Methode vorgestellt, mit der VHDL-Entwürfe syntaktisch und funktional überprüft werden können. Da das Erlernen von VHDL wird deutlich erleichtert, wenn eigene Entwürfe sofort auf Korrektheit überprüft werden können.

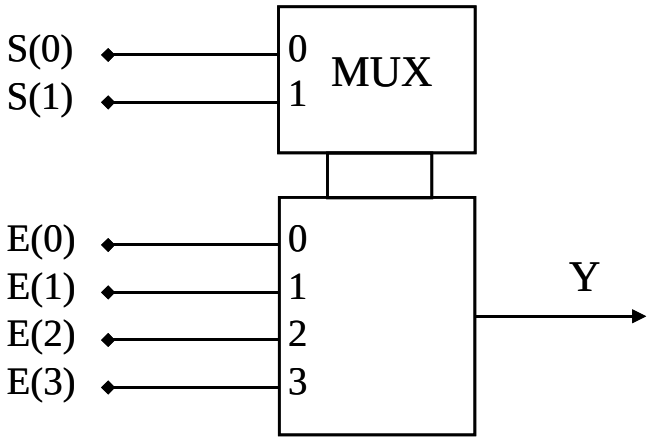
Es wird davon ausgegangen, daß keine komfortable Umgebung zur interaktiven Eingabe der Stimuli zur Verfügung steht. Daher wird der prinzipielle Aufbau einer einfachen Testumgebung (Testbench) erläutert. Dabei können alle Entwurfseinheiten, also auch die Testbench, zusammen in einer Datei mit der Erweiterung ***.VHD** abgespeichert werden. Bessere Übersicht erhält man jedoch, wenn die einzelnen **entity/architecture–**Paare jeweils in einer eigenen Datei abgelegt werden. Nach erfolgreicher Compilation der Datei, die das zu untersuchende Modell enthält, wird dessen Objektcode in einer VHDL–Bibliothek mit dem vordefinierten Namen **WORK** abgelegt. Anschließend muß eine Datei angelegt und compiliert werden, die die **entity** und **architecture** der Testbench enthält.

Diese **Testbench–Entity** enthält keinerlei Schnittstellensignale. Die Schnittstellensignale des zu untersuchenden Modells werden vielmehr als lokale Signale in der **Testbench–Architecture** deklariert. Diese enthält eine Komponente, die der zu untersuchenden **entity** entspricht und an die die lokalen Testbenchsignale übergeben werden. Die Definition des Zeitverlaufs der Stimuli erfolgt durch nebenläufige Anweisungen, in denen die Zeitpunkte der Signalübergänge festgelegt sind. Der syntaktische Aufbau einer Testbench sieht wie folgt aus:

```
entity TEST is
end TEST;
architecture TESTBENCH of TEST is
signal      <Signalliste>;              -- alle Schnittstellensignale der
                                        -- zu untersuchenden entity
component <Komponentenname>             -- identisch mit Entityname der
                                        -- zu untersuchenden entity
            <Port-Liste>                -- identisch mit der Port-Liste
                                        -- der zu untersuchenden entity
end component;
for all:    <Komponentenname>  use  entity  work.<Entityname>
            (<Architekturname>);
begin
            -- nebenläufige Signalzuweisungen an die Stimuli
            -- mit Angabe des Zeitpunkts der Signalübergänge
[<Bezeichner>]: <Komponentenname>
port map (< Liste der angeschlossenen aktuellen Signale>);
end TESTBENCH;
```

Als Beispiel wird eine Testbench für folgenden 4-zu-1 Multiplexer angegeben.

```
entity MUX4X1 is
port (    S: in bit_vector(1 downto 0); -- 2 Select-Eingänge S(0), S(1)
          E: in bit_vector(3 downto 0); -- 4bit Eingangsdaten E
          Y: out bit);                  -- 1-bit-Ausgang Y
end MUX4X1;
architecture VERHALTEN of MUX4X1 is
begin
with S select-- Auswahlsignal
          Y <=      E(0) when "00",
                    E(1) when "01",
                    E(2) when "10",
                    E(3) when "11",
end VERHALTEN
```




