

# 5 Rechnerarchitektur

Computer bestehen aus einer Zentraleinheit (engl. *Central Processing Unit*, kurz *CPU*), einem Arbeitsspeicher (engl. *Random Access Memory*, kurz *RAM*) und Peripheriegeräten. Alle diese Teile sind hochkomplexe Schaltkreise, die hauptsächlich aus *Transistoren* aufgebaut sind. Transistoren werden hier als elektrisch gesteuerte Ein-Aus-Schalter eingesetzt. Durch geschickte Kombination vieler solcher Schalter entstehen Schaltkreise, die jedes gewünschte Verhalten realisieren können. Die *boolesche Algebra*, die wir in der ersten Hälfte dieses Kapitels kennen lernen, erlaubt es uns, zu einer beliebigen Schaltaufgabe einen entsprechenden Schaltkreis auszurechnen. Damit ausgerüstet zeigen wir, wie die wichtigsten Bauelemente eines Rechners, nämlich ALU (engl. *Arithmetic Logic Unit*, kurz *ALU*) und Speicher, aus einfacheren Schaltkreisen aufgebaut werden können. Aus diesen konstruieren wir danach eine mikroprogrammierte CPU und vollziehen damit den Übergang von der Hard- zur Software. Wir verfolgen diesen bis zum Maschinencode und Assembler und diskutieren anschließend noch RISC (engl. *Reduced Instruction Set Computer*) als alternative CPU-Architektur.

Dieses Kapitel erläutert also prinzipiell, wie durch geschickte Kombination von Transistoren ein komplexes Gerät wie ein PC entsteht. Wenn man wollte, könnte man Transistoren auch durch optische Schalter ersetzen und mit den gleichen Prinzipien einen optischen Computer konstruieren. Durch die schnelleren Umschaltzeiten optischer Bauteile darf man sich einen erheblichen Geschwindigkeitsgewinn erhoffen. Allerdings sind optische Schalter heute noch nicht so einfach zu realisieren wie Transistoren. Insbesondere ist eine technische Lösung für die Zusammenfassung (Integration) von Tausenden oder gar Millionen optischer Bauelemente auf einem Chip noch in weiter Ferne. Heute ist die CMOS-Technik in der Realisierung von Transistorschaltungen führend. Wir werden lernen, wie sich in dieser Technik besonders leistungsfähige Bauelemente entwerfen und realisieren lassen. Auch einige Aspekte der Herstellung elektronischer Chips wollen wir in diesem Kapitel beleuchten.

## 5.1 Vom Transistor zum Chip

Das für uns wichtigste elektronische Bauelement ist der so genannte *MOS-Transistor*. MOS ist die Abkürzung für den englischen Begriff *metal oxide semiconductor* (Metalloxid-Halbleiter). Es gibt verschiedene Arten von MOS-Transistoren, alle sind, wie auch in der folgenden Abbildung zu sehen, aus mehreren Materialschichten aufgebaut. Ausgangspunkt ist kristallines Silizium, das durch Einbringung von Fremdatomen *dotiert* (verunreinigt) ist. Man unterscheidet zwischen n-dotiertem und p-dotiertem Silizium. Im ersten Fall entsteht durch die

Fremdatome ein Elektronenüberschuss und damit freie negative Ladungsträger, im Falle von p-dotiertem Silizium ein Mangel an Elektronen, was man als freie positive Ladungsträger interpretieren kann.

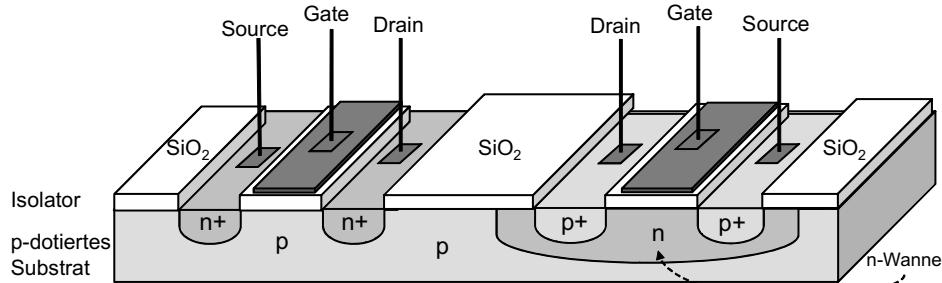


Abb. 5.1: n-MOS-Transistor und p-MOS Transistor auf gemeinsamem p-Substrat

Zwischen den p- und n-dotierten Bereichen bilden sich Grenzschichten aus, in denen der Elektronenüberschuss der n-Schicht den Elektronenmangel der angrenzenden p-Schicht ausgleicht. Dadurch entsteht eine neutrale Zone, in der keine freien Ladungsträger vorhanden sind, so dass diese als Isolator wirkt. Eine zwischen den mit *Source* und *Drain* bezeichneten stark dotierten Bereichen (n<sup>+</sup>, bzw. p<sup>+</sup>) angelegte Spannung bewirkt daher keinen Stromfluss.

Hier kommt der in der obigen Figur als *Gate* bezeichnete metallische Kontakt ins Spiel, der durch einen Isolator (SiO) von der schwach p-dotierten Schicht (p) getrennt ist. Legt man eine positive Spannung zwischen Gate und Source an, so lädt sich das Gate positiv auf. Wird dabei ein gewisser Schwellwert überschritten, so wird in dem p-dotierten Bereich unter dem Gate ein elektrisch leitender Kanal aus negativen Ladungsträgern induziert. Eine ausreichende Spannung am Gate schaltet somit eine elektrische Verbindung zwischen Source und Drain; fällt diese Spannung unter einen Schwellwert, so wird die Verbindung wieder unterbrochen.

Wenn man in der obigen Erklärung die n-dotierten und p-dotierten Bereiche austauscht, erhält man einen p-MOS Transistor. Eine negative Spannung am Gate induziert im n-Substrat einen Kanal positiver Ladungsträger. Da sowohl n-MOS als auch p-MOS Transistoren auf dem gleichen Substrat aufgebracht werden müssen, fertigt man zunächst eine in das p-Substrat eingeschlossene n-Wanne, in der man dann den p-MOS Transistor aufbaut. Im Schaltbild wird der p-MOS Transistor durch einen kleinen Kreis am Gate kenntlich gemacht. Den Grund dafür werden wir in Abschnitt 5.4.1 erfahren.



Abb. 5.2: Schaltbilder für n-MOS und p-MOS Transistoren

Für uns ist einstweilen nur diese Schalterwirkung der Transistoren von Interesse. Dies soll auch in den symbolischen Schaltbildern zum Ausdruck kommen. Steigt die Spannung zwischen Gate und Source über einen Schwellwert, dann schaltet Source zu Drain durch. Ein Abfallen der Spannung unterbricht diese Verbindung.

### 5.1.1 Chips

Ein *Chip* ist ein dünnes Silizium-Scheibchen, auf das die Transistorschaltung beim Herstellungsprozess aufgebracht wird. Da auf einer daumennagelgroßen Fläche eine sehr große Anzahl von Schaltgliedern zu einem Schaltkreis zusammengefasst werden, nennt man das entstandene Bauteil auch *Integrated Circuit (IC)*. Mit den Jahren wuchs die Anzahl der Baulemente auf einem einzigen Chip um mehrere Größenordnungen, entsprechend wandelte sich auch der Name über *LSI (large scale IC)* zu *VLSI (very large scale IC)*. Heutige CPU-Chips enthalten einige 100 000 000 Transistoren auf einer Fläche von weniger als 100 mm<sup>2</sup>. Speicherchips können aufgrund ihrer regelmäßigeren Struktur noch höher integriert werden.

Die Dicke eines Chips beträgt nur etwa 1/10 mm, die der *aktiven Schicht* ist noch erheblich geringer. In der aktiven Schicht finden sich die Transistoren, Dioden, Widerstände und die Leitungen. Der Chip ist in ein Gehäuse aus Kunststoff oder Keramik eingebettet, das erheblich größer ist als das Silizium-Scheibchen. Die Verbindungen von dem inneren Silizium-Scheibchen zu den Außenkontakten des Chip-Gehäuses werden mithilfe hauchdünner Golddrähtchen hergestellt. Klassische Chips sind in einem rechteckigen Gehäuse mit zwei Reihen seitlich angebrachter Anschlussdrähte, den *Beinchen* (engl. *pin*), untergebracht. Die Anzahl der Außenverbindungen ist bei solchen Chips auf etwa 64 beschränkt. Chips mit mehr Anschlüssen (bis zu etwa 100) setzt man oft in ein quadratisches Gehäuse mit Anschlussdrähten an allen vier Seiten. Noch mehr Außenverbindungen schafft man durch Anbringung der Beinchen unter dem Chip. Durch diese *Pin Grid Array (PGA)* genannte Technik lassen sich Chips bauen, die mehrere hundert Verbindungen aufweisen können. Diese Technik wurde weiterentwickelt; heute üblich sind *Land Grid Arrays (LGA)*. Die Anschlüsse sind auf einem *Sockel* angeordnet. Dieser hat federnde Kontaktstifte, das Prozessorgehäuse nur mehr Kontaktflächen, sogenannte Lands. Der erste Intel Pentium Prozessor hatte ein PGA mit 273 Pins, die ersten Versionen des Pentium-4 kamen auf 423 Pins. Der neueste Prozessor aus der Intel 8086 Serie, der Core i7 960, hat ein LGA mit 1366 Pins.

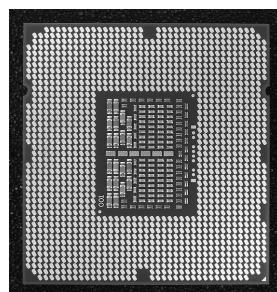


Abb. 5.3: Ansicht eines LGA. Beispiel Unterseite des Intel i7 975 (Bild: Rainer Knaepper)

Eine Leiterplatte ist in der Regel mit Chips unterschiedlicher Bauart bestückt und enthält zusätzlich einzelne klassische Bauelemente wie Kondensatoren oder Widerstände. Die Verdrahtung erfolgt meist in mehreren, mindestens jedoch zwei Verdrahtungsebenen. Diese stehen auf der Leiterplatte zur Verfügung, sind untereinander isoliert und haben Querverbindungen zu den anderen Ebenen. Werden mehrere Leiterplatten benötigt, sind diese meist senkrecht in eine Systemplatine (engl. *motherboard*) eingesteckt, die die Verbindungen enthält. Mit Anschlussbuchsen für genormte, mehrpolige Stecker kann ein Anschluss zu Netzteilen, externen Geräten etc. erfolgen.

In heutigen Computern finden sich meist eine oder mehrere Leiterplatten mit weniger als 50 Chips. In unmittelbarer Zukunft wird man durch höhere Integration die Anzahl der Chips in einem Rechner auf weniger als zehn reduzieren und zur selben Zeit die Leistung der Geräte um mehrere Größenordnungen steigern können.

### 5.1.2 Chipherstellung

Für die Herstellung eines Chips wird zunächst gereinigtes Silizium (Quarzsand) auf über tausend Grad erhitzt, bis es flüssig wird. Aus dieser Schmelze werden so genannte Einkristalle gezogen, die bis zu 2 m lang sein können und einen Durchmesser von etwa 20 bis 30 cm haben. Sie werden nach dem Erkalten in dünne Scheiben gesägt und poliert. Diese Scheiben sind das Ausgangsmaterial für den Herstellungsprozess, im Laufe dessen auf jeder einzelnen hunderte von Chips in einem Arbeitsgang entstehen.

Komplexe Chips erfordern mehrere hundert Herstellungsschritte. Sie können viele Millionen individueller Transistoren enthalten. Für jeden Schritt kommt, in jeweils abgewandelter Form, ein fotolithografisches Grundverfahren zur Anwendung. Dabei wird jedesmal zunächst eine Materialschicht aufgetragen und mit Fotolack überzogen. Dieser wird mithilfe einer Maske, auf der die Chipstrukturen ausgespart sind, belichtet. Nach der Entwicklung werden die unbefeuerten Stellen bearbeitet, das heißt entweder weggeätzt, dotiert oder mit Kontakten versehen. Dann wird der restliche Fotolack entfernt.

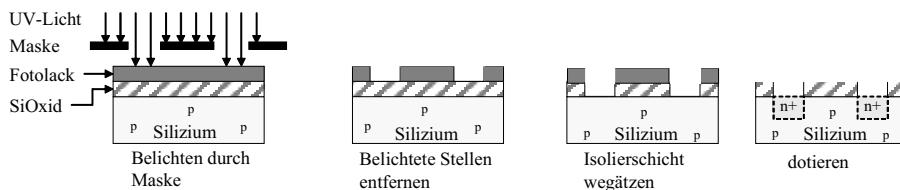
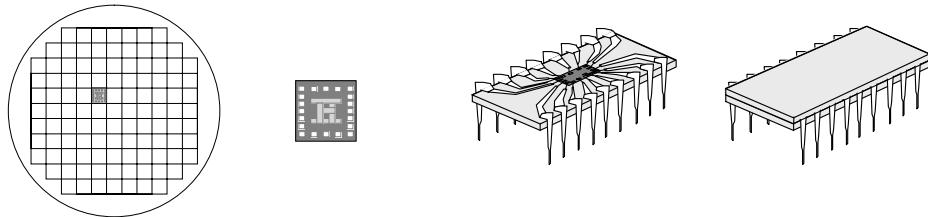


Abb. 5.4: Photolithographische Chipbearbeitung

Nach dem Aufbringen der Transistoren und Leiterbahnen entsteht auf den rechteckigen Siliziumscheiben, in einem durch die verwendeten Masken definierten Gebiet, ein waffelartiges Muster einzelner Chips. Daher werden die Siliziumscheiben auch *Wafer* genannt. Sie werden zersägt, in die Gehäuse eingebaut und mit den Anschlussdrähten verbunden (engl.: *bonding*). Das Gehäuse wird endgültig verschlossen – und fertig ist der Chip.

Gegenwärtig ist Silizium der Rohstoff der Wahl für die Fertigung von Chips. Es ist billig und einfacher zu bearbeiten, als der alternative Rohstoff Galliumarsenid, aus dem man Chips mit erheblich kürzeren Schaltzeiten fertigen kann, die in Supercomputern und anderen kritischen Anwendungen gelegentlich eingesetzt werden.



*Abb. 5.5: Wafer mit Chips, ausgesägter Chip, bonding und gekapselter Chip*

### 5.1.3 Kleinstre Strukturen

Ein wesentlicher Parameter bei der Chip-Herstellung ist die Größe der kleinsten Strukturen. Dabei handelt es sich um Leitungen, den Abstand zwischen zwei Leitungen oder um die Größe von Transistorzellen. Die kleinsten erzeugbaren Strukturen lagen lange Zeit im Bereich von 100 nm bis 1 $\mu$  ( $1\mu = 1$  Mikrometer =  $10^{-6}$  m, 1nm = 1 Nanometer =  $10^{-9}$ m). Zurzeit werden Strukturen von 32 nm bis 100 nm verwendet. Ein weiteres Absenken der kleinsten Strukturen auf 22 nm, 16 nm usw. ist für die nächsten Jahre geplant. Die Schwierigkeiten beim Verkleinern der Chip-Strukturen bestehen im Herstellen geeigneter Masken für die verschiedenen fotolithografischen Prozesse, in der exakten Positionierung der Masken, in Belichtungsproblemen, wenn die Wellenlänge des für die Belichtung verwendeten Lichts erreicht wird, und in mikroskopischen Ungenauigkeiten beim Ätzen, Beschichten etc.

Diese Schwierigkeiten konnten bisher immer wieder bewältigt werden. Meist waren dafür jedoch langwierige Forschungs- und Entwicklungsarbeiten erforderlich, so dass die kleinsten beherrschbaren Strukturen nur relativ langsam von 2  $\mu$  auf 1  $\mu$  und dann schrittweise auf 32 nm verkleinert werden konnten. Die Verkleinerung auf Werte in der Größenordnung von 10 bis 30 nm wird weitere technische Innovationen erfordern. Als konsequente Fortsetzung der optischen Lithografie hin zu kürzeren Wellenlängen gilt z.B. die EUV-Lithografie (*Extreme Ultra Violet*). Dabei werden Wellenlängen im Bereich 13,5 nm genutzt, um Strukturen zwischen 45 nm und 32 nm und kleiner zu erzeugen.

### 5.1.4 Chipfläche und Anzahl der Transistoren

Die Herstellung von Chips ist ein langwieriger und fehleranfälliger Prozess. Der Anteil von funktionsfähigen Chips beträgt daher nur etwa 5 bis 50%, bezogen auf die Gesamtproduktion, je nach der bereits gewonnenen Produktionserfahrung mit einem bestimmten Herstellungsprozess. Die Fehlerrate bei den einzelnen Chips ist von der Fläche des produzierten Chips abhängig. Um die Produktion wirtschaftlich zu machen, versucht man, die Chipfläche auf ein vertretbares Minimum zu reduzieren. Nur wenn es nicht anders geht, erhöht man die Chipflä-

che, um die Anzahl der Transistoren zu erhöhen. Gegenwärtig ändert sich die effektiv ausgenutzte Chipfläche von ca. 100 bis 250 mm<sup>2</sup> nur wenig, da die Herstellungsprozesse so häufig verbessert werden, dass eine Vergrößerung der Chipfläche kaum notwendig ist.

Der Prozessor des Core i7 980X wird seit März 2010 gefertigt und verfügt über 1,17 Milliarden Transistorfunktionen auf einer Fläche von 248 mm<sup>2</sup>, gefertigt wird er mit einem 32 nm Prozess. Das ältere Modell Core 2 Duo E8600 wird mit einem 45 nm Prozess hergestellt und besitzt über 410 Millionen Transistorfunktionen auf einer Fläche von 107 mm<sup>2</sup>. Beide Prozessoren unterscheiden sich in der Anzahl der CPU-Kerne und der Cache-Größe. Das neuere Modell hat 6 Prozessorkerne und einen Cache von 12 MB, das ältere hat nur 2 Prozessorkerne und 6 MB Cache. In beiden Modellen benötigen die Prozessorkerne vermutlich jeweils etwa 20 bis 30 Millionen Transistorfunktionen. Bei zukünftigen Generationen wird die Anzahl der Transistorfunktionen vermutlich weiter steigen – und für eine größere Zahl von Prozessorkernen bzw. für noch mehr Cache-Speicher genutzt werden.