Die in der folgenden Testbench deklarierten Signale S1, E1 und Y1 werden als aktuelle Signale an die als Komponente C1 plazierte **entity** MUX4X1 übergeben. Zum Zeitpunkt t=0 wird der Eingangsvektor E1 mit "1010" und zum Zeitpunkt t=400ns mit "0101" vorbelegt. Das Auswahlsignal S1 durchläuft mit einer Impulsdauer von 100ns zweimal die Sequenzen "00", "01", "10" und "11", womit sich eine Simulationsdauer von 800ns ergibt.

```
-- Testbench für 4x1 multiplexer
entity TEST is
end TEST;
architecture VERHALTEN of TEST is
signal S1: bit_vector(l downto 0);
signal E1: bit_vector(3 downto 0);
signal Yl: bit;
component MUX4X1 is
          port(     S: in bit_vector(l downto 0);
                    E: in bit_vector(3 downto 0);
                    Y: out bit);
end component;
```

```
for all: MUX4X1
use entity work.MUX4X1(VERHALTEN); -- Modell aus Bibliothek
        begin
        E1 <= "1010", "0101" after 400 ns;
        S1 <= "00", "01" after 100 ns,"10" after 200 ns,
        "11", after 300 ns, "00" after 400 ns, "01" after 500 ns,
        "10" after 600 ns, "11" after 700 ns;
C1:     MUX4X1 port map(S1, E1, Y1);
end VERHALTEN;
```

Bei Verwendung der Zeiteinheiten ist zu beachten, daß zwischen dem Zahlenwert und der Einheit ein Leerzeichen stehen muß.

Der wesentliche Vorteil einer rein VHDL–basierten Testumgebung liegt in der Unabhängigkeit von der Kommandosprache eines speziellen Simulators und in der einheitlichen Beschreibung von Testobjekt und Quellen der Anregungssignale (Stimuli). Eine entscheidende Qualitätssteigerung für den Test komplexer Entwürfe entsteht dadurch, daß die Testumgebung um Funktionen erweitert werden kann, die ergänzend zu den Stimuli einen Soll-Ist-Vergleich der Ausgangssignale durchführen.

Im folgenden Beispiel werden Varianten der Zuweisung bzw. Abfrage von **bit_vector–**Konstanten in einer Wahrheitstabelle vorgestellt. Dazu dient ein Impulsmustergenerator mit vier Ausgangskanälen **A**, der 8 Signalmuster erzeugen kann, die durch einen 3bit breiten, Eingangssignalvektor **E** ausgewählt werden. Diese Muster sind in Tabelle 5.7 dargestellt. Im VHDL–Code sieht man, daß dem Bit-String die Basis der Zahlensysteme voranzustellen ist.

```
-- Vierkanal-Impulsgenerator mit Wahrheitstabelle
entity IMPULSGEN is
port (  E: in bit_vector(2 downto 0);
        A: out bit_vector(3 downto 0));
end IMPULSGEN;
architecture W_TAB of IMPULSGEN is
begin
        P1: process (E)  begin
            case E is
            when o"0" => A <= x"7";
            when o"1" => A <= x"A";
            when o"2" => A <= x"3";
            when o"3" => A <= x"F";
            when o"4" => A <= x"6";
            when o"5" => A <= x"C";
```

| E | A |
|---|---|
| 0 | 7h |
| 1 | Ah |
| 2 | 3h |
| 3 | Fh |
| 4 | 6h |
| 5 | Ch |
| 6 | 0h |
| 7 | Eh |

Tabelle 5.7: 8 Signalmuster

```
            when o"6" => A <= x"0";
            when o"7" => A <= x"E";
end case;
end process Pl;
end W_TAB;
```