Bei Speicherchips mit ca. 100 mm<sup>2</sup> effektiver Nutzfläche wird in Anwendung einer 32 nm Technik gegenwärtig eine Zahl von ca. 10 000 000 000 Transistoren erreicht. Über weitere Steigerungsmöglichkeiten kann man derzeit nur spekulieren.

### 5.1.5 Weitere Chip-Parameter

Je geringer die kleinsten Strukturen auf einem Chip sind, desto geringer sind die Schaltverzögerungen pro Transistor und der Energieverbrauch pro Schaltvorgang. Wenn dieser Energieverbrauch, der gegenwärtig ca. 1 pJ (Picojoule) beträgt, nicht um eine ganze Größenordnung gesenkt werden könnte, wäre eine Erhöhung der Transistorzahl gar nicht möglich – die Chips würden zu heiß werden.

Die Schaltverzögerung von modernen MOS-Transistoren beträgt weniger als 0,1 ns (NanoSekunden). Die Schnelligkeit einer ganzen Leiterplatte wird nicht nur durch die Geschwindigkeit der Transistoren in den Chips bestimmt, sondern auch durch die Zahl und die Länge der Verbindungen der verschiedenen Chips untereinander. Je mehr Transistoren in einem Chip untergebracht werden können, desto weniger Inter-Chip-Verbindungen sind erforderlich – um so schneller ist die Leiterplatte.

### 5.1.6 Speicherbausteine

Auch der Speicher eines Rechners ist aus Chips aufgebaut, den so genannten RAM-Chips. *RAM* ist die Abkürzung für den englischen Begriff *Random Access Memory* – zu deutsch: Speicher mit wahlfreiem Zugriff. Verwendet man die Ladung auf dem Gate eines Transistors zur Speicherung eines Bit, kommt man, zusammen mit der Adressierlogik, auf Speicherbausteine mit weniger als 1,5 Transistoren pro Bit. Allerdings verlieren diese *dynamischen* Speicherbausteine (*DRAM*) nach kurzer Zeit ihre Ladung wieder. Jedes Bit muss innerhalb einer bestimmten Zeit, die im Nanosekundenbereich liegt, wieder aufgefrischt, also gelesen und neu geschrieben werden. Eine Alternative ist die Verwendung *statischer* Speicherbausteine (*SRAM*). Diese müssen zwar nicht ständig aufgefrischt werden, benötigen aber mehrere Transistoren pro Bit. Sowohl dynamische als auch statische RAM-Chips verlieren die gespeicherte Information, wenn kein Strom vorhanden ist. Dies kann durch bestimmte, aufwändige Schaltungen oder durch Verwendung von Akku-Puffern verhindert werden. Heute werden dynamische RAM-Chips mit 512

MBit, 1, 2 und 4 GBit Speicherkapazität gefertigt – in absehbarer Zeit wird es voraussichtlich auch 8 und 16 GBit RAM-Chips geben. Die Entwicklungsgeschichte der Speicherbausteine illustriert Abbildung 5.6. Bei Speicherbausteinen spielt der Preis eine wesentliche Rolle. Daher werden derzeit hauptsächlich RAM-Chips mit 1 und 2 GBit Speicherkapazität zu günstigen Preisen angeboten. 4 GBit Speicherbausteine sind noch vergleichsweise teuer.

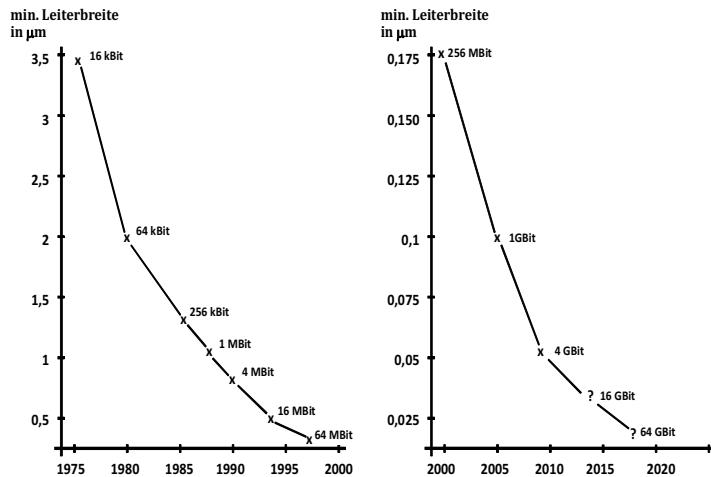


Abb. 5.6: Entwicklung von Speicherchips

Heutzutage werden Speicherbausteine vom Typ DDR-SDRAM (Double Data Rate Synchronous Dynamic Random Access Memory) verwendet. Gegenwärtig ist die dritte Generation DDR3-SDRAM aktuell, eine vierte Generation soll 2012 auf den Markt kommen. Die leistungsfähigsten Chips vom Typ DDR3-1600 haben einen Speichertakt von 200MHz, einen „effektiven“ Takt von 1600 MHz und erreichen eine Datentransferrate von 12,8 GByte/Sekunde.

Speicherbausteine kauft man nicht einzeln, sondern als Speichermodule, auf denen mehrere RAM-Chips und weitere Logikbausteine untergebracht sind. Die Speichermodule vom Typ DIMM (Dual Inline Memory Module) sind zum Einbau in normale PCs gedacht, die vom Typ SO-DIMM (Small Outline - DIMM) zum Einbau in Notebookcomputer. Handelsüblich sind Module mit einer Kapazität von 1, 2 oder 4 GByte. Module mit den oben erwähnten DDR3-1600 Chips werden auch als PC3-12800 bezeichnet.

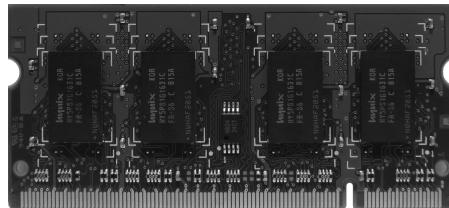


Abb. 5.7: Ein 1 GByte SO-DIMM mit DDR2 SDRAM (Quelle: Matthieu Riegler, Wikimedia Commons)

In den letzten Jahren sind Flash-Speicher sehr populär geworden. Diese verwenden Speicherzellen, die die gespeicherte Information nicht verlieren, wenn kein Strom vorhanden ist. Sie beruhen auf dem *EEPROM* Prinzip (*Electrically Erasable Programmable Read-Only Memory*). Derartige Speicherbausteine können genauso einfach gelesen werden, wie normale Speicherzellen. Das Schreiben erfordert allerdings einen speziellen Mechanismus, das *Umprogrammieren* der Speicherzellen durch Anlegen einer relativ hohen Spannung. Das Schreiben erfolgt meist blockweise und ist sehr viel langsamer als das Schreiben in normale Speicherzellen. Flash-Speicher benutzen Bausteine mit spezieller EEPROM Technologie und einer Blockgröße für Schreiboperationen von typischerweise 64 Bit. Derzeit sind preiswerte Flash-Speicher mit einer Kapazität von 1 GB bis 32 GB im Angebot. Bei höherer Kapazität ist der Preis derzeit noch überproportional höher. Derartige Speicherbausteine werden in MP3-Playern verwendet, in Speicherkarten für Kameras und in den mittlerweile ubiquitären USB-Sticks. Außerdem werden Flash-Speicher in Halbleiterlaufwerken, sogenannten Solid State Drives (SSD) verbaut, die zwar erheblich teurer sind als herkömmliche Festplattenlaufwerke, dafür aber sehr robust sind, keine Geräusche erzeugen, wenig Energie verbrauchen und wesentlich schneller sind.

### 5.1.7 Logikbausteine

Speicherbausteine bestehen aus vielen gleichartigen Speicherzellen und aus einer Lese- und Schreiblogik. Entsprechende Schaltpläne können in jeder Größe relativ rasch angefertigt werden. Daher verwundert es nicht, dass mit jedem neuen Herstellungsprozess, der eine bestimmte Maximalzahl von Transistoren ermöglicht, als erstes Speicherbausteine gebaut werden können. Anders sieht es bei den *Logikbausteinen* aus. Hierbei handelt es sich um Mikroprozessoren oder um sonstige Spezialschaltungen. Die entsprechenden Schaltpläne sind nicht so einfach herzustellen wie die der Speicherbausteine. Sie bestehen nicht aus immer wieder kopierten Speicherzellen, sondern im Extremfall aus lauter unterschiedlichen Funktionsgruppen. Im Fall der ersten Mikroprozessoren, die nur über wenige Tausend Transistorfunktionen verfügten, konnten die entsprechenden Schaltpläne noch am Reißbrett entworfen werden. Spätere Mikroprozessoren, wie der Intel 8086 mit ca. 30 000 und der Motorola 68000 mit 68 000 Transistorfunktionen, konnten mit den damaligen Werkzeugen nur entwickelt werden, weil Teile des Mikroprozessors wiederum als Speicher ausgelegt waren – als *Mikrogrammspeicher*. Der Übergang zu höher integrierten Schaltungen, wie z.B. der des 80286 mit 150 000 Transistoren, war nur mit computerunterstützten Methoden möglich. Heute stehen

ausgereifte Werkzeuge zum Entwurf hochintegrierter Logikbausteine zur Verfügung, mit denen Mikroprozessoren, wie z.B. der Pentium-4-Prozessor mit seinen ca. 42 Millionen Transistorfunktionen und der neuere Core i7 980X Prozessor mit seinen 1,17 Milliarden Transistorfunktionen, entwickelt werden können.

### 5.1.8 Schaltungsentwurf

Die wesentlichen Hilfsmittel für den Entwurf hochintegrierter Schaltungen sind CAD-Systeme und Simulationsprogramme. CAD steht für *Computer Aided Design* – zu deutsch etwa: *Rechnergestütztes Entwurfsverfahren*. Von Hand gezeichnete Schaltpläne für einige 100 000 Transistoren würden so groß wie Fußballfelder sein. Solche Schaltungen können daher nur noch mit elektronischen Entwurfssystemen beherrscht werden. Ähnliches gilt auch für das Testen der Schaltung. Früher wurde ein Prototyp hergestellt und dieser anschließend getestet. Die Herstellung von Chip-Prototypen ist jedoch sehr aufwändig. Es kann Monate dauern, bis ein Prototyp fertig ist, nur um dann nach Stunden wegen eines Fehlers verworfen zu werden.

Bei einem Chip mit einigen 1000 Transistoren kann die Schaltung im Elektroniklabor mithilfe von Oszilloscopen getestet werden. Wenn einige 100 000 oder sogar einige 100 000 000 Transistoren auf Funktionstüchtigkeit und hinsichtlich ihres Zusammenspiels getestet werden müssen, ist ein solches Verfahren nicht mehr möglich. Aus diesen Gründen verlagert man die Produktion der Prototypen von der Hardware in die Software. Statt eines physischen Probanden wird ein abstraktes Modell des zukünftigen Chips definiert und dessen Verhalten auf einem Rechner simuliert.

Dennoch verbleibt das Risiko von unentdeckten Entwurfsfehlern. Wie das Beispiel des „Pentium-FDIV-Bug“ zeigt, wurde ein Fehler in dem Pentium-Prozessor weder bei den Simulationen vor Produktionsbeginn noch beim Testen der ersten Serien von Prozessoren entdeckt. Erst anderthalb Jahre nach Beginn der Serienproduktion kam dieser Fehler mehr oder weniger zufällig ans Licht.

Der Entwurf hochintegrierter Transistorschaltungen ist ein ähnlich anspruchsvolles Problem wie der Entwurf und die Programmierung eines Softwaresystems, das aus mehreren hochkomplexen Programmen besteht. Es verwundert daher nicht, dass in beiden Fällen ähnliche Techniken angewendet werden.

*Der modulare Entwurf:* Ein komplexes System wird in mehrere einfachere Module mit klaren Schnittstellen zerlegt. Ein hochintegrierter Chip besteht häufig aus einzelnen Modulen, die über Schaltungskanäle verdrahtet sind.

*Standardschaltungen:* Für bestimmte wiederkehrende Aufgaben werden immer die gleichen Schaltungen verwendet, die in *Zellbibliotheken* verwaltet werden. Diese Zellbibliotheken bestehen aus logischen Standardschaltungen und deren Implementierung, jeweils in einem bestimmten Herstellungsprozess.

Nach wie vor ist die Struktur vieler Mikroprozessoren so, dass möglichst viele Funktionen in *Mikroprogramme* verlegt werden, die wiederum in Speicherzellen abgelegt sind. So haben heutige Mikroprozessoren einen Anteil von 20 bis 50% an Transistoren mit Speicherfunktion.

Ein alternativer Weg zur Vereinfachung von Mikroprozessoren wird von den noch zu diskutierenden *RISC-Prozessoren* eingeschlagen. Bei RISC-Prozessoren wird der Befehlssatz so weit vereinfacht, dass man mit sehr wenigen Mikroprogrammen auskommt. Durch die konsequente Verwendung von *regulären Strukturen* erreicht man eine Vereinfachung des Schaltungsentwurfes und damit eine schnellere Anwendung eines moderneren Herstellungsprozesses auch für Logikschaltungen.

## 5.2 Boolesche Algebra

Die Prinzipien heutiger Computer lassen sich weitgehend auf der Basis abstrakter Ein/Aus-Schalter verstehen, egal in welcher Technologie diese realisiert werden. Das mathematische Werkzeug um aus einfachen Ein/Aus-Schaltern hochkomplexe Schaltkreise zu konstruieren liefert die *boolesche Algebra*, die wir in diesem Kapitel kennenlernen werden. Sie entstand aus den Arbeiten des Engländer *George Boole* (1815–1864), dessen eigentliches Ziel es war, die Logik formal zu begründen. Es ging darum, die Wahrheit oder Falschheit von Aussagen zweifelsfrei feststellen zu können, ähnlich wie man auch das Ergebnis einer Addition oder Multiplikation ausrechnen kann. Die Objekte, mit denen Boole operierte, waren *Wahrheitswerte* (*wahr* und *falsch*) doch kann man sie genauso gut als Bitwerte (**0** und **1**) oder Stromzustände (Strom fließt/Strom fließt nicht) interpretieren.

### 5.2.1 Serien-parallele Schaltungen

Information wird in einem Rechner letztlich durch eine Folge von *Bits* realisiert. Jedes Bit kann zwei Zustände haben, die wir mit **1** und **0** bezeichnen. Technisch können diese Zustände durch Spannungen realisiert werden, z.B. **0** durch eine Spannung zwischen 0,0 und 0,4 V und **1** durch eine Spannung zwischen 2,4 und 5,0 V.

In einem einfachen Stromkreis, bestehend aus einer Batterie *B*, einem Schalter *S* und einem Lämpchen *L* können wir die Bitwerte **1** und **0** dadurch realisieren, dass das Lämpchen brennt oder erlischt. Das Verhalten des Kreises kann man in einer Tabelle darstellen. Wenn wir die Stellungen des Schalters, offen bzw. geschlossen, mit **0** bzw. **1** bezeichnen, erhalten wir die folgende Tabelle für den Zustand der Lampe *L*:

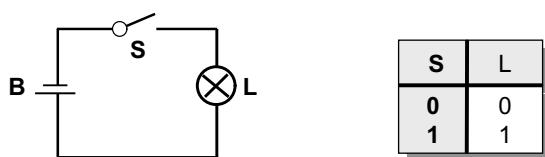


Abb. 5.8: Schaltungen

Ersetzen wir den Schalter *S* durch zwei Schalter, *S<sub>1</sub>* und *S<sub>2</sub>*, so ergeben sich zwei Kombinationsmöglichkeiten, die *Parallelschaltung* und die *Serienschaltung*:

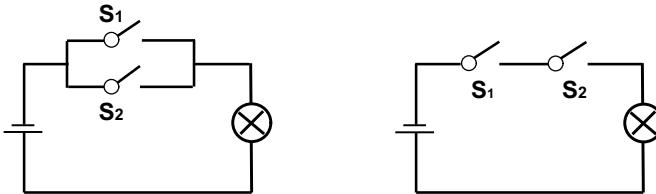


Abb. 5.9: Parallelschaltung und Serienschaltung

Die zugehörigen *Schalttabellen* beschreiben alle möglichen Stellungen von  $S_1$  und  $S_2$  zusammen mit dem Ergebnis, das wir in Spalte L angeben:

$S_1$	$S_2$	L
0	0	0
0	1	1
1	0	1
1	1	1

$S_1$	$S_2$	L
0	0	0
0	1	0
1	0	0
1	1	1

S

Abb. 5.10: Schalttabellen für Parallelschaltung und für Serienschaltung

## 5.2.2 Serien-parallele Schaltglieder

Wir untersuchen das Verhalten von Schaltkreisen, die aus einfacheren Schaltern zusammengebaut sind. Beginnend mit Elementarschaltern bauen wir neue Schaltglieder durch Serien- bzw. Parallelschaltung. Jedes Schaltglied hat einen Eingang und einen Ausgang. Sind  $S_1$  und  $S_2$  Schaltglieder, so erhält man durch Parallelschaltung das Schaltglied  $S_1 + S_2$  und durch Serienschaltung das Schaltglied  $S_1 \cdot S_2$ . Bei der Parallelschaltung genügt es, wenn einer der Schalter,  $S_1$  oder  $S_2$ , eingeschaltet ist, damit die gesamte Schaltung Strom durchlässt, bei der Serienschaltung ist es notwendig, dass  $S_1$  und  $S_2$  eingeschaltet sind. Daher wird die Parallelschaltung auch als *Oder-Schaltung*, die Serienschaltung als *Und-Schaltung* bezeichnet. Durch fortgesetzte Kombination von Schaltkreisen durch Serien- oder durch Parallelschaltung erhalten wir beliebig komplexe Schaltkreise, die *Serien-Parallel-Kreise*.



Abb. 5.11: Zwei Schaltungen mit gleichem Verhalten

Beginnend mit einfachen Ein-Ausschaltern, die wir mit  $x, y, z$  etc. bezeichnen, können wir schrittweise durch Parallel- und Serienschaltung komplexere Kreise zusammenbauen. Wenn zwei Schalter mit dem gleichen Namen vorkommen, wie die mit x bezeichneten Schalter in Abb. 5.11, so stellen wir uns vor, dass diese mechanisch oder elektrisch so gekoppelt sind,

dass sie immer in der gleichen Schaltposition (offen bzw. geschlossen) sein müssen. In der Darstellung verzichten wir auf die Batterie und das Lämpchen.

Verschieden aufgebaute Schaltungen können das gleiche Verhalten zeigen wie dies auch der Fall der beiden Schaltungen in Abb. 5.11 zeigt: Die Schaltungen sind jeweils geschlossen, wenn sowohl  $x$  als auch mindestens einer der Schalter  $y$  oder  $z$  geschlossen sind.

### 5.2.3 Schaltoperationen

Für die Parallelschaltung führen wir das Operationszeichen „+“ ein und für die Serienschaltung das Operationszeichen „•“. Sind also  $S_1$  und  $S_2$  Schaltkreise, so sei  $S_1 + S_2$  der Kreis, der aus  $S_1$  und  $S_2$  durch Parallelschaltung entsteht und  $S_1 \cdot S_2$  der Kreis, der aus  $S_1$  und  $S_2$  durch Serienschaltung entsteht. Die in Abb. 5.11 gezeigten Schaltungen kann man dann durch die Ausdrücke „ $x \cdot (y+z)$ “ und „ $(x \cdot y) + (x \cdot z)$ “ beschreiben.

„+“ und „•“ sind also Operationen auf Schaltkreisen. Dabei interessiert uns lediglich, ob bei einer bestimmten Stellung der Bestandteile das zusammengesetzte Schaltglied geöffnet oder geschlossen ist. Mit der Abkürzung Ein = 1, Aus = 0 erhalten wir die Operationstafeln für  $S_1 + S_2$  und  $S_1 \cdot S_2$ . Diese Tabellen kann man auch so beschreiben:

$S_1 + S_2$  hat Wert 1, wenn mindestens einer der Bestandteile,  $S_1$  oder  $S_2$ , den Wert 1 hat.

$S_1 \cdot S_2$  hat Wert 1, wenn beide Bestandteile,  $S_1$  und  $S_2$ , den Wert 1 haben.

Dieses Verhalten entspricht genau den Operationen | (*OR*) sowie & (*AND*) auf Werten vom Typ *boolean* in Programmiersprachen. Setzt man *true* für 1 ein und *false* für 0, so werden aus den obigen Operationstafeln genau die Verknüpfungstafeln für *AND* bzw. *OR*.

	false	true	&	false	true
false	false	true	false	false	false
true	true	true	true	false	true

Abb. 5.12: OR - und AND-Verknüpfung

### 5.2.4 Boolesche Terme

Bezeichnet man die elementaren Ein-Aus-Schalter mit Variablen  $x, y, z, \dots$ , so lässt sich jeder seri-parallele Schaltkreis durch einen seri-parallelen Term (kurz *SP-Term*) beschreiben. Diese sind induktiv folgendermaßen definiert:

- (i) 0 und 1 sind SP-Terme
- (ii) Jede Variable  $x, y, z, \dots$  ist ein SP-Term.
- (iii) Sind  $t_1$  und  $t_2$  SP-Terme, so auch  $t_1 + t_2$  und  $t_1 \cdot t_2$ .

0 bzw. 1 stehen in dieser Definition für Schalter, die immer offen bzw. immer geschlossen sind. Die Operationszeichen  $+$  und  $\bullet$  haben natürlich eine andere Bedeutung als gewohnt. Um eine Verwechslung auszuschließen, benutzt man statt ihrer auch die Zeichen  $\vee$  und  $\wedge$  oder schreibt sie aus als **OR** und **AND**. Zusätzlich erlauben wir Klammern, um die Entstehung eines Terms aufgrund der definierenden Regeln klarzustellen.

Die Schalttabelle, für einen SP-Term lässt sich schrittweise ermitteln. Für den Term  $x \bullet (y + z)$  baut man zunächst den inneren Teilterm  $(y + z)$  aus den Schaltern  $y$  und  $z$  zusammen und schaltet den erhaltenen Kreis in Serie mit dem Schalter  $x$ . Ähnlich erstellt man schrittweise die zugehörige Schalttabelle:

<b>x</b>	<b>y</b>	<b>z</b>	<b>y + z</b>	<b>x•(y+z)</b>
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	1	0
1	0	0	0	0
1	0	1	1	1
1	1	0	1	1
1	1	1	1	1

Abb. 5.13: Zum Term  $x \bullet (y + z)$  gehörende Schalttabelle

### 5.2.5 Schaltfunktionen

Eine *Schaltfunktion* ist eine  $n$ -stellige Operation auf der Menge  $\{0, 1\}$ , also eine Abbildung  $f: \{0,1\}^n \rightarrow \{0,1\}$ . Jeder SP-Term beschreibt mittels seiner Schalttabelle eine Schaltfunktion. Der Term  $x \bullet (y + z)$  realisiert z.B. die Funktion  $f: \{0,1\}^3 \rightarrow \{0,1\}$  mit:

$$\begin{array}{llll} f(0,0,0) = 0 & f(0,0,1) = 0 & f(0,1,0) = 0 & f(0,1,1) = 0 \\ f(1,0,0) = 0 & f(1,0,1) = 1 & f(1,1,0) = 1 & f(1,1,1) = 1 \end{array}$$

Verschiedene Terme können durchaus dieselbe Schaltfunktion beschreiben, wie zum Beispiel im Fall der Terme  $t_1 = x \bullet (y + z)$  und  $t_2 = (x \bullet y) + (x \bullet z)$ . Dies erkennt man durch Tabellierung der zugehörigen Schaltfunktionen oder durch Analyse der zugehörigen Schaltkreise.

### 5.2.6 Gleichungen

Eine Gleichung  $t_1 = t_2$  besteht aus zwei Termen, die dieselbe Schaltfunktion beschreiben. Um nachzuweisen, dass eine Gleichung  $t_1 = t_2$  gilt, kann man daher die Schaltfunktionen der beiden Terme tabellieren und die Ergebnisse vergleichen. Während die Gleichung  $x \bullet (y + z) = x \bullet y + x \bullet z$  noch vertraut aussieht, überrascht vielleicht die folgende Gleichung:

$$x + (y \bullet z) = (x + y) \bullet (x + z).$$

Durch Tabellierung erhalten wir aber identische Ergebnisse für alle Belegungen der Variablen:

<b>x</b>	<b>y</b>	<b>z</b>	$y * z$	$x + (y * z)$	$(x+y)$	$(x+z)$	$(x+y) * (x+z)$
0	0	0	0	0	0	0	0
0	0	1	0	0	1	0	0
0	1	0	0	1	0	0	0
0	1	1	1	1	1	1	1
1	0	0	0	1	1	1	1
1	0	1	0	1	1	1	1
1	1	0	0	1	1	1	1
1	1	1	1	1	1	1	1

Abb. 5.14: Vergleich der Schaltfunktionen  $x + (y * z)$  und  $(x+y) * (x+z)$

Es gibt eine recht überschaubare Menge von gültigen Gleichungen, aus denen sich alle anderen Gleichungen ableiten lassen. Eine Struktur, die diese Gleichungen erfüllt, heißt *distributiver Verband*.

$x + x = x$	Idempotenz	$x * x = x$
$x + y = y + x$	Kommutativität	$x * y = y * x$
$x + (y + z) = (x + y) + z$	Assoziativität	$x * (y * z) = (x * y) * z$
$x * (x + y) = x$	Absorption	$x + (x * y) = x$
$x * (y + z) = (x * y) + (x * z)$	Distributivität	$x + (y * z) = (x + y) * (x + z)$

Abb. 5.15: Gleichungen eines distributiven Verbandes

## 5.2.7 Dualität

Es fällt auf, dass sich die Gleichungen in der linken und in der rechten Spalte entsprechen, sofern man  $+$  durch  $\bullet$  und  $\bullet$  durch  $+$  ersetzt. Dieses Phänomen ist unter dem Namen *Dualität* bekannt. Die Terme 0 und 1, die den immer geöffneten, bzw. den immer geschlossenen Schaltkreis bezeichnen, werden durch die folgenden Gleichungen charakterisiert. Auch hier entsprechen sich die Gleichungen, wenn man zusätzlich noch 0 mit 1 vertauscht:

$$\begin{array}{ll} x + 0 = x & x \bullet 1 = x \\ x + 1 = 1 & x \bullet 0 = 0 \end{array}$$

Allgemein erhalten wir den zu einem Term  $t$  dualen Term  $t^d$  wenn wir  $+$  und  $\bullet$  sowie 0 und 1 vertauschen. Beispielsweise ist  $(x+0)\bullet y$  dual zu  $(x\bullet 1)+y$ . Durch zweimaliges Dualisieren erhalten wir den alten Term zurück. Wir stellen also fest, dass mit jeder Gleichung  $t_1 = t_2$  auch die duale Gleichung  $t_1^d = t_2^d$  gilt. Dieses *Dualitätsprinzip* setzt sich auch auf alle Gleichungen fort, die wir aus den Basisgleichungen in Abb. 5.15 folgern können. Wir haben also immer  $t^{dd} = t$ , und zu jeder Gleichung  $t_1 = t_2$  automatisch auch die duale Gleichung  $t_1^d = t_2^d$ .

### 5.2.8 SP-Schaltungen sind monoton

Wir wissen schon, dass jeder Term eine Schaltfunktion realisiert, es gibt aber Schaltfunktionen, die mit den bisher gesehenen Verknüpfungen,  $+$  und  $\cdot$  nicht realisierbar sind. Ein einfaches Beispiel ist eine Wechselschaltung, bei der eine Lampe durch zwei Schalter  $x$  und  $y$  unabhängig voneinander ein- oder ausgeschaltet werden soll. Gehen wir davon aus, dass am Anfang, wenn beide Schalter ausgeschaltet sind, die Lampe nicht brennen soll, also  $f(0, 0) = 0$ , so ergeben sich die restlichen Einträge aus der Forderung, dass bei jeder Veränderung eines der Schalter sich der Zustand der Lampe verändern muss:

$x$	$y$	Lampe
0	0	0
0	1	1
1	0	1
1	1	0

Abb. 5.16: Schalttabelle für Wechselschalter

Der Grund warum diese Schaltfunktion nicht allein mit  $+$  und  $\cdot$  realisierbar ist, liegt darin, dass sowohl  $+$  als auch  $\cdot$  monotone Operationen in folgendem Sinne sind: Setzt man  $0 \leq 1$  dann wird diese Ordnung von den Operationen  $+$  und  $\cdot$  respektiert, das bedeutet:

$$x_1 \leq y_1 \wedge x_2 \leq y_2 \Rightarrow x_1 + x_2 \leq y_1 + y_2$$

und analog für  $\cdot$ . Kompositionen von monotonen Operationen sind offensichtlich monoton. Für den Wechselschalter haben wir aber  $f(0, 1) = 1$  und  $f(1, 1) = 0$ , somit ist die gewünschte Funktion  $f$  nicht monoton, kann also nicht allein aus  $+$  und  $\cdot$  aufgebaut werden.

### 5.2.9 Negation

Die einfachste nicht monotone Schaltfunktion ist durch die folgende Tabelle gegeben. Wenn der Schalter  $x$  offen ist, ist das Schaltglied geschlossen – und umgekehrt. Das entsprechende Schaltglied heißt *Negation*. Als Operationszeichen benutzen wir ein Apostroph:  $'$ .

$x$	$x'$
0	1
1	0

Abb. 5.17: Schalttabelle für Negation

**Definition:** Ist  $S$  ein Schaltglied, so sei  $S'$  dasjenige Schaltglied, das genau dann offen ist, wenn  $S$  geschlossen ist.  $S'$  heißt die Negation von  $S$ .

Für die zweifache Negation gilt offensichtlich:  $S'' = S$ . Andere Namen für die Negation sind auch: *Komplement* oder *Inverses*. Statt dem Apostroph ' verwendet man gelegentlich auch einen Überstrich, also  $\bar{x}$  statt  $x'$ .

In elektrischen Schaltkreisen lässt sich die Negation durch ein *Relais* realisieren: Fließt Strom durch  $S$ , so wird durch die Magnetwirkung einer Spule der Schalter  $S'$  geöffnet. Mit Transistoren gelingt die Realisierung der Negation einfacher und natürlicher, siehe Abschnitt 5.3.

### 5.2.10 Boolesche Terme

Ein Schaltkreis, in dem neben Serien- und Parallel-Schaltung auch noch die Negation verwendet werden darf, heißt *boolesche Schaltung*. Der einer booleschen Schaltung entsprechende Term heißt *boolescher Term*. Formal definieren wir:

- (i) 0 und 1 sind boolesche Terme.
- (ii) Jede Variable ist ein boolescher Term.
- (iii) Sind  $t, t_1$  und  $t_2$  boolesche Terme, so auch:  $t_1 + t_2$ ,  $t_1 \cdot t_2$  und  $t'$ .

Die Gleichheit boolescher Terme definiert man analog zu der Gleichheit von SP-Termen: Zwei Terme heißen gleich, wenn sie identische Schaltfunktionen besitzen. Als Beispiel einer booleschen Gleichung betrachten wir die *deMorgansche Regel*

$$(x + y)' = x' \cdot y'$$

und ihre Herleitung durch Vergleich der entsprechenden Spalten der Schaltfunktionen.

<b>x</b>	<b>y</b>	<b>x+y</b>	<b>(x+y)'</b>	<b>x'</b>	<b>y'</b>	<b>x'•y'</b>
<b>0</b>	<b>0</b>	<b>0</b>	<b>1</b>	<b>1</b>	<b>1</b>	<b>1</b>
<b>0</b>	<b>1</b>	<b>1</b>	<b>0</b>	<b>1</b>	<b>0</b>	<b>0</b>
<b>1</b>	<b>0</b>	<b>1</b>	<b>0</b>	<b>0</b>	<b>1</b>	<b>0</b>
<b>1</b>	<b>1</b>	<b>1</b>	<b>0</b>	<b>0</b>	<b>0</b>	<b>0</b>

Abb. 5.18: Ablesen der Gleichheit  $(x+y)' = x' \cdot y'$  aus der Wertetabelle

Die wichtigsten Gleichungen, die das Verhalten der Negation bestimmen, sind:

$$(x + y)' = x' \cdot y' \quad \text{de Morgansche Regeln} \quad (x \cdot y)' = x' + y'$$

$$x + x' = 1 \quad \text{Komplementregeln} \quad x \cdot x' = 0$$

$$x'' = x$$

Eine algebraische Struktur, in der neben den Gleichungen eines distributiven Verbandes und den Gleichungen für 0 und 1 auch noch die obigen Komplementgleichungen gelten, heißt *boolesche Algebra*.

### 5.2.11 Dualitätsprinzip

Auch für boolesche Terme, die die Negation enthalten, gilt ein Dualitätsprinzip. Allerdings muss man beim Dualisieren die Negation ignorieren, insgesamt hat man dann:

$$\begin{aligned}x^d &= x, \text{ wenn } x \text{ eine Variable ist, } 0^d = 1 \text{ und } 1^d = 0 \\(t_1 + t_2)^d &= t_1^d \bullet t_2^d, \text{ sowie } (t_1 \bullet t_2)^d = t_1^d + t_2^d, \text{ aber} \\(t')^d &= (t^d)'.\end{aligned}$$

Aus dieser Definition kann man durch einfache strukturelle Induktion beweisen:

*Ist  $t = t(x_1, \dots, x_n)$ , dann erhält man den dualen Term, indem man alle Variablen komplementiert und danach das Ergebnis, kurz:  $t^d = t(x_1', \dots, x_n')$ .*

Beispielsweise erhalten wir den zu  $t = x \bullet y + x' \bullet z$  dualen Term:  
 $t^d = (x \bullet y + x' \bullet z)^d = (x' \bullet y' + x'' \bullet z')' = (x' \bullet y')' \bullet (x'' \bullet z')' = (x'' + y'') \bullet (x''' + z'') = (x + y) \bullet (x' + z)$ .

### 5.2.12 Realisierung von Schaltfunktionen

In der Praxis stellt sich häufig das Problem, zu einer gegebenen Schaltfunktion einen entsprechenden booleschen Term zu finden, der diese Schaltfunktion realisiert. Dazu betrachten wir zunächst spezielle boolesche Terme, so genannte *Literale* wie z.B.  $x$ ,  $x'$ ,  $x_1$ ,  $x_3'$  und *Monome*, wie z.B.  $x'yz'$ ,  $xy'z$  oder  $x_1'x_2'x_3x_4$ . Wir haben hier, wie auch in der Arithmetik üblich, das Multiplikationszeichen weggelassen, d.h. wir schreiben kurz:  $t_1t_2$  für  $t_1 \bullet t_2$ .

**Definition:** Ein Literal ist eine Variable oder eine negierte Variable. Ein Monom ist ein Produkt von Literalen.

Die Schaltfunktion eines Monoms kann nur dann 1 sein, wenn jedes darin vorkommende Literal 1 ist, d.h. wenn jede vorkommende Variable mit 1 und jede vorkommende negierte Variable mit 0 belegt ist.  $x'yz'$  ist also nur dann 1, falls  $x = z = 0$  und  $y = 1$  sind. Der boolesche Term, der aus der Summe zweier Monome besteht, hat genau dort eine 1, wo mindestens eines der Monome eine 1 hat. So hat der Term  $t = x'yz' + xy'z$  eine 1 für  $x = z = 0, y = 1$  sowie für  $x = z = 1, y = 0$ . Durch Summierung von geeigneten Monomen kann man also an beliebigen Stellen einer Schaltfunktion eine 1 realisieren.