Die Buchstaben zur Kennzeichnung von Zahlenbasen sind:

| Basis | Buchstabe | Beispiel |
|---|---|---|
| Dual | B oder b | b'1010_1100' |
| Oktal | O oder o | o'1_4_7' |
| HEX | X oder x | x'AF_fe' |

### 5.4.3 Simulationsspezifische Prozesse für Testumgebungen

Hier werden Prozesse vorgestellt, mit denen der Entwurf von Testumgebungen vereinfacht werden kann. Diese nicht synthesefähigen Prozesse ergänzen eine zu synthetisierende **architecture** um spezielle Stimuli–Prozesse zu einer einfachen Testbench. Dabei werden unterschieden:

- Prozesse, die diskrete Stimuli beschreiben
- Prozesse, die periodische Stimuli beschreiben

Bei der hier vorgestellten Methode müssen die Eingangssignale der zu synthetisierenden **entity** während der Testphase aus der **port**–Anweisung entfernt werden und als lokale Signale deklariert sein. Nach Abschluß des Tests müssen die Erweiterungen wieder entfernt werden.

Als Beispiel dient der obige Impulsmustergenerator.

```
-- Vierkanal-Impulsgenerator mit Testbench
entity IMPULSGEN is
port (A: out bit_vector(3 downto 0));
end IMPULSGEN;

architecture W_TAB of IMPULSGEN is
signal E: bit_vector(2 downto 0); -- lokales Signal für Testbench
begin
-- zu synthetisierender Prozeß
Pl: process (E)
            begin
            case E is
            when o"0" => A <= x"7";
```

```
            when o"1" => A <= x"A";
            when o"2" => A <= x"3";
            when o"3" => A <= x"F";
            when o"4" => A <= x"6";
            when o"5" => A <= x"C";
            when o"6" => A <= x"0";
            when o"7" => A <= x"E";
end case;
end process Pl;
-- nicht synthetisierbare Testbench–Prozesse
STIMULI: process        -- nichtperiodische Stimuli
        begin
        E(1) <=  '0', 'l' after 100 ns, '0' after 200 ns, 'l' after 300 ns;
        E(2) <='0', '1' after 200 ns;
        wait;           -- keine weiteren Signaländerungen
end process STIMULI;
TAKTGEN: process
            begin       -- periodischer Stimulus
            E(0) <= '0';
            wait for 50 ns;
            E(0) <= '1';
            wait for 50 ns;
end process TAKTGEN;
-- Ende der Testbench
end W_TAB;
```

Aufgabe der Testumgebung ist es, alle Varianten des Eingangsvektors E zu generieren. Dafür wird das Eingangssignal E aus der ursprünglichen **entity** entfernt und lokal deklariert. Dem niederwertigen Bit E(0) wird in dem periodischen Stimulus TAKTGEN alle 50ns ein neuer Wert zugewiesen. Da der ohne Empfindlichkeitsliste formulierte Prozeß nach Ablauf der zweiten Wartephase erneut mit der Ausführung beginnt, wird ein Signal mit einer Periodendauer von 100ns und einem Tastverhältnis von 50% erzeugt.

Die Signalwechsel der beiden anderen Bits E(l) und E(2) werden durch verzögerte, unbedingte Signalzuweisungen innerhalb des Prozesses STIMULI realisiert. Wie das Beispiel zeigt, ist es möglich, mehrere Signalübergänge in einer einzigen unbedingten Signalzuweisung durch Verwendung des Schlüsselworts **after** zu definieren. Wichtig ist jedoch, daß alle Signalübergänge eines Signals

in **einer** Anweisung stehen, da eine zweite Anweisung zu einem **Signaltreiberkonflikt** führen würde. Dieser Prozeß endet mit einer unbedingten **wait**–Anweisung.

Die Signale E(l) und E(2) hätten bei den gewählten Signalmustern auch durch je einen weiteren Taktgeneratorprozeß mit einer Periode von 100ns bzw. 200ns definiert werden können.