Als Beispiel sei eine Schaltfunktion gesucht, die es gestattet, eine Lampe von drei verschiedenen Schaltern  $x$ ,  $y$  und  $z$  unabhängig ein- und auszuschalten. Die gesuchte Schaltfunktion  $g(x, y, z)$  ist in der linken Tabelle spezifiziert:

x	y	z	$g(x,y,z)$	$m_1$	$m_2$	$m_3$	$m_4$	$m_1+m_2+m_3+m_4$
0	0	0	0	0	0	0	0	0
0	0	1	1	0	0	0	1	1
0	1	0	1	0	1	0	0	1
0	1	1	0	0	0	0	0	0
1	0	0	1	0	0	1	0	1
1	0	1	0	0	0	0	0	0
1	1	0	0	0	0	0	0	0
1	1	1	1	0	0	0	1	1

Abb. 5.19: Schaltfunktion - Spezifikation und Realisierung als Summe von Monomen

Diese Schaltfunktion liefert an vier Stellen den Wert 1, sie lässt sich also als Summe von vier Monomen  $m_1, m_2, m_3, m_4$  schreiben. Die benötigten Monome sind:

$$m_1 = x'y'z, \quad m_2 = x'yz', \quad m_3 = xy'z', \quad m_4 = xyz.$$

Der gesuchte boolesche Term ist daher:

$$g(x, y, z) = m_1 + m_2 + m_3 + m_4 = x'y'z + x'yz' + xy'z' + xyz.$$

Der durch die obige Vorgehensweise gebildete Term hat eine spezielle Form, die man auch als *disjunktive Normalform* (DNF) bezeichnet. Darunter versteht man eine Summe von Monomen, wobei verlangt ist, dass jede Variable in jedem Monom (entweder direkt oder negiert) vorkommt. Zu jeder Schaltfunktion gibt es dann genau eine disjunktive Normalform und diese lässt sich auf die oben beschriebene Weise gewinnen.

Das Verfahren ist offensichtlich für jede Schaltfunktion durchführbar, so dass gilt:

Jede Schaltfunktion lässt sich durch einen booleschen Term in DNF realisieren.

### 5.2.13 Konjunktive Normalform

Die vorgestellte Methode liefert für jede Schaltfunktion einen booleschen Term, welcher um so komplizierter ist, je mehr 1-en die Schaltfunktion hat. Das Dualitätsprinzip deutet eine zweite Vorgehensweise an, die immer dann sinnvoll ist, wenn die Schaltfunktion mehr Einsen als Nullen hat.

Wir definieren eine *Elementarsumme* als Summe von Literalen. Die Schaltfunktion einer Elementarsumme ergibt genau für einen Input eine 0, sonst immer 1. Sind  $e_1$  und  $e_2$  Elementarsummen, so hat das Produkt  $e_1e_2$  genau dort eine 0, wo  $e_1$  oder  $e_2$  eine 0 haben. Jede Schaltfunktion kann man als Produkt von Elementarsummen schreiben.

**Beispiel:** Gegeben sei die Schaltfunktion  $h(x, y, z)$ , durch die Werte in der vierten Spalte der folgenden Tabelle.

x	y	z	$h(x,y,z)$		x	y	z	$e_1$	$e_2$	$e_3$	$e_1 \cdot e_2 \cdot e_3$
0	0	0	0		0	0	0	0	1	1	0
0	0	1	1		0	0	1	1	1	1	1
0	1	0	1		0	1	0	1	1	1	1
0	1	1	0		0	1	1	0	1	0	0
1	0	0	1		1	0	0	1	1	1	1
1	0	1	0		1	0	1	1	0	0	0
1	1	0	1		1	1	0	1	1	1	1
1	1	1	1		1	1	1	1	1	1	1

Abb. 5.20: Schaltfunktion und Darstellung als konjunktive Normalform

Die drei Nullwerte geben Anlass für drei Elementarsummen  $e_1 = x + y + z$ ,  $e_2 = x + y' + z'$  und  $e_3 = x' + y + z'$ . Ihr Produkt ergibt den gesuchten booleschen Term:

$$h(x, y, z) = e_1 e_2 e_3 = (x + y + z)(x + y' + z')(x' + y + z').$$

Den so gewonnenen Term nennen wir auch *konjunktive Normalform*.

### 5.2.14 Algebraische Umwandlung in DNF oder KNF

Die Gesetze der booleschen Algebra erlauben auch eine direkte Umwandlung eines beliebigen Terms in seine disjunktive bzw. konjunktive Normalform. Dies geschieht in drei Schritten, die wir am Beispielterm  $t = (x \cdot y)' \cdot (y \cdot z)' + (y' + (z + x))'$  nachvollziehen wollen:

- deMorgansche Regeln anwenden, um ' nach innen zu bringen  

$$\begin{aligned} t &= (x \cdot y)' \cdot (y \cdot z)' + (y' + (z + x))' \\ &= (x' + y') \cdot (y' + z') + y'' \cdot (z + x)' \\ &= (x' + y') \cdot (y' + z') + y'' \cdot z' \cdot x' \end{aligned}$$
- doppelte Negationen entfernen,  

$$t = (x' + y') \cdot (y' + z) + y \cdot z' \cdot x'$$
- ausdistribuieren - für DNF mit  $a \cdot (b+c) = a \cdot b + a \cdot c$ , für KNF mit  $a+b \cdot c = (a+b) \cdot (a+c)$ ,  

$$\begin{aligned} t &= (x' + y') \cdot (y' + z) + y \cdot z' \cdot x' \\ &= x' \cdot y' + x' \cdot z + y' \cdot y' + y' \cdot z + y \cdot z' \cdot x' \\ &= x'y' + x'z + y'y' + y'z + yz'x' \end{aligned}$$
- gleiche Terme zusammenfassen und umordnen,  

$$t = x'y' + x'z + y' + y'z + x'y'z'$$
- verschmelzen (Absorption) - für DNF mit  $a + (a \cdot b) = a$ , für KNF mit  $a \cdot (a + b) = a$ .  

$$\begin{aligned} t &= x'y' + x'z + y' + x'y'z' \\ &= x'z + y' + x'y'z' \end{aligned}$$

Jetzt hat man den Term als Summe von Monomen (bzw. als Produkt von Elementarsummen) dargestellt. Meist lässt man diese Form schon als DNF (KNF) gelten. Genau genommen müsste man aber noch jedes Monom (jede Elementarsumme) durch die Verwendung der Gleichungen  $a = a \cdot 1 = a \cdot (b+b') = a \cdot b + a \cdot b'$  ( bzw.  $a = (a+b) \cdot (a+b')$  ) aufblähen, so dass es alle Variablen enthält. Im Beispiel:

$$\begin{aligned} t &= x'z + y' + x'y'z' \\ &= x'y'z + x'y'z + y' + x'y'z' \\ &= x'y'z + x'y'z + x'y' + x'y' + x'y'z' \\ &= x'y'z + x'y'z + x'y'z + x'y'z' + x'y'z + x'y'z' + x'y'z' \\ &= x'y'z + x'y'z + x'y'z + x'y'z' + x'y'z' + x'y'z'. \end{aligned}$$

Da wir für die Implementierung aber einen möglichst einfachen Term anstreben, werden wir auf diesen letzten Schritt in der Regel verzichten.

### 5.2.15 Aussagenlogik

George Boole hatte die Absicht, die *Gesetze des Denkens* zu formalisieren. Er ging dazu von *Elementaraussagen* aus, von denen er lediglich verlangte, dass sie entweder wahr ( $T = \text{true}$ ) oder falsch ( $F = \text{false}$ ) sind. Beispiele solcher Elementaraussagen sind z.B.

„ $2 + 2 = 5$ “

„Microsoft ist eine Biersorte.“

„Blei ist schwerer als Wasser.“

„Jede ungerade Zahl größer als 3 ist Summe zweier Primzahlen.“.

Durch Verknüpfung mit den logischen Operationen  $\wedge$  (und),  $\vee$  (oder),  $\neg$  (nicht) erhält man neue, zusammengesetzte Aussagen. Formal:

- Jede Elementaraussage ist eine Aussage.
- Sind  $A$ ,  $A_1$  und  $A_2$  Aussagen, so auch  $A_1 \vee A_2$ ,  $A_1 \wedge A_2$  und  $\neg A$ .

Der *Wahrheitswert* einer zusammengesetzten Aussage berechnet sich aus den Wahrheitswerten der Teilaussagen anhand der *Wahrheitstabellen*. Die Wahrheitstabellen für  $\vee$ ,  $\wedge$  und  $\neg$  ergeben sich aus den entsprechenden Tabellen für  $+$ ,  $\cdot$  und  $'$ , indem man überall 0 durch  $F$  und 1 durch  $T$  ersetzt (siehe auch S. 16):

$\vee$	F	T
F	F	T
T	T	T

$\wedge$	F	T
F	F	F
T	F	T

$\neg$		
F	T	
T	F	

Abb. 5.21: Verknüpfungstabellen der logischen Operatoren Oder, Und und Negation

Für die Äquivalenz von Aussagen gelten genau die Gleichungen der booleschen Algebra. Man muss lediglich  $+$ ,  $\cdot$ ,  $'$ ,  $0$ ,  $1$  durch  $\vee$ ,  $\wedge$ ,  $\neg$ ,  $F$ ,  $T$  ersetzen. Beispielsweise hat man:

$A \vee (A \wedge B) = A$ , d.h. eine Aussage der Form  $A \vee (A \wedge B)$  ist genau dann wahr, wenn  $A$  wahr ist. Zusätzliche logische Verknüpfungen kann man als Kombination aus den vorhandenen definieren, zum Beispiel:

$$A \Rightarrow B \quad \text{durch} \quad \neg A \vee B.$$

### 5.2.16 Mengenalgebra

Ausgehend von einer festen Grundmenge  $M$  betrachten wir  $\mathcal{P}(M)$ , die Menge aller Teilmengen von  $M$ . Auf  $\mathcal{P}(M)$  untersuchen wir die Operationen  $\cup$ ,  $\cap$ ,  $\neg$  (Vereinigung, Schnitt und Komplement). Hier gelten die Gleichungen der booleschen Algebra, wenn man  $+$ ,  $\cdot$ ,  $'$ ,  $0$ ,  $1$  durch  $\cup$ ,  $\cap$ ,  $\neg$ ,  $\emptyset$  und  $M$  ersetzt. Beispielsweise gilt für beliebige  $U, V \in \mathcal{P}(M)$  die deMorgansche Regel  $U \cap V = U \cup V$

## 5.3 Digitale Logik

In der *digitalen Logik* geht es darum, Schaltfunktionen technisch zu realisieren. In der Anfangszeit der Informatik wurden dazu in der Tat seriell-parallele Schaltkreise zusammen mit Relais eingesetzt, mit welchen man die Negation realisieren konnte. Später benutzte man Vakuumröhren, danach Transistoren als Elementarschalter. In der Optoelektronik gibt es auch optische Schalter die entsprechend kombiniert werden. In allen Fällen ist es notwendig, zunächst gewisse elementare Schaltglieder in der gewählten Technik zu realisieren. Mit diesen elementaren Bausteinen als Ausgangsbasis können wir den Aufbau eines Rechners unabhängig von der angewendeten Technik komplett verstehen.

Die gegenwärtig dominierende Technik ist die CMOS-Technik und wir werden auf die Besonderheiten dieser Technik eingehen und auf einige Kunstgriffe, die uns diese Technik zur besonders effizienten Lösung von Problemen anbietet.

### 5.3.1 Logikgatter

Schaltfunktionen stellt man gerne graphisch als „schwarzen Kasten“ (black box) dar, in den man vorne (oder oben) die Eingaben füttert, und bei dem hinten (oder unten) das Ergebnis herauskommt. Diese Darstellung hat den Vorteil, dass man die Komposition  $f(g_1, \dots, g_n)$  von Schaltfunktionen  $f$ , und  $g_1, \dots, g_n$  besonders anschaulich darstellen kann, denn dabei werden einfach die Ausgänge der  $g_1, \dots, g_n$  in die Eingänge von  $f$  geführt. Die folgende Zeichnung stellt dies für den Fall  $f=OR$ ,  $g_1=x \cdot y$  und  $g_2=x \cdot y'$  dar.

Die Funktionen AND und OR haben jeweils zwei Argumente. Ihre Box-Darstellung hat also jeweils links zwei Eingänge und rechts einen Ausgang. Die Funktion NOT hat nur einen Eingang und einen Ausgang. Der Ausgang des ersten NOT und der Eingang  $y$  führen zum Eingang des ersten AND, die Ausgänge der AND-Boxen werden zu den Eingängen des OR.

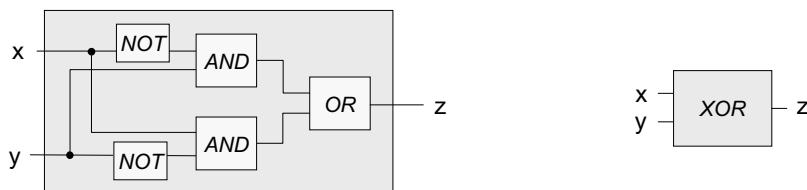


Abb. 5.22: Komposition von Funktionen und resultierende Funktion.

Insgesamt wird daraus eine Funktion mit zwei Eingängen,  $x$  und  $y$ , und einem Ausgang. Dieser Funktion kann man jetzt einen Namen geben, z.B. XOR, und sie als Funktionsbox mit Eingängen  $x$  und  $y$  und Ausgang  $z$  darstellen. Sie stellt offenbar die Funktion  $f(x, y) = x' \cdot y + x \cdot y'$  dar. Diese bezeichnet man oft auch mit  $\oplus$ , also  $x \oplus y = x' \cdot y + x \cdot y'$ .

Man beachte, dass sich die Eingänge  $x$  und auch  $y$  aufspalten.  $x$  führt sowohl in den Eingang des obersten NOT als auch direkt in den Eingang des unteren AND. Die Aufspaltungsstelle ist mit einem Punkt (der manchen Praktiker an eine Lötstelle erinnern wird), markiert. Im Gegensatz dazu kreuzen sich die Eingangslinien von  $x$  und  $y$  zu den AND-Gliedern, ohne sich zu berühren. Solche Überkreuzungen sind beim Zeichnen auf zweidimensionalem Papier unvermeidlich. Dass sie sich nicht berühren sollen wird an der fehlenden „Lötstelle“ deutlich.

Eingänge dürfen sich aufspalten; im Gegensatz dazu dürfen sich Ausgänge nicht gegenseitig berühren. Dies ist in gewissen Techniken zwar möglich, man spricht dann von einem „wired or“, in den meisten Techniken, so auch in der CMOS-Technik führt eine Berührung von Ausgangsleitungen zu einem Kurzschluss. Dieser ist in der Regel dadurch zu vermeiden, dass die Ausgangsleitungen durch ein OR zusammengeführt werden, wie auch in unserem Beispiel.

Da wir wissen, dass sich jede Schaltfunktion aus den Operationen AND ( $\bullet$ ) OR( $+$ ) und NOT( $'$ ) aufbauen lässt, reichen diese als Basisschaltglieder eigentlich aus. Es gibt eine offizielle Norm der IEC (*International Electric Commission*), in der die offizielle Aufschrift und Form der sogenannten Logikgatter definiert ist. Diese ist in der folgenden Figur zu sehen.

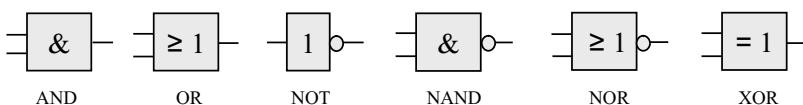


Abb. 5.23: Offizielle Gatterdarstellung der IEC

Der Kreis am Ausgang mancher der Schaltglieder soll bedeuten, dass das Ergebnis negiert wird – aus dem AND wird durch Negierung eine NAND. Ein allein stehendes Negationsglied zeichnet man als Identitätsfunktion (1) gefolgt von einem Negationskreis. Die Aufschrift  $\geq 1$  auf dem OR-Glied soll andeuten, dass das Ergebnis der Funktion 1 ist, wenn mindestens ein Eingang 1 ist. Analog ist die Aufschrift auf dem XOR-Glied zu verstehen.

Diese normierte Darstellung der IEC sollte die früher gebräuchliche Kennzeichnung der Gattersymbole anhand ihrer Form ablösen. Sie konnte sich aber nicht durchsetzen, was 1991 auch die IEC eingestehen musste. Vor allem in der englischsprachigen Literatur dominiert daher

weiter die Unterscheidung der Gattersymbole durch ihre Form, wie sie in der folgenden Abbildung zu sehen sind, und wie wir sie auch im Folgenden verwenden wollen.

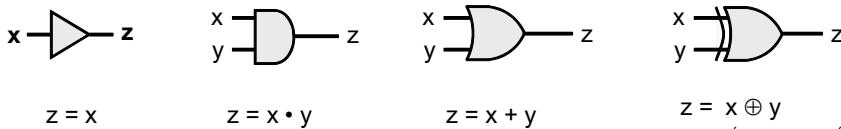


Abb. 5.24: Gattersymbole

Die einfachste Schaltung gibt den Input unverändert an den Output weiter. Sie wird als *Puffer* (engl. *buffer*) bezeichnet. Auf der logischen Ebene scheint dieses Gatter nutzlos zu sein, in der Praxis stellt es ein Verstärkerglied dar, das dafür sorgt, dass ein analoger Spannungswert in dem zugehörigen Bereich für logisch 0 bzw. logisch 1 gehalten wird. AND bzw. OR Gatter stellen boolesche Konjunktion  $\cdot$  bzw. Disjunktion  $+$  dar. Das XOR-Gatter entspricht dem *exklusiven oder* und kann als  $z = x' \cdot y + x \cdot y'$  definiert werden. Da es der bitweisen Addition ohne Übertrag entspricht, schreibt man häufig auch  $z = x \oplus y$  und spendiert dieser Operation eine eigene Gatterdarstellung.

Das Symbol für das NOT-Glied setzt sich aus dem Puffer-Symbol und einem kleinen Kreis zusammen. Dieser Kreis taucht auch in anderen Schaltgliedern auf und deutet an, dass das Signal entlang der bezeichneten Leitung invertiert wird. Der Kreis macht aus dem AND ein NAND und aus dem OR ein NOR. Übrigens werden wir sehen, dass in der CMOS-Technik die negierten Varianten grundlegender sind, als die unnegierten. Gelegentlich findet man die elementaren Gattersymbole für OR und AND auch mit mehr als zwei Inputs, oder mit einer oder mehreren negierten Eingangsleitungen wie in dem folgenden Bild gezeigt.

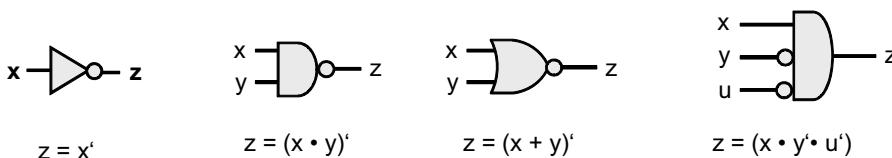


Abb. 5.25: Gattersymbole mit Negation

Es ist leicht einzusehen, dass nicht alle Gatter unbedingt notwendig sind. Man könnte theoretisch sogar allein mit dem NAND-Gatter auskommen. Das Komplement kann man als  $x' = x \text{ NAND } x$  gewinnen, dann die Konjunktion durch  $x \cdot y = (x \text{ NAND } y)'$  und die Disjunktion durch  $x + y = (x' \text{ NAND } y')$ .

### 5.3.2 Entwurf und Vereinfachung boolescher Schaltungen

Jede boolesche Funktion  $f$  lässt sich durch einen Schaltkreis realisieren. Aus der Schalttabelle für  $f$  lesen wir die Monome ab, die den Wert 1 zur gesuchten booleschen Funktion beitragen. Jedes dieser Monome wird durch ein entsprechendes AND-Gatter realisiert und diese werden

zum Schluss in einem OR-Gatter summiert. Wir demonstrieren dies an einem kleinen Beispiel und verwenden ab sofort auch die Gatterdarstellung boolescher Funktionen.

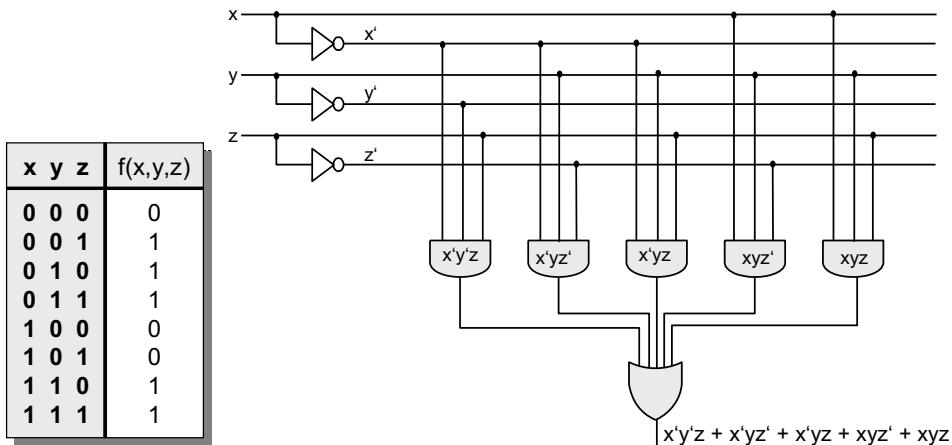


Abb. 5.26: Realisierung einer booleschen Funktion

Die Darstellung der Funktion  $f(x, y, z) = x'y'z + x'yz' + x'yz + xyz' + xyz$  aus Abb. 5.26 kann noch vereinfacht werden, womit wir evtl. Schaltglieder einsparen können. Die Summe des ersten und dritten Monoms vereinfacht zu  $x'z(y' + y) = x'z$ . Analog vereinfacht die Summe des zweiten und vierten Monoms zu  $x'yz' + xyz' = yz'$ . Insgesamt erhalten wir also  $f(x, y, z) = x'z + yz' + xyz$ . Damit haben wir bereits zwei AND-Gatter eingespart und eine weitere Vereinfachung scheint zunächst nicht mehr möglich.

Die möglichen Vereinfachungsschritte kann man auch schon in der ursprünglichen Schalttabelle erkennen. Es gilt  $f(0, 0, 1) = f(0, 1, 1) = 1$  was besagt, dass  $f(0, y, 1) = 1$  ist, unabhängig vom Wert von  $y$ . In der DNF-Darstellung von  $f$  erhalten wir also den Summanden  $x'z$ . Analog gilt  $f(0, 1, 0) = f(1, 1, 0) = 1$  was das vereinfachte Monom  $yz'$  in der DNF von  $f$  ergibt. Zusammen mit dem Monom  $xyz$  welches für den Funktionswert  $f(1, 1, 1) = 1$  verantwortlich ist, erhalten wir also die obige vereinfachte Darstellung  $f(x, y, z) = x'z + yz' + xyz$ .

### 5.3.3 KV-Diagramme

Offenbar lohnt es sich, Zeilen in der Schalttabelle zu suchen, in denen der Funktionswert 1 ist, und deren  $(x, y, z)$ -Werte sich nur in einer Komponente unterscheiden, wie etwa  $(0, 0, 1)$  und  $(0, 1, 1)$  bzw.  $(0, 1, 0)$  und  $(1, 1, 0)$ . Dies wird erheblich durch eine Darstellung der Schaltfunktion in einem sogenannten *Karnaugh-Veitch-Diagramm* (kurz: KV-Diagramm) vereinfacht. Jeder Kombination von Eingabewerten entspricht ein Kästchen in einer Tabelle, in dem der zugehörige Funktionswert eingetragen wird. Die Kästchen sind so angeordnet, dass Eingabekombinationen, die sich nur an einer Position unterscheiden, nebeneinanderliegen. Für den Fall einer dreistelligen Funktion ergibt sich eine  $2 \times 4$ -Tabelle, deren Zeilen den Werten von  $x$

entsprechen und deren Spalten den Kombinationen von  $y$  und  $z$ . Dabei sind die Spaltenbezeichnungen so angeordnet, dass sich zwei benachbarte immer in genau einer Position unterscheiden. Eine solche Anordnung nennt man auch *Gray-Code*. Eigentlich müsste die letzte Spalte zur ersten Spalte benachbart sein, so dass man sich die Tabelle auf einen Zylinder aufgewickelt vorstellen sollte.

	$yz$	00	01	11	10
$x$	0	0	1	1	1
	0	0	0	1	1
	1	0	0	1	1

	$yz$	00	01	11	10
$x$	0	0	1	1	1
	0	0	0	1	1
	1	0	0	1	1

Abb. 5.27: KV-Diagramme für eine dreistellige boolesche Funktion

In einem KV-Diagramm suchen wir nun nach Rechtecken mit Seitenlängen 1, 2 oder 4, die gänzlich mit 1-en gefüllt sind. Solche Rechtecke entsprechen gerade den Monomen in denen einige Variablen nicht vorkommen. Beispielsweise entsprechen dem  $1 \times 2$ -Rechteck und dem  $2 \times 1$ -Rechteck der linken Figur gerade die Monome  $x'z$  und  $yz'$ . Zusammen mit dem trivialen  $1 \times 1$ -Rechteck für  $xyz$  überdecken diese genau alle 1-en der Operationstabelle, so dass sich für die Schaltfunktion die Darstellung  $f(x, y, z) = x'z + yz' + xyz$  ergibt.

In der rechten Figur erkennen wir, dass wir die 1-en der Tabelle auch mit zwei Rechtecken hätten abdecken können, nämlich dem  $2 \times 2$ -Rechteck welches dem Monom  $y$  entspricht und dem  $1 \times 2$ -Rechteck für das Monom  $x'z$ . Dass diese Rechtecke sich überlappen, ist unerheblich. Auf diese Weise erreichen wir eine noch einfachere Darstellung der Funktion  $f$  als  $f(x, y, z) = x'z + y$ . Gegenüber der in Abb. 5.26 gezeigten Implementierung erzielen wir also eine erhebliche Ersparnis an logischen Gattern.

Die Vereinfachung boolescher Funktionen mit Hilfe von KV-Diagrammen funktioniert auch noch für Funktionen mit vier Variablen, z.B.  $x, y, z, u$ . Dabei trägt man auf einer Achse alle vier Kombinationen von  $x$  und  $y$  auf und auf der anderen alle Kombinationen von  $z$  und  $u$ . Für die Reihenfolge der Kombinationen verwendet man wieder den bekannten Gray-Code, so dass nebeneinanderliegende Kästchen sich in genau einer Position unterscheiden. Umgekehrt liegen die Kästchen für Variablenkombinationen die sich in genau einer Position unterscheiden nicht notwendig nebeneinander. Dazu müsste man in Gedanken die obere Kästchenreihe an die untere ankleben und die rechte Spalte an die linke Spalte. In der Figur wären dann insbesondere alle vier Eckfelder benachbart.

Im rechten Diagramm der folgenden Abbildung zeigen die Balken bei den Variablennamen die Zeilen bzw. Spalten an, in denen die betreffende Variable unnegiert erscheint. So steht z.B. das Feld mit Inhalt 1 in dem zentralen Quadrat für das Monom  $xy'uz'$ . Das über den Rand reichende „Quadrat“ in der zweiten und dritten Zeile und der ersten und vierten Spalte hat die Koordinaten  $xu'$ . Die Überdeckung durch Rechtecke, die in der rechten Figur gezeigt ist liefert, von links nach rechts übersetzt, die Darstellung der booleschen Funktion  $f(x, y, z, u) = xu' + x'z'u + xy'uz' + x'z + zu'$ .

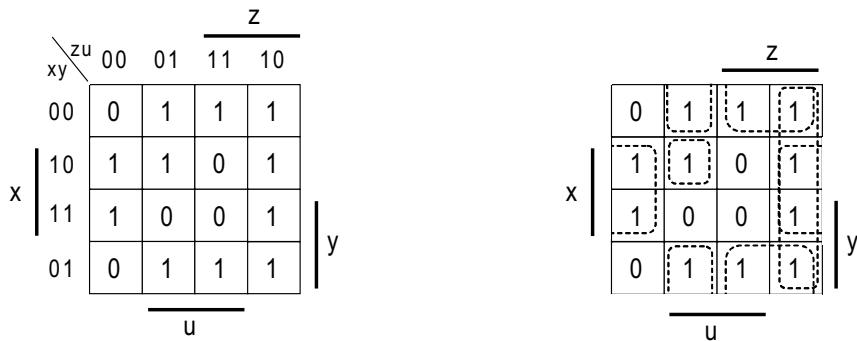


Abb. 5.28: Karnaugh-Veitch-Diagramme für eine vierstellige boolesche Funktion

Für Funktionen mit mehr als 4 Variablen lässt sich die Vereinfachung von booleschen Funktionen durch Überdeckung der 1-gefüllten Positionen der Operationstabelle mit Rechtecken nicht mehr anschaulich durch KV-Diagramme darstellen. Die Methode an sich funktioniert aber weiterhin. Den maximalen 1-gefüllten Rechtecken entsprechen die sogenannten *Primimplikanten*. Ein *Primimplikant* ist ein Monom mit möglichst wenigen Literalen, für das die Funktion  $f$  den Wert 1 hat. Im obigen Beispiel ist z.B.  $x'z'u$  ein Primimplikant: Wo das Monom  $x'z'u$  den Wert 1 hat, gilt auch  $f(x, y, z, u) = 1$ , also  $x'z'u \Rightarrow f(x, y, z, u)$  (d.h.  $x'z'u$  ist ein *Implikant*) und kein echter Teil des Monoms ist noch ein Implikant.

Den Primimplikanten entsprechen gerade die maximalen 1-gefüllten Rechtecke der KV-Diagramme. Die Suche nach optimalen Implementierungen von booleschen Funktionen wird also zu einer Suche nach einer überdeckenden Menge von Primimplikanten. Obwohl dieses Problem NP-vollständig ist (siehe S. 776), gibt es ein gut brauchbares Verfahren von W. Quine und E. McCluskey, für das wir auf die weiterführende Literatur verweisen.

### 5.3.4 Spezielle Schaltglieder

Eine wichtige zusammengesetzte Schaltung ist der *Multiplexer*, oder *MUX-Glied*. Wenn  $c = 1$  ist, dann ist  $z = x$ , ansonsten ist  $z = y$ . Diese Schaltung implementiert ein *if-then-else*, denn  $z = if\ c\ then\ x\ else\ y$ . Ein Multiplexer ist häufiger Bestandteil in digitalen Schaltungen. Man stellt derartige Bausteine oft nur als Blockschaltbild dar:

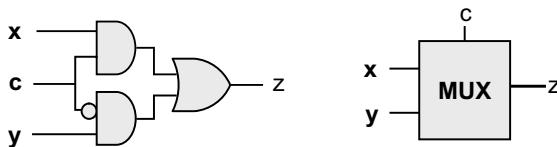


Abb. 5.29: Multiplexer: Schaltung und Schaltsymbol

Multiplexer sind *universelle Schaltglieder*, wie man aus den folgenden Tatsachen erkennt. Zunächst versteht man unter einem  $2^n$ -Kanal Multiplexer einen solchen, der je nach Kombination seiner  $n$  Kontrolleingänge einen der  $2^n$  Dateneingänge durchschaltet. Man kann beliebige Multiplexer aus einfachen MUX-Gliedern zusammenbauen, wie die folgende Abbildung für den Fall  $n=2$  zeigt. Repräsentieren die Eingänge  $c_1 c_0$  eine Binärzahl  $k < 4$ , also  $k = (c_1 c_0)_2$ , so wird der  $k$ -te Eingang,  $x_k$ , nach  $z$  durchgeleitet.

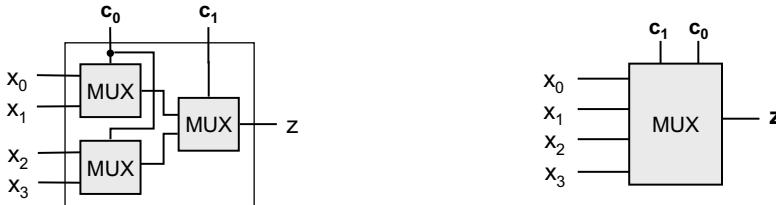


Abb. 5.30: Vierkanal-Multiplexer

Jede Schaltfunktion  $f: \{0, 1\}^n \rightarrow \{0, 1\}$  lässt sich auch unmittelbar mit einem  $2^n$ -Kanal Multiplexer realisieren: Sei  $0 \leq k < 2^n$  eine Zahl mit der Binärdarstellung  $k = (c_{n-1}, \dots, c_0)_2$ . Man verbindet den Eingang  $x_k$  mit 1, falls  $f(c_{n-1}, \dots, c_0) = 1$  gewünscht ist, und sonst mit 0. Legt man nun  $c_{n-1}, \dots, c_0$  an den Kontrolleingängen an, so berechnet die Schaltung den Wert  $z = f(c_{n-1}, \dots, c_0)$ . MUX ist also ein universelles Schaltelement.

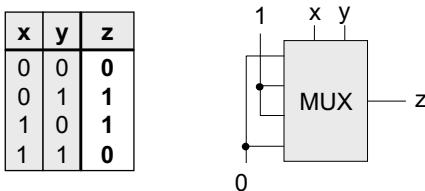


Abb. 5.31: MUX als universelles Schaltelement

### 5.3.5 Gatter mit mehreren Ausgängen

Gatter mit mehreren Ausgängen realisieren Funktionen  $f: \{0, 1\}^m \rightarrow \{0, 1\}^n$ . Ein entsprechendes Schaltglied hat also  $m$  Eingänge und  $n$  Ausgänge. Jedes solche Schaltglied kann aus  $n$  Schaltgliedern mit je  $m$  Eingängen und einem Ausgang aufgebaut werden. Mathematisch stellen wir  $f$  als Tupel von  $n$  Schaltfunktionen dar:  $f = (f_1, \dots, f_n)$ . Technisch kann man jede Komponente getrennt aufbauen und die Eingänge zusammenfassen. In Einzelfällen lassen sich einige Schaltglieder auch gemeinsam nutzen.

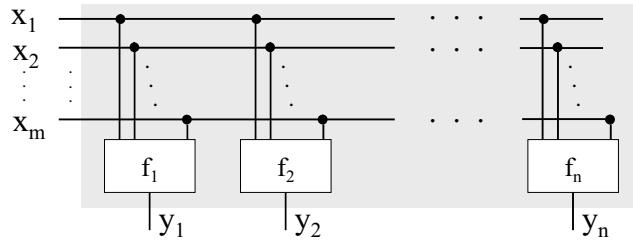


Abb. 5.32: Zusammensetzung eines Schaltgliedes aus Schaltfunktionen

### 5.3.6 Codierer und Decodierer

Eng verwandt mit dem Multiplexer, und ähnlich nützlich sind die Schaltungen eines Codierers und eines Decodierers. Ein *Decodierer* hat  $n$  Eingänge und  $2^n$  viele Ausgänge. Die Folge von 0-en und 1-en am Eingang wird als Binärzahl  $k$  interpretiert, dann wird die  $k$ -te Ausgangsleitung auf 1 gesetzt, alle anderen auf 0. Decodierer werden beispielsweise eingesetzt, um eine bestimmte Speicherzelle zu adressieren. Am Eingang wird die binäre Adresse einer Speicherzelle angelegt. Nur die Ausgangsleitung, die zu der entsprechenden Zelle führt, wird dadurch aktiviert, alle andere bleiben 0.