### 5.4.4 VHDL-Namenskonventionen und reservierte Bezeichner

The following **naming conventions** apply to VHDL designs:

- VHDL is not case sensitive.
- Two dashes -- are used to begin comment lines.
- Names can use alphanumeric characters and the underscore character.
- Names must begin with an alphabetic letter.
- You may not use two underscores in a row, or use an underscore a the last character in the name.
- Spaces are not allowed within names.
- Object names must be unique. For example, you cannot have a signal named A and a bus named A(7downto 0).

The following is a list of the VHDL reserved keywords:

| | | | | |
|---|---|---|---|---|
| abs | downto | library | postponed | subtype |
| access | else | linkage | procedure | then |
| after | elsif | literal | process | to |
| alias | end | loop | pure | transport |
| all | entity | map | range | type |
| and | exit | mod | record | unaffected |
| architecture | file | nand | register | units |
| array | for | new | reject | until |
| assert | function | next | rem | use |
| attribute | generate | nor | report | variable |
| begin | generic | not | return | wait |
| block | group | null | rol | when |
| body | guarded | of | ror | while |
| buffer | if | on | select | with |
| component | inertial | others | signal | |
| bus | impure | open | severity | xnor |
| case | in | or | shared | xor |
| configuration | inout | out | sla | |
| constant | is | package | sra | |
| disconnect | label | port | srl | |

Tabelle 5.8: Reservierte Schlüsselwörter in VHDL

### 5.4.5 Bewertung von VHDL

Der Einsatz einer standardisierten Hardwarebeschreibungssprache bietet viele Vorteile für den Entwurf komplexer elektronischer Schaltungen. Das soll kein uneingeschränktes Lob für VHDL sein, denn es gibt leider auch Nachteile. So kann die umfangreiche VHDL–Syntax einerseits zwar viele Modellierungsmöglichkeiten bieten, hat aber andererseits den Nachteil eines hohen Einarbeitungsaufwands.

#### 5.4.5.1 Vorteile von VHDL

##### Vielseitigkeit

Es ist zweifellos ein Vorteil, daß VHDL eine Sprache für viele Zwecke ist, d.h. sowohl eine spezifikations- und simulationsgeeignete Sprache als auch eine Ein- und Ausgabesprache für die Schaltungssynthese. Durch die firmenunabhängige Standardisierung ist ein Datenaustausch zwischen verschiedenen Programmen, zwischen verschiedenen Entwurfsebenen, zwischen verschiedenen Projektteams und zwischen Entwickler und Hersteller möglich.

##### Programmunabhängigkeit

Neben VHDL gibt es auch noch andere Hardwarebeschreibungssprachen und Datenformate. Man unterscheidet dabei zwischen programmspezifischen und nicht programmspezifischen Sprachen bzw. Formaten. Erstere sind an einen bestimmten Software-Hersteller gebunden und werden meist nicht von den Programmen anderer Hersteller unterstützt. Auch die heutige Hauptkonkurrenz von VHDL, **Verilog–HDL**, war lange Zeit herstellerspezifisch und hat sich erst vor kurzem zu einer unabhängigen HDL entwickelt. VHDL ist vom Ansatz her programmunabhängig. Es gibt eine Vielzahl von Software-Anbietern, die für viele Entwurfsschritte integrierter Schaltungen VHDL–Lösungen anbieten. Man kann somit unter mehreren Alternativen die zur Lösung der anstehenden Aufgabe optimale Software aus wählen. Bei VHDL stand von Anfang an die Unabhängigkeit von Rechnersystemen im Vordergrund. Falls dennoch systemabhängige Aspekte erforderliche sind, können sie in **Packages** gekapselt werden, so daß das VHDL–Modell selbst unabhängig vom eingesetzten Rechnersystem bleibt, d.h. portierbar ist.