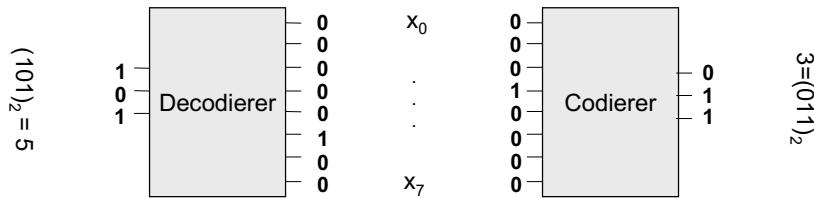


Abb. 5.33: 3-8-Decodierer und 8-3-Codierer

Die Funktionsweise eines *Codierers* ist genau umgekehrt. Er besitzt  $2^n$  viele Eingänge und  $n$  Ausgänge. Wenn am  $k$ -ten Eingang der Wert 1 liegt und an allen anderen Eingängen der Wert 0, dann wird die Zahl  $k$  an den Ausgängen  $z_0, \dots, z_{n-1}$  binär dargestellt. Für alle anderen Eingabewerte ist das Ergebnis unspezifiziert. Dies gibt dem Implementierer die Freiheit, zu bestimmen, was mit denjenigen Inputs geschehen soll, bei denen mehrere 1-en auftauchen. So gibt es zum Beispiel einen *Prioritäts-Codierer*, der immer die Nummer der ersten Leitung codiert, die das Signal 1 trägt. Eine andere Strategie ist, die Werte so zu wählen, dass man mit möglichst wenigen Schaltelementen auskommt.

Wir illustrieren dies am Beispiel des 4-2-Codierers. Stellen wir sein KV-Diagramm auf, so kann man an allen bis auf 4 Positionen einen beliebigen Wert wählen. Das KV-Diagramm für die niedrigstwertige Position  $z_0$  ist dann wie in dem linken KV-Diagramm dargestellt:  $f(0, \dots, 0, x_k, 0, \dots, 0) = k \bmod 2$ . Alle freien Einträge können beliebig gewählt werden. Wir

können sämtliche 1-en in einem gemeinsamen  $2 \times 2$ -Quadrat unterbringen, so dass sich die boolesche Funktion  $x_0'x_2'$  ergibt.

		$x_2$					
		00	01	11	10		
		$x_0$	$x_1$	$x_2$	$x_3$		
		00	1		0		
		10	0				
		11					
		01	1				
				$x_2$			
00	01	11	10	00	01	11	10
10	00	01	11	00	01	00	00
11	01	10	11	01	00	00	00
01	11	11	01	01	11	00	00

### 5.3.7 Addierer

Als weiteres Beispiel einer Schaltung mit mehreren Ausgängen betrachten wir den *Halbaddierer*, eine Schaltung, die zwei Binärziffern  $x$  und  $y$  addiert. An den Eingängen  $x$  und  $y$  liegen die zu addierenden Binärziffern, am Ausgang  $s$  entsteht das Summenbit und am Ausgang  $c$  der Übertrag (engl. *carry*). Aufbau, Tabelle, Schaltzeichen und definierende Gleichungen zeigt die folgende Abbildung:

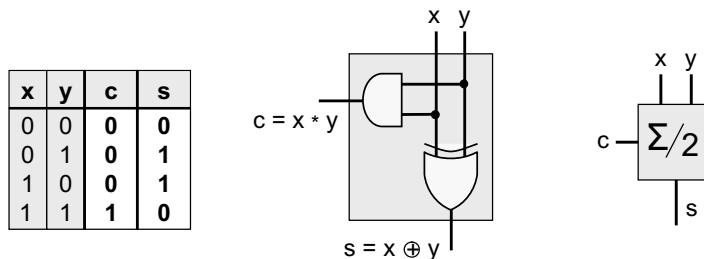


Abb. 5.34: Halbaddierer: Schaltfunktion, Implementierung und Blocksymbol

Ein *Volladdierer* soll ebenfalls zwei Binärziffern  $x$  und  $y$  addieren können. Er muss aber ggf. noch einen von einer niedrigeren Zifferposition kommenden Übertrag  $ci$  (*carry-in*) berücksichtigen. Das Ergebnis ist die (letzte) Ziffer der Summe sowie ein Übertrag  $co$  (*carry-out*). Er lässt sich aus zwei Halbaddierern und einem OR-Glied aufbauen:

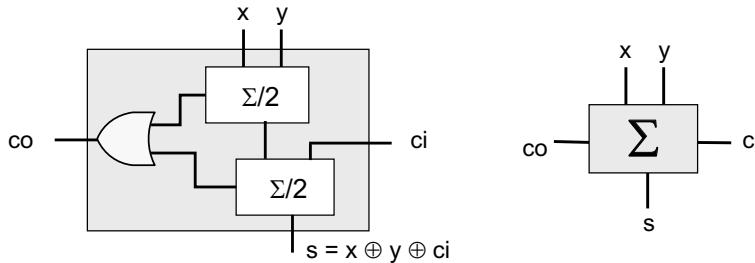


Abb. 5.35: Volladdierer: Schaltplan und Schaltzeichen

Mit einer Kaskade von  $n$  Volladdierern und einem Halbaddierer kann man ein Addierwerk zusammensetzen, um zwei  $(n+1)$ -stellige Binärzahlen  $x_n \dots x_0$  und  $y_n \dots y_0$  zu addieren. Jeder Ein-Bit-Addierer ist für eine Zifferposition verantwortlich. Der  $co$ -Ausgang jedes Addierers wird mit dem  $ci$ -Eingang des nächsten verbunden. Der  $co$ -Ausgang der höchsten Bit-Stelle stellt das *Carry Flag C* dar.

Die gleiche Schaltung funktioniert auch für Zweierkomplementzahlen. Hierbei wird die höchste Bitstelle als Vorzeichenbit interpretiert. Eine Bereichsüberschreitung ist daran erkennbar, dass die  $co$ -Ausgänge der höchsten und der zweithöchsten Bitstelle verschieden sind. Wenn wir diese mit  $\oplus$  verknüpfen erhalten wir folglich das *Overflow Bit O*.

Für die Subtraktion machen wir uns zunutze, dass Zweierkomplementzahlen  $X$  und  $Y$  subtrahiert werden, indem man zu  $X$  das Zweierkomplement von  $Y$  addiert und zum Schluss eine 1 addiert. Wir erreichen dies, indem wir jedem  $Y$ -Eingang ein mit dem Signal  $Neg$  verbundenes XOR-Gatter vorschalten. Falls  $Neg=0$ , hat dies wegen  $Y = 0 \oplus Y$  keinen Einfluss, falls  $Neg=1$  wird wegen  $\bar{Y} = 1 \oplus Y$  das bitweise Komplement von  $Y$  zu  $X$  addiert. Schließlich wird eine 1 addiert, weil der  $Neg$ -Eingang auch als Carry des niedrigsten Addierers eingespeist wird.

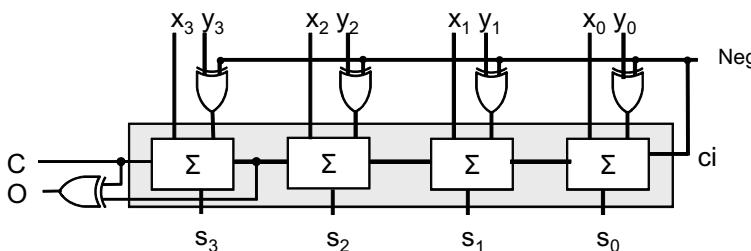


Abb. 5.36: Kaskade von 4 Addierern zur Addition und Subtraktion

Der Carry-Ausgang des Höchstwertigen (linken) Addierers  $C$  ergibt den Übertrag und ein XOR der beiden höchstwertigen Carry-Ausgänge das Overflow Bit  $O$ .

### 5.3.8 Logik-Gitter

Umfangreiche Schaltkreise werden nicht individuell aus einzelnen Schaltelementen zusammengesetzt. Man verwendet Standardmodule, die auf einfache Weise angepasst werden können, um die jeweils gewünschte Schaltung zu realisieren. Ein *Logik-Gitter* (engl. *logic array*) ist ein zweidimensionales Leitungsgitter, dessen Kreuzungspunkte jeweils von einem Gitterbaustein gebildet werden. Man kommt mit 4 verschiedenen Gitterbausteinen aus, einem *Identer*, einem *Multiplizierer*, einem *Negat-Multiplizierer* und einem *Addierer*. Dies sind jeweils einfache Bausteine mit zwei Eingängen und zwei Ausgängen. In seiner Position im Gitter erhält ein solcher Baustein einen Input  $x$  von seinem linken Nachbarn und einen zweiten Input  $y$  von seinem oberen Nachbarn. Die Ausgänge  $r$  und  $u$  führen entsprechend zu dem rechten bzw. unteren Nachbarn.

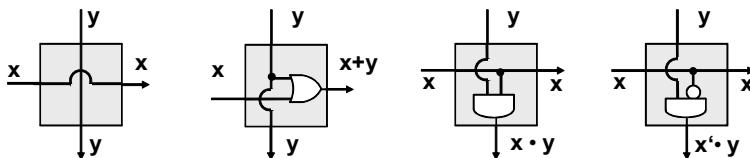


Abb. 5.37: Identer, Addierer, Multiplizierer und Negat-Multiplizierer

Multiplizierer und Negat-Multiplizierer leiten den Input, den sie von links erhalten, unverändert nach rechts weiter. Nach unten jedoch geben sie den verknüpften Wert  $x \cdot y$  bzw.  $x' \cdot y$  aus. Der Addierer reicht den von oben erhaltenen Wert nach unten durch, während er nach rechts die Summe seiner Eingabewerte ausgibt. Der Identer leitet sowohl horizontal als auch vertikal seinen Input unverändert weiter.

In einem Gitter, das an den Kreuzungspunkten nur drei dieser Bausteine, nämlich Identer, Multiplizierer oder Negat-Multiplizierer hat, legen wir an den oberen Spalteneingängen jeweils eine 1 an und an den linken Zeileneingängen die Werte  $x_1, x_2, \dots, x_n$ . An den unteren Ausgängen der Spalten entsteht dann jeweils ein Monom. Eine Variable  $x_i$  kommt im Monom der  $k$ -ten Spalte genau dann komplementiert vor, wenn am Kreuzungspunkt der  $i$ -ten Zeile mit der  $k$ -ten Spalte ein Negat-Multiplizierer sitzt, unkomplementiert, wenn es sich um einen Multiplizierer handelt.

Ein Identer bewirkt, dass die entsprechende Variable im Monom nicht erscheint. Dies geht aus dem oberen Teil der Abbildung hervor. Diesen Teil nennt man auch die **UND-Ebene**.

Im unteren Teil des Gitters, der so genannten **ODER-Ebene**, werden die an kommenden Monome addiert und der Wert nach rechts ausgegeben. Jede Zeile dieses Teiles enthält nur Identer oder Addierer und summiert auf diese Weise nur die benötigten Monome. Da man jede gewünschte Schaltfunktion durch Summe von Monomen darstellen kann, bieten Logik-Gitter ein einfaches Schema, um beliebige Schaltfunktionen zu realisieren. Allgemeiner hat man nicht nur eine, sondern mehrere Zeilen in der ODER-Ebene, so dass man gleichzeitig mehrere boolesche Terme realisieren kann.

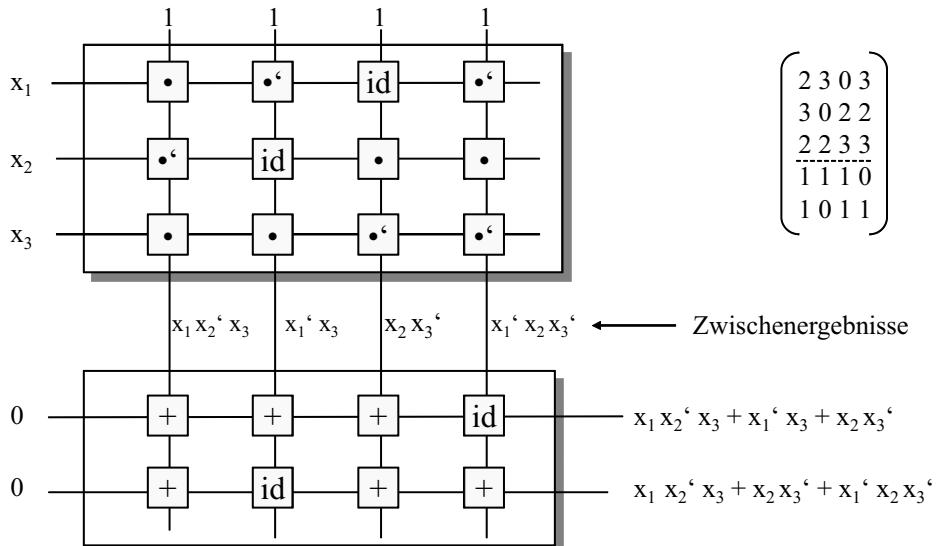


Abb. 5.38: Ein Logik-Gitter für zwei Schaltfunktionen und die zugehörige Matrix

In der obigen Figur sind die Bauteile Identer, Addierer, Multiplizierer und Negat-Multiplizierer mit den Symbolen id, +, • und •' bezeichnet – üblicherweise nummeriert man die Bauteile in dieser Reihenfolge einfach von 0 bis 3 durch. Dann kann ein Logik-Gitter einfach durch eine  $(n+m) \times k$  Matrix spezifiziert werden, wobei  $n$  die Anzahl der Variablen bestimmt,  $m$  die Anzahl der verschiedenen booleschen Terme und  $k$  die Anzahl der benötigten Monome. In den ersten  $n$  Zeilen der Matrix kommen nur die Werte 0, 2, 3 vor, in den letzten  $m$  Zeilen nur 0 oder 1. Das vorige Beispiel wird also durch die dargestellte  $(3+2) \times 4$ -Matrix beschrieben.

### 5.3.9 Programmierbare Gitterbausteine

Zu einem universellen Werkzeug wird ein Logik-Gitter erst, wenn wir die Gitterbausteine nicht fest an den Kreuzungspunkten des Gitters platzieren, sondern stattdessen einen programmierbaren Gitterbaustein verwenden, der sich, abhängig von einem externen Input, wie ein beliebiger Gitterbaustein verhalten kann. Für die Spezifikation, um welchen der 4 Gitterbausteine es sich handeln soll, benötigt man 2 zusätzliche Bit  $(b_1, b_0)$ , so dass der universelle Gitterbaustein vier Eingänge  $(b_1, b_0, x, y)$  und zwei Ausgänge  $(r, u)$  besitzt. Über die Eingänge  $(b_1, b_0)$  kann er verändert (programmiert) werden, was den Namen *PLA (programmable logic array)* erklärt. Für die Schaltfunktion dieses universellen Gitterbausteins liest man unmittelbar aus der Tabelle ab:

$$r = x + b_1'b_0y \quad \text{sowie} \quad u = b_1'y + b_1b_0'xy + b_1b_0x'y.$$

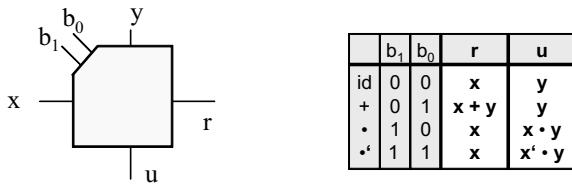


Abb. 5.39: Programmierbarer Gitterbaustein

## 5.4 CMOS Schaltungen und VLSI Design

Die Boolesche Algebra beginnt mit Elementarschaltern und konstruiert daraus durch Negation, Serien- und Parallelschaltung beliebige Schaltkreise. Als Elementarschalter werden in der Praxis *Transistoren* eingesetzt. Ein Transistor hat einen Eingang (*Source*), einen Ausgang (*Drain*) und einen Steuerungseingang (*Gate*). Legt man eine Spannung zwischen Source und Drain, so fließt nur dann Strom, falls auch eine Steuerspannung am Gate anliegt.

Wir beschränken uns hier auf die modernere und stromsparende CMOS-Technik (*complementary metal oxide semiconductor*), bei der sowohl p-MOS als auch n-MOS Feldeffekttransistoren (MOSFET) in zueinander komplementären Schaltkreisen eingesetzt werden. CMOS-Schaltungen verbrauchen im Gegensatz zu den älteren Schaltungen mit Bipolartransistoren nur wenig Strom was auch einen geringeren Aufwand zur Kühlung der Chips bedingt.

Die Schaltung wird mit einer positiven Versorgungsspannung  $V_{CC}$  (*voltage of the common collector*) betrieben, für die man z.B. 2,9V (oder 5V) wählen kann. Die logischen Werte 0 und 1 entsprechen dann idealerweise den Spannungspegeln 0 V und 2,9 V. In der Praxis kann man aber den Bereich 0-0,5 V als logisch „0“ und 2,4-2,9 V als logisch „1“ interpretieren.

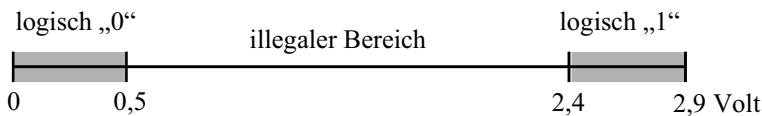


Abb. 5.40: Analoge und logische Werte

Wir hatten p-MOSFETs und n-MOSFETs als ideale Schalter eingeführt. In der Praxis unterscheiden sich ihre Schaltcharakteristiken je nachdem ob sie an  $V_{CC}=1$  oder an  $Gnd=0$  (*ground=Erde*) angeschlossen sind: p-MOS-Transistoren lassen das Signal 1 fast ungedämpft durch, während das 0-Signal gedämpft wird, bei n-MOS-Transistoren ist es genau umgekehrt. Daher bestehen CMOS-Schaltungen immer aus zwei Teilschaltungen - einer sogenannten *pull-up* Schaltung, die für das Ausgangssignal 1 zuständig ist und nur aus p-MOS Transistoren besteht sowie einer *pull-down* Schaltung aus n-MOS-Transistoren, die das Ausgangssignal 0 produziert.

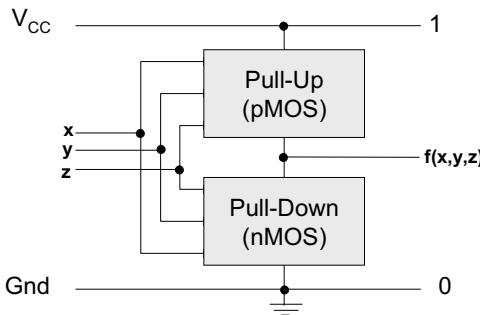


Abb. 5.41: Aufbau einer CMOS-Schaltung zur Realisierung einer booleschen Funktion  $f(x,y,z)$

Selbstverständlich muss dafür gesorgt werden, dass der Ausgang nie gleichzeitig mit 1 ( $V_{CC}$ ) und mit 0 ( $Gnd$ ) verbunden sein kann. Dies hätte einen Kurzschluss zur Folge! Aus diesem Grund sind pull-up und pull-down Schaltung immer komplementär zueinander aufgebaut: Einer Parallelschaltung im pull-up Teil entspricht eine Serienschaltung im pull-down Kreis. Dies erklärt auch den Namen CMOS (complementary MOS).

### 5.4.1 Logikgatter in CMOS-Technik

Die einfachste CMOS-Schaltung ist der in der folgenden Figur gezeigte *CMOS-Inverter* der nur aus einem n-MOS und einem p-MOS besteht. Ist der Eingang  $x = 1$ , so sperrt der p-MOS Transistor, denn dessen Source und Gate liegen auf dem gleichen Spannungsniveau. Gleichzeitig ist am n-MOS-Transistor die Spannung zwischen Gate und Source maximal, so dass dieser öffnet und am Ausgang  $z$  das Spannungspiegel  $Gnd = 0$  liegt.

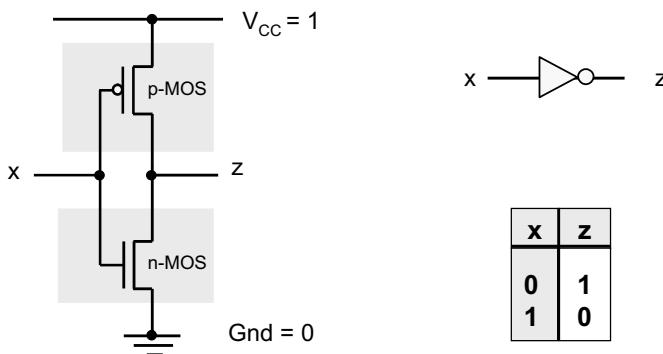


Abb. 5.42: CMOS-Inverter, Gattersymbol und Schalttabelle

Genau umgekehrt sind die Verhältnisse im Fall  $x=0$ . Jetzt liegen Gate und Source des n-MOS auf gleichem Niveau, so dass dieser sperrt. Dagegen ist die Spannung zwischen Gate und Source des p-MOS maximal, so dass dieser öffnet und dem Ausgang  $z$  das gleiche Span-

nungsniveau beschert wie dem Source des p-MOS also logisch 1. Insgesamt hat also  $z$  immer den entgegengesetzten logischen Wert von  $x$ , weshalb die gezeigte Schaltung der Negation entspricht.

Da im Allgemeinen p-MOS Transistoren nur im pull-up Teil verwendet werden und n-MOS nur im pull-down Teil, ist in diesen Fällen eine Spannung zwischen Gate und Source eines p-MOS gleichbedeutend mit dem Signal 0 am Gatter. Das heißt, dass in einer CMOS-Schaltung ein p-MOS Transistor leitend ist, wenn logisch 0 am Gatter liegt und analog ein n-MOS Transistor, wenn logisch 1 am Gatter liegt. Dies erklärt den Kreis im Schaltbild des p-MOS Transistors.

Vor diesem Hintergrund sind die folgenden Schaltungen auch leichter zu verstehen. Die erste Abbildung zeigt die CMOS-Schaltung für NOR, das Gattersymbol und die Wertetabelle. Man sieht wie die Serienschaltung im pull-up Teil einer Parallelschaltung im pull-down-Kreis entspricht. Nur wenn  $x = 0$  und  $y = 0$  sind, ist der Ausgang  $z$  mit 1 ( $V_{CC}$ ) verbunden. Gleichzeitig sind beide n-MOS Transistoren gesperrt. Falls  $x = 1$  oder  $y = 1$  ist die Verbindung von  $z$  zu 0 (Gnd) hergestellt.

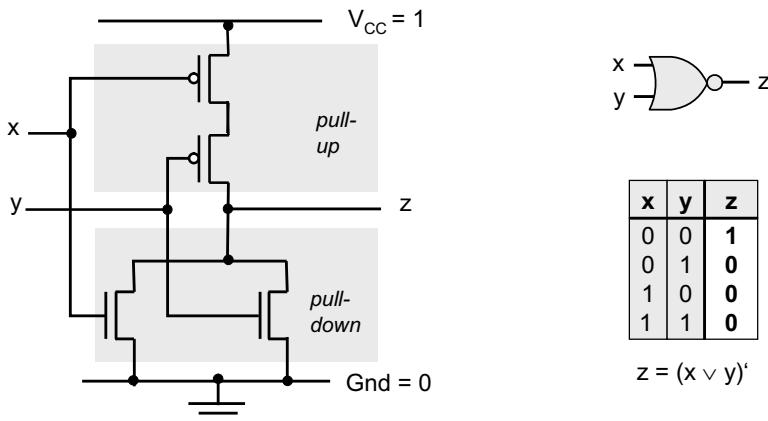


Abb. 5.43: CMOS Schaltung für NOR, Gattersymbol und Schalttabelle

Die CMOS-Schaltung für NAND ist dual zur Schaltung für NOR. Die Dualität drückt sich darin aus, dass das pull-up Netz der einen dem pull-down Netz der anderen Schaltung entspricht, wobei selbstverständlich im pull-up Teil stets nur p-MOS und im pull-down Teil nur n-MOS Transistoren verwendet werden.

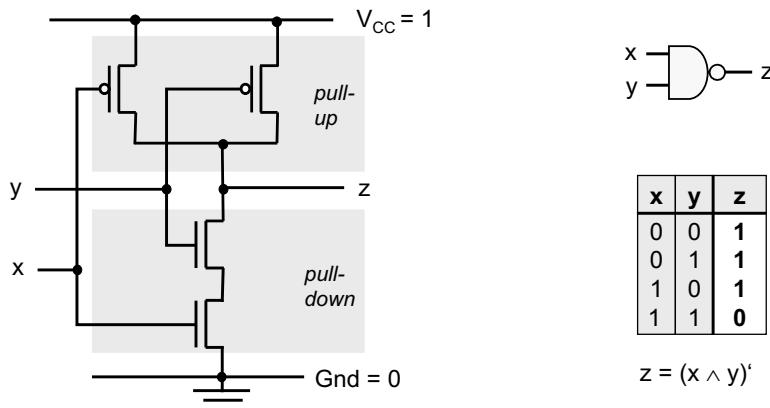


Abb. 5.44: CMOS Schaltung für NAND, Gattersymbol und Schalttabelle

Die Schaltglieder für AND und OR werden durch nachgelagerte Inverter realisiert, wie in der folgenden Figur am Beispiel von AND gezeigt wird.

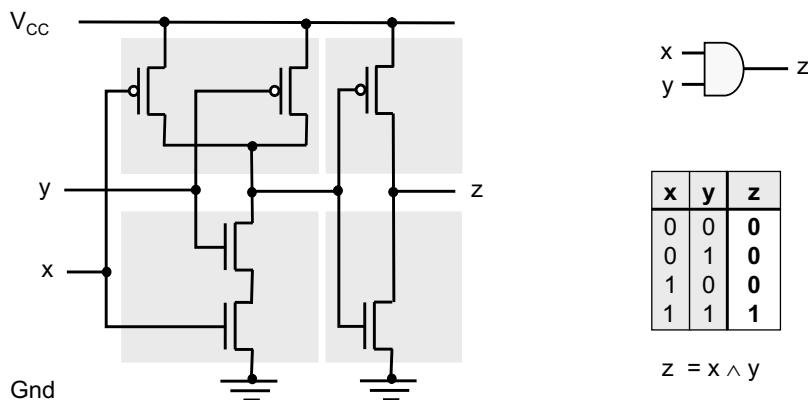


Abb. 5.45: CMOS-Implementierung von AND

## 5.4.2 CMOS-Entwurf

Es ist nun einfach festzustellen, wie eine beliebige Schaltung in CMOS entworfen werden kann. Sei  $f(x_1, \dots, x_n)$  der Boolesche Term. Da die pull-down Schaltung genau dann das Ergebnis mit Gnd verbinden soll, wenn  $f(x_1, \dots, x_n) = 0$  ist, negieren wir den Ausgangsterm zu  $f(x_1, \dots, x_n)'$ , vereinfachen diesen, und interpretieren dann jedes Literal als n-MOS Transistor, jedes + als Parallelschaltung, jedes • als Serienschaltung.

Im pull-up Teil leitet ein p-MOS-Transistor genau dann, wenn sein Gate 0 ist. Daher negieren wir alle Literale von  $f(x_1, \dots, x_n)$ , und bauen die Schaltung, die  $f(x_1', \dots, x_n')$  entspricht. Es folgt, dass die pull-up Schaltung und die pull-down Schaltung zueinander dual sind.

Zur Illustration betrachten wir den Term  $f(x, y, z) = (x' \bullet y) + z'$ . Für die pull-up-Schaltung invertieren wir die Literale und erhalten  $f(x', y', z') = x \bullet y' + z$ , was einer Parallelschaltung von  $z$  mit der Serienschaltung von  $x$  und  $y'$  entspricht. Für die pull-down-Schaltung vereinfachen wir  $f(x, y, z)' = ((x' \bullet y) + z')'$  zu  $(x + y') \bullet z$ , erhalten also eine Reihenschaltung von  $z$  mit der Parallelschaltung von  $x$  und  $y'$ . Eigentlich muss man jetzt noch die Negation  $y'$  von  $y$  bereitstellen. In der Praxis hat man zu jeder Eingangsvariablen oft schon an anderer Stelle auch deren Negation „vorrätig“, so dass man diese einfach abgreifen kann.

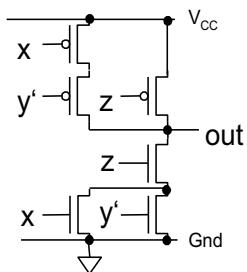


Abb. 5.46: CMOS-Entwurf - Beispiel

### 5.4.3 Entwurf von CMOS Chips

Die gezeigten CMOS Schaltungen erwecken den Eindruck, als müsse man alle Logikgatter bzw. sogar die Transistoren einzeln bauen und diese dann entsprechend verbinden. In Wirklichkeit werden ganze Schaltungen nach dem logischen und dem CMOS-Entwurf anhand von sogenannten Zellbibliotheken entworfen. Es beginnt mit dem Entwurf der Schaltung in einer modularen Hardwarebeschreibungssprache, z.B. *VHDL* oder *Verilog* oder *SystemC*. In solchen Sprachen kann man, aufgrund ihrer Modularität, beliebig komplexe Schaltungen spezifizieren, simulieren und testen, bevor man die teure und aufwendige Herstellung des Chips in Angriff nimmt. Eine vollständige Verilog-Implementierung einer CPU ist in dem Buch von K. Stroetmann: *Computerarchitektur* (s. Literaturverzeichnis) angegeben und genau erklärt.

Die Beschreibung eines Volladdierers in Verilog könnte folgendermaßen beginnen:

```

module fulladder(input a,b,c, output s, cout);
    sum s1(a,b,c,s);
    carry c1(a,b,c,cout);
endmodule

```

Das Modul *fulladder* bezieht sich auf zwei Untermodulen, *sum* und *carry*, von denen wir das letztere schon direkt boolesch beschreiben können.

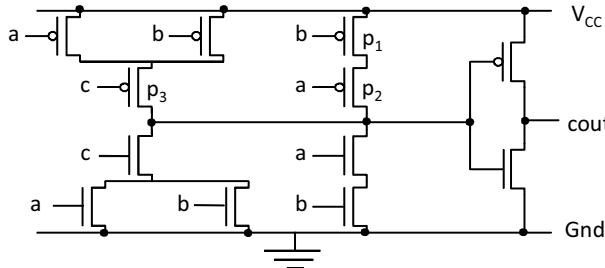
```

module carry(input a,b,c, output cout)
    assign cout = (a&b) | (a&c) | (b&c);
endmodule

```

Die *carry*-Schaltung berechnet also einfach den logischen Ausdruck  $a \bullet b + a \bullet c + b \bullet c$ , der zu  $a \bullet b + c \bullet (a + b)$  vereinfacht werden kann. Die Summe wird analog als  $a \oplus b \oplus c$  spezifiziert.

Aus der Verilog-Beschreibung kann automatisch die *Netzliste*, d.h. die Teileliste mit ihren Verbindungen, somit auch der Schaltplan der CMOS-Schaltung gewonnen werden.



*Abb. 5.47:* CMOS-Schaltung zur Carry-Berechnung

Rechts in der Abbildung erkennt man die typische Inverter-Schaltung. Links daneben wird zunächst das Komplement von  $a \cdot b + c \cdot (a + b)$  berechnet, das dann im Inverter wieder invertiert wird.

Diese Schaltung sieht auf den ersten Blick außergewöhnlich aus, da die Komplementarität von pull-up und pull-down Teil nicht unmittelbar ersichtlich ist. Eigentlich müssten im pull-up Teil die in Serie geschalteten p-MOS Transistoren  $p_1$  und  $p_2$  mit  $p_3$  parallel geschaltet sein. Dies kann man aber offensichtlich zu der gezeigten Schaltung vereinfachen, die den Vorteil hat, dass nirgends mehr als 2 Transistoren in Reihe geschaltet sind.

Wir wollen  $a \cdot b + c \cdot (a+b)$  implementieren. Weil wir uns die Invertierung jedes der Eingangssignale sparen wollen, entschließen wir uns, das Komplement  $f(a,b,c) = [a \cdot b + c \cdot (a+b)]'$  zu implementieren und dieses anschließend zu invertieren.

Für die Pull-up Schaltung erhalten wir:

$$\begin{aligned} f(a', b', c') &= [a' \bullet b' + c' \bullet (a' + b')] J' = (a+b) \bullet (c+a \bullet b) = (a+b) \bullet c + (a+b) \bullet (a \bullet b) = (a+b) \bullet c + (a \bullet b) \\ &= (a \bullet b) + c \bullet (a+b). \end{aligned}$$

Für die Pull-down Schaltung erhalten wir:  $f(a,b,c)' = [a \cdot b + c \cdot (a+b)]'' = a \cdot b + c \cdot (a+b)$  also die identische Schaltung. Damit haben wir gezeigt, dass der Ausgangsterm *selbstdual*, also identisch zu seinem dualen ist:

$$f(x_1, \dots, x_n)' = f(x_1', \dots, x_n') .$$

Nach der Erstellung der *Netzliste*, d.h. der Liste aller Schaltglieder mit ihren Verbindungen, werden die Bestandteile der Schaltung in Zellbibliotheken gesucht, die das layout der p- und n-dotierte Bereiche, Gates, Isolierung Kontakte etc. bestimmen, aus denen dann die Masken für das Belichten, Ätzen und dotieren bestimmt werden.

## 5.4.4 VLSI-Werkzeuge

Die Konstruktion komplexer CMOS-Chips kann heute nur mit umfangreicher Werkzeugunterstützung gelingen. Der gezeigte Bildschirmabzug zeigt das freie VLSI-Entwurfswerkzeug *Electric* von *Static Free Software*. Mit diesem System haben wir eine CMOS-Schaltung,

$y = (a \cdot b + c)$ ', graphisch entworfen und im linken Bild das zugehörige Chip-Layout zusammengestellt. Rechts sieht man die CMOS-Schaltung und darunter die genannte Netzliste.

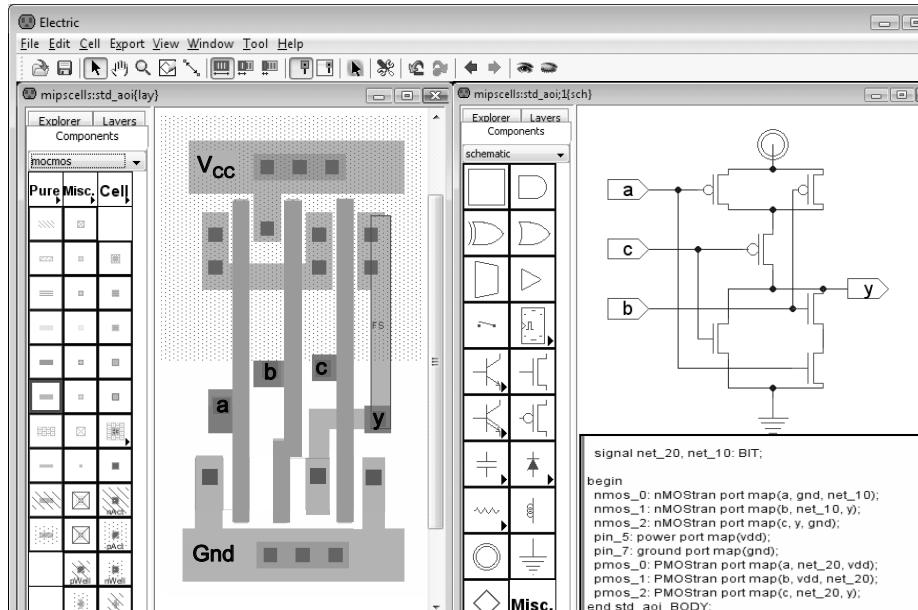


Abb. 5.48: Electric - ein VLSI-Design-Tool

Die aufgeführten Bauteile *nmos\_0*, *nmos\_1* und *nmos\_2* bzw. *pmos\_0*, *pmos\_1* und *pmos\_2* sind die n-MOS- bzw. p-MOS-Transistoren; *pin\_5* und *pin\_7* sind die Anschlüsse für *vdd* (=V<sub>CC</sub>) und *gnd*. Zusätzlich benötigt man noch zwei Verbindungen, die weder an *a*, *b*, *c*, *y*, *gnd* oder *vdd* angeschlossen sind. Es handelt sich um die Verbindung (*net\_20*) der durch *a* und *b* angesteuerten n-MOS und die Verbindung (*net\_10*) vom Source des von *c* angesteuerten p-MOS zu den Drains der beiden anderen p-MOS.

Diese Netzliste ist eine eindeutige textuelle Charakterisierung der gezeigten CMOS-Schaltung. Um sie in Silikon zu realisieren, müssen wir das physikalische Layout der Transistoren, der p- und n-dotierten Bereiche, der Gates und der elektrischen Verbindungen bestimmen. Ein Vorschlag dafür ist in dem linken Fenster dargestellt.