##### Technologieunabhängigkeit

VHDL ist technologieunabhängig. Die Entscheidung für eine bestimmte Technologie (FPGA, Gate–Array, Standardzelle) muß erst relativ spät getroffen werden. Ein danach eventuell nötiges Umschwenken auf eine andere Technologie verursacht kein komplettes Redesign. Durch die Freiheit zur

- Definition von eigenen Logiktypen mit entsprechenden Operatoren
- technologiespezifischen Signalverknüpfungen (wired–or, wired–and)
- neuen, bei strukturalen Modellen eingesetzten Komponentenbibliotheken

können die spezifischen Eigenschaften aller Technologien mit VHDL abgebildet werden.

##### Modellierungsmöglichkeiten

VHDL stellt zahlreiche Konstrukte zur Beschreibung von Schaltungen und Systemen zur Verfügung. Damit lassen sich Modelle z.B. auf algorithmischer, Register-Transfer und Logikebene erstellen. Die VHDL–Modelle können dabei Unterkomponenten verschiedener Entwurfssichten und -ebenen enthalten. Bei der strukturalen Modellierung kann eine mehrstufige Hierarchie implementiert werden. Beschreibungen auf abstraktem Niveau haben mehrere Vorteile: Sie sind kompakt und

überschaubar, der Überblick über den gesamten Entwurf bleibt erhalten. Sie ermöglichen kürzere Entwicklungszeiten, benötigen weniger Rechenzeit bei der Simulation und erlauben eine frühzeitige Verifikation.

### Entwurf komplexer Schaltungen

VHDL hat die Verbreitung von Synthesewerkzeugen verstärkt und so eine neue und produktivere Entwurfsmethodik mit strukturierter Top−Down−Vorgehensweise herbeigeführt. Wesentliche Aspekte dabei sind:

- frühzeitige Entwurfsüberprüfung, da die Spezifikation eine simulierbare Beschreibung darstellt
- Verhaltensbeschreibungen können synthetisiert werden
- zahlreiche Sprachkonstrukte zur Parametrisierung von Modellen erlauben ein einfaches Design von Varianten
- die Entwicklung wiederverwendbarer Modelle wird unterstützt
- rasche Umsetzungsmöglichkeiten auf verschiedene Technologien sind gegeben

### Selbstdokumentation

VHDL ist direkt vom Menschen lesbar. Die Syntax ist sehr ausführlich und hat selbsterklärende Befehle. Wählt man geeignete Objektnamen, so enthält eine Beschreibung genügend Informationen, um zu einem späteren Zeitpunkt problemlos ohne zusätzliche Kommentare interpretiert werden zu können.

#### 5.4.5.2 Nachteile von VHDL

Mit der Verwendung von VHDL beim Entwurf integrierter Schaltungen ist weit mehr verbunden als mit der Nutzung irgendeiner neuen Programmiersprache oder eines neuen Formates. Der gesamte Entwurfsablauf hat sich von der früheren manuellen Vorgehensweise mit Schaltplaneingabe auf Logikebene zur Schaltungsbeschreibung auf RT-Ebene mit anschließender Synthese gewandelt. Dies hat mehrere Folgen:

- Es ist ein grundsätzlich neuer Entwurfsstil nötig, der mit einem ziemlich radikalen Umdenken bei den Hardware-Entwicklern verbunden ist. Erfahrungsgemäß haben gerade Hardwareentwickler, die in ihrer Ausbildung nicht mit modernen Programmiersprachen und der erforderlichen strukturierten Vorgehensweise vertraut gemacht wurden, dabei große Probleme.
- Die erforderlichen Aus- und Weiterbildungsmaßnahmen verursachen Kosten und Ausfallzeiten. Hinzu kommt meist die Neuanschaffung von Rechnern mit genügend Leistung sowie Software-Lizenzen, deren Kosten die der Rechner in der Regel weit übersteigen.

### Modellierung analoger Systeme

VHDL wurde zunächst mit umfangreichen Beschreibungsmitteln für Digitaltechnik ausgestattet. Im Laufe der Zeit wurden auch Konstrukte zur Modellierung analoger Systeme entwickelt, die auch Komponenten mit mechanischen, optischen, thermischen, akustischen oder hydraulischen Eigenschaften und Funktionen einschließen. Damit ist VHDL auf dem Weg, eine vollständige Verhaltensmodellierung technischer Systeme zu ermöglichen. Die Definition einer erweiterten Norm IEEE 1076.1, AHDL, soll speziell die Modellierung analoger elektronische Schaltungen mit wert- und

zeitkontinuierlichen Signalen gestatten. Von der allgemeinen Verwendbarkeit zur Schaltungssynthese – wie bei Digitalschaltungen – ist man jedoch leider noch weit entfernt.

### Komplexität

Obwohl die Komplexität von VHDL aufgrund der vielen Modellierungsmöglichkeiten generell sicher ein Vorteil ist, kann sie sich im Einzelnen aber auch nachteilig auswirken:

- VHDL erfordert einen hohen Einarbeitungsaufwand. Es ist mit einer weitaus längeren Einarbeitungszeit als bei jeder anderen Programmiersprache zu rechnen. Insbesondere muß ein geeigneter Modellierungsstil **eingeübt** werden.
- Das Verhalten eines komplexen VHDL−Modells in der Simulation ist für den Neuling kaum nachvollziehbar, da die zugehörigen Mechanismen nicht von gängigen Programmiersprachen abgeleitet werden können.
- Die Semantik wurde ursprünglich an vielen Stellen nicht eindeutig und klar festgelegt. In der Überarbeitung (1993) wurden einige Unklarheiten beseitigt. Trotzdem bleibt das Nachschlagewerk für VHDL, das Language Reference Manual (LRM) eines der am schwersten zu lesenden Bücher der Welt.

### Synthese−Subsets

VHDL ist als Sprache bezüglich der Syntax und Simulationssemantik standardisiert, nicht jedoch für die Anwendung als Eingabe für Synthesewerkzeuge. Außerdem enthält VHDL Konstrukte, die sich **prinzipiell nicht** in eine Hardware umsetzen lassen. Darüber hinaus unterstützt jedes Synthesewerkzeug einen etwas anderen VHDL−Sprachumfang (**Subset**) und erfordert einen spezifischen, angepaßten Modellierungsstil. VHDL−Modelle müssen somit meist auf ein spezielles Synthesewerkzeug zugeschnitten sein. Dies verhindert einen unkomplizierten Wechsel des Werkzeugs und erhöht die Abhängigkeit vom Werkzeughersteller. Der Entwickler muß die Anforderungen des gewählten Werkzeuges kennen und von Anfang an bei der Modellerstellung berücksichtigen.

### Ausführlichkeit

Die Ausführlichkeit von VHDL kann auch als Nachteil empfunden werden. Der manchmal als „geschwätzig“ bezeichnete Stil verursacht lange und umständliche Beschreibungen. Bei der Modellerstellung per Eingabe in einem Texteditor verhindert der Umfang des einzugebenden Textes oft ein schnelles Vorgehen.

## 6 Literatur


1. Peter Urbanek: Mikrocomputertechnik. B.G. Teubner, Stuttgart, 1999. ISNB: 3-519-06262-3
2. Uwe Brinkschulte, Theo Ungerer: Mikrocontroller und Mikroprozessoren. Springer, 2002, ISBN: 3-540-43095-4
3. Klaus Wüst: "Mikroprozessortechnik", Vieweg-Verlag, 2003
4. P. Lapsley, J. Bier, a: Shoham, E.A. Lee, DSP Processor Fundamentals. IEEE Press, 1995. ISBN: 0-7803-3405-1
5. John F. Wakerly: Digital Design. Prentice Hall, 2001. ISBN: 0-13-089896-1
6. J. Reichardt/B. Schwarz: "VHDL-Synthese", Oldenbourg-Verlag, 2000. ISBN: 3-486-25128-7

# 7 Anhang: Digitale Schaltsymbole