Man erkennt oben und unten zwei große Blöcke, die metallische Bereiche darstellen, an denen die Versorgungsspannung - oben *V<sub>CC</sub>* und unten *Gnd* - angeschlossen werden. Die drei vertikal verlaufenden, mit *a*, *b* und *c* kontaktierten Balken stellen die Gates der 6 Transistoren dar - der 3 p-MOS Transistoren im oberen (pull-up) Bereich und der 3 n-MOS Transistoren im unteren (pull-down) Bereich. Der *u*-förmige Balken sowie der einem Fragezeichen ähnelnde Balken am rechten Rand mit dem Kontakt für *y* sind metallische Verbindungen. Die leichte Schattierung der oberen Hälfte des Layouts zeigt die schwach n-dotierte Wanne an, in die der p-MOS-Teil eingebaut wird.

Für einen Strom von  $V_{CC}$  zu  $y$  gibt es nur den Weg über den  $u$ -förmigen Bereich. Dieser stellt damit den Drain für die beiden linken p-MOS dar. Ihre gemeinsame Source ist die Ausbuchung des  $V_{CC}$ -Anschlusses. Es reicht also, wenn  $a$  oder  $b$  schaltet. Von dem  $u$ -förmigen Bereich kann der Strom schließlich zu  $y$  gelangen, sofern der p-MOS  $c$  schaltet. Weil es sich um p-MOS handelt heißt dies:  $(a' + b') \cdot c'$ , also  $(a \cdot b + c)'$ .

Das Layout des pull-down Teils ist analog erklärbar. Von  $Gnd$  zu  $y$  gibt es zwei Wege - entweder direkt über den mit  $c$  angesteuerten n-MOS oder über  $a$  und  $b$  zu  $y$ .

Bei jeder Manipulation dieses Layouts überprüft das System stets alle notwendigen Entwurfsrestriktionen - den Abstand der leitenden Teile, die Kapazität der (unbeabsichtigt) entstehenden Kondensatoren, etc. Wenn das Design fertig ist kann es in einer Zellbibliothek gespeichert und später wiederverwendet werden.

## 5.5 Sequentielle Logik

Boolesche Schaltkreise ohne Rückkopplung und ohne Zeitsignal kann man in erster Näherung so behandeln, als würden sie ihr Ergebnis instantan, also ohne zeitliche Verzögerung produzieren. Mit der Uhr und mit speichernden Gliedern wie Flip-Flops spielt der Aspekt der zeitlichen Aufeinanderfolge von Ereignissen eine wichtige Rolle. Insbesondere können Flip-Flop-Schaltungen einen Zustand speichern und das Ergebnis einer Schaltung kann ein zustandsabhängiger Wert sein, verbunden mit einer Änderung des Zustandes. Solche zustandsabhängigen Systeme sind die Regel in einem realen Chip, sei es ein Speicherchip, eine Digitaluhr oder die CPU eines Rechners.

### 5.5.1 Gatterlaufzeiten

In der Praxis schaltet ein Gatter nicht augenblicklich, sondern es benötigt eine gewisse Zeit. Dies ist der Tatsache geschuldet ist, dass bei einem MOS-Transistor das Gate und das gegenüberliegende Substrat wie zwei Platten eines Kondensators, mit der Isolationsschicht als Dielektrikum wirken. Jeder Schaltvorgang bringt Elektronen auf das Gate, bzw leitet sie vom Gate weg. Nur dadurch kommt ein Stromfluss zustande, was auch die geringe Leistungsaufnahme von MOS-Schaltungen begründet. Das Laden und Entladen eines Kondensators benötigt nur wenig Zeit, so dass die Gatterlaufzeiten bei CMOS-Schaltungen weniger als 1 nsec betragen.

Vor diesem Hintergrund hat die Tatsache, dass sich jede boolesche Funktion  $f$  in disjunktiver Normalform, also als Summe von Monomen, darstellen lässt, die praktische Konsequenz, dass die Berechnung von  $f$  in drei Stufen vor sich gehen kann: Zunächst werden die Negationen der EingabevARIABLEN berechnet, danach die relevanten Monome und im dritten Schritt deren Summe. Gehen wir vereinfachend davon aus, dass in der Praxis jedes Logikgatter eine Schaltzeit  $\delta$  benötigt, so ist die Berechnung einer in Normalform dargestellten booleschen Funktion nach der Zeit  $3\delta$  beendet und zwar unabhängig von der Anzahl der Variablen oder der Komplexität der Funktion.

Im Falle der Additionsschaltung in Abb. 5.36, die nicht in Normalform vorliegt, ist die Laufzeit abhängig von der Anzahl der Summationsglieder, weil in jeder Ziffernposition die Addition erst korrekt durchgeführt werden kann, wenn aus der jeweils niedrigeren Bitposition das Carry-Bit durchgereicht wurde. Weil sich die Übertragbits von rechts nach links wie eine sich kräuselnde Welle ausbreiten, nennt man den gezeigten Addierer auch „*ripple-carry adder*“.

Theoretisch könnte man eine Additionsschaltung für zwei  $n$ -Bit Zahlen auch auf Basis einer entsprechend großen Schalttabelle als Boolesche Schaltung mit  $2n+1$  Eingängen und  $n+1$  Ausgängen entwerfen. Dann käme man wieder auf die oben diskutierten Schaltzeit  $3\delta$ . Jede der Schalttabellen für die  $n+1$  Summenbits hätte dann aber  $2^{2n+1}$  Zeilen und in dieser Größenordnung läge auch die Anzahl der benötigten Schaltglieder, so dass dieses Verfahren in der Praxis für  $n=16$  oder  $n=32$  nicht möglich ist. Ein Kompromiss besteht darin, 4-Bit-Addierer als boolesche Funktionen in Normalform zu realisieren und dann diese 4-Bit-Addierer zu einem entsprechend größeren Addierer zusammensetzen.

Ein *carry-lookahead Addierer* berechnet für alle Bitpositionen  $i$  gleichzeitig das Carry  $c_i$  mit Hilfe einer booleschen Funktion  $c_i(x_{i-1}, \dots, x_0, y_{i-1}, \dots, y_0, cin)$ , so dass alle Summationen ebenfalls gleichzeitig stattfinden können.

Einen  $2n$ -Bit Addierer kann man auch aus drei  $n$ -Bit Addierern aufbauen. Der erste berechnet die Summe der niederwertigen Bits, jeder der beiden anderen berechnet die Summe der höherwertigen Bits – einer unter der Annahme, dass aus den niederwertigen Bits ein Carry propagiert werden wird, der andere unter der Annahme, dass kein Carry kommen wird. Das Carry Bit des ersten Addierers kontrolliert über zwei MUX-Glieder, wessen Ergebnis schließlich verwendet wird. Das Ergebnis des anderen, Summe und Carry, wird verworfen. Dies ist die Struktur eines *Carry-Select-Addierers*, der für den Fall  $n=1$  in der folgenden Figur gezeigt ist. Unabhängig von  $n$  kann man so die Addition von  $2n$ -Bit-Zahlen mit 50% extra Materialaufwand fast um den Faktor 2 beschleunigen.

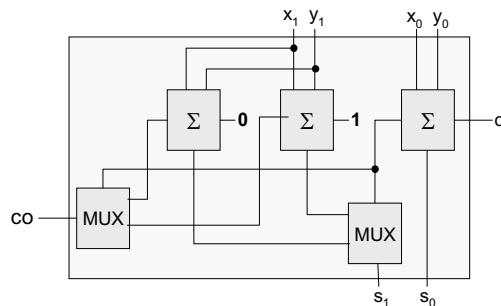


Abb. 5.49: Carry-Select Addierer

## 5.5.2 Rückgekoppelte Schaltungen

Ein Schaltkreis heißt *rückgekoppelt*, wenn der Ausgang eines Schaltgliedes wieder in dessen Eingang geleitet wird. Dies kann direkt oder auf dem Umweg über andere Zwischenglieder

geschehen. Schaltkreise, die wir aus booleschen Termen gewinnen, sind nie rückgekoppelt. Folglich können wir einen rückgekoppelten Schaltkreis nicht unmittelbar durch einen booleschen Term beschreiben. Wozu brauchen wir aber rückgekoppelte Schaltkreise, wenn wir doch jede Schaltfunktion durch einen booleschen Term und damit durch eine nicht-rückgekoppelte Schaltung realisieren können?

Die Antwort ist, dass rückgekoppelte Schaltungen ein *Gedächtnis* haben können. Mit unseren bisherigen Methoden könnten wir zwar Schaltkreise bauen, die elementare Operationen, wie Addition oder Multiplikation, realisieren, wir können aber noch keine *Speicherzelle* konstruieren. Um diese Phänomene zu studieren, analysieren wir ein OR-Gatter, mit Eingängen  $x$  und  $y$ , dessen Ausgang  $z$  mit dem Eingang  $y$  verbunden wurde.

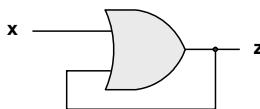


Abb. 5.50: Rückgekoppeltes OR-Gatter

Für  $x = 1$  gilt offensichtlich  $z = 1$ , doch für  $x = 0$  ist sowohl  $z = 1$  als auch  $z = 0$  möglich. War aus irgendeinem Grund einmal  $z = y = 0$ , so bleibt dieser Zustand erhalten, solange wir  $x$  auf 0 halten. Wird  $x$  einmal auf 1 gesetzt, so wird  $z = 1$  und dieser Zustand bleibt hinfest erhalten, auch wenn  $x$  wieder 0 wird. Der Kreis hat sich also „gemerkt“, dass  $x$  früher einmal 1 war.

Von einer Speicherzelle werden wir aber eine bessere Merkfähigkeit erwarten, denn sie muss sich zwei mögliche Werte merken können. Eine solche Speicherzelle können wir bereits mit 2 NOR-Gattern herstellen. Die Schaltung trägt den scherhaften Namen *Flip-Flop*, benannt nach den beiden Zuständen, in denen sie sich befinden kann. Im deutschen Sprachgebrauch findet man auch die Bezeichnung *bistabile Kippschaltung*. Wir betrachten zunächst den *set-reset-Flip-Flop*, der auch als *RS-Flip-Flop* bezeichnet wird. Er besteht aus zwei NOR-Gliedern, deren Ausgänge mit je einem Eingang des jeweils anderen NOR-Gliedes verbunden sind. Die beiden freien Eingänge heißen  $s$  und  $r$ , die Ausgänge  $q$  und  $\bar{q}$ .

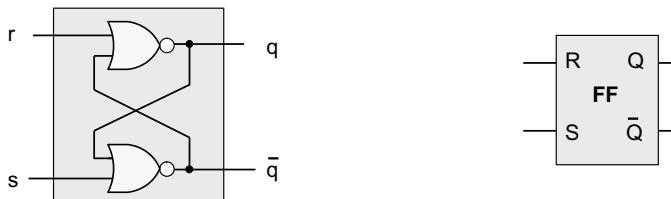


Abb. 5.51: Flip-Flop: Gatterdarstellung und Ersatzschaltbild

Das Verhalten des Kreises kann man durch zwei gekoppelte Gleichungen beschreiben:

$$q' = (r + \bar{q})' \quad \text{und} \quad \bar{q}' = (s + q)'$$

Für  $r = 0$  folgt aus der ersten Gleichung  $q' = (0 + \bar{q})' = \bar{q}'$ , also  $q' = \bar{q}$ . Für  $s = 0$  folgt aus der zweiten Gleichung  $\bar{q}' = (0 + q)' = q'$ , also ebenfalls  $q' = \bar{q}$ . Ist also  $r = 0$  oder  $s = 0$ , so

liegt an  $\bar{q}$  immer das Komplement von  $q$ . Im praktischen Einsatz wird der RS-Flip-Flop nie in dem Zustand  $r = s = 1$  betrieben, so dass man, unter dieser Voraussetzung, immer davon ausgehen kann, dass  $q' = \bar{q}$  ist.

Außer für  $r = s = 0$  hat das obige Gleichungssystem immer genau eine Lösung für  $q$  und  $\bar{q}$ : Für  $s = 1$  folgt  $\bar{q} = 0$  also  $q = 1$  und für  $r = 1$  folgt  $q = 0$ . Für  $r = s = 0$  dagegen ist das Gleichungssystem unterbestimmt: Sowohl  $q = 0$  als auch  $q = 1$  sind mögliche Lösungen. Beide Lösungen sind *stabil*, das heißt, dass die Schaltung nicht zwischen den beiden Lösungen *schwanken* kann: Ist z.B.  $\bar{q} = 1$ , so liegt dieser Wert am Eingang des zweiten NOR-Gliedes und bewirkt, dass  $q = 0$  ist.  $q = 0$  liegt zusammen mit  $s = 0$  am ersten NOR-Glied und bestätigt  $\bar{q} = 1$ . Ebenso würde auch  $\bar{q} = 0$  sich selbst stabilisieren.

Demzufolge wird der RS-NOR-Flip-Flop folgendermaßen betrieben: Der Ruhezustand ist  $r = s = 0$ . Ein Impuls 1 auf  $s$  (set) setzt  $q$  auf 1. Ein Impuls 1 auf  $r$  (reset) setzt  $q$  auf 0. Fällt der Impuls (auf  $r$  oder  $s$ ) wieder auf 0 ab, so bleibt der vorige Wert von  $q$  erhalten. Damit *merkt* sich die Schaltung also, ob die letzte Aktion ein *set* oder ein *reset* war.

In der folgenden Figur wurde durch Vorschalten zweier AND-Gatter und einer Negation dafür gesorgt, dass die Eingänge des inneren Flip-Flops nie gleichzeitig auf 1 liegen können. Einen solchen Baustein nennt man *D-Flip-Flop*. Falls *Enable* gesetzt ist, wird der Wert von D beim nächsten CLK-Signal im Flip-Flop gespeichert. Wir haben ein rudimentäres 1-Bit Register vorliegen.

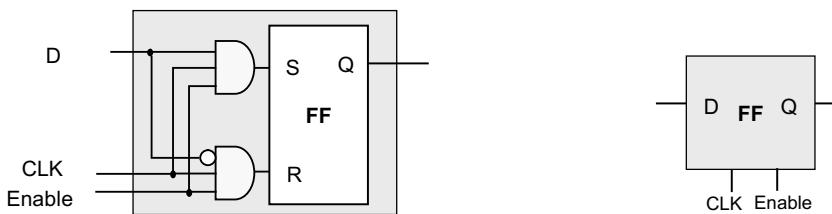


Abb. 5.52: D-Flip-Flop als einfaches pegelgesteuerter Register und zugehöriges Schaltsymbol

Ersetzt man die NOR-Glieder eines RS-NOR-Flip-Flops durch NAND-Glieder, so entsteht ein RS-NAND-Flip-Flop, dessen Verhalten dual zu dem des RS-NOR-Flip-Flop ist.

### 5.5.3 Einfache Anwendungen von Flip-Flops

Flip-Flops finden vielfältige Verwendung, nicht nur als Speicherbausteine. Ein kleines Beispiel soll hier stellvertretend erwähnt werden. Wir stellen uns einen mechanischen Schalter vor, der geöffnet oder geschlossen wird. Jede Taste der Computertastatur ist ein solcher Schalter. Man erwartet, dass bei Betätigung des Schalters eine elektrische Größe (Strom oder Spannung) von einem alten Wert zu einem neuen Wert springt, zum Beispiel von 0 V auf 5 V, und auf dem neuen Wert verharrt, bis die Schalterstellung wieder verändert wird. In Wirklichkeit beobachtet man, dass der Schalter *prellt*, das heißt, dass die Spannung für eine kurze Weile zwischen dem alten und dem neuen Wert hin- und herspringt, bis sie nach einer Weile auf dem

endgültigen Wert verharrt. Bei einer Computertastatur kann dies dazu führen, dass das einmalige Drücken einer Taste den entsprechenden Buchstaben mehrfach auf den Bildschirm bringt.



Abb. 5.53: Prellender und idealer Schalter

Um einen solchen Schalter zu *entprellen*, bedient man sich eines Flip-Flops. Am Eingang des Wechselschalters liegt der boolesche Wert 1, der Schalter leitet diesen alternativ zum Set- oder Reset-Eingang eines RS-Flip-Flops. Wird der Schalter eingeschaltet, so gelangt der Wert 1 an den Set-Eingang. Auch wenn dieser zwischenzeitlich auf 0 fällt, bleibt nach dem ersten 1-Puls auf Set der Wert 1 am Q-Ausgang so lange erhalten, bis der Wechselschalter umgelegt wird und den logischen Wert 1 auf den Reset-Eingang legt.

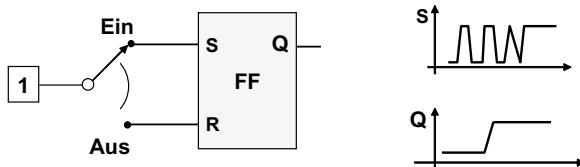


Abb. 5.54: Entprellung mit Flip-Flop

Die wichtigste Verwendung finden Flip-Flops allerdings beim Aufbau von Speicherzellen. Durch einen 1-Puls auf den Set- bzw. den Reset-Eingang speichert man eine 1 bzw. eine 0. Der gespeicherte Wert liegt am Ausgang  $Q$  und bleibt so lange erhalten, bis er durch einen erneuten Puls auf Set oder auf Reset überschrieben wird.

### 5.5.4 Technische Schwierigkeiten

Dass beim Umlegen eines mechanischen Schalters eine Spannung nicht augenblicklich von einem alten zu einem neuen Wert umspringt, haben wir bereits erwähnt. Auch wenn wir Transistoren bzw. AND-Glieder als Schalter einsetzen, dauert es immer eine kurze Zeit, bis sich der neue Schaltzustand eingestellt hat. Deshalb können wir auch die Taktrate eines Prozessors nicht beliebig erhöhen. Weil aber Schaltglieder eine gewisse Zeit brauchen, um den neuen Zustand einzunehmen, und verschiedene Glieder je nach Komplexität verschiedene Zeiten, können in der Zwischenzeit kurzfristig unbeabsichtigte Schaltzustände auftreten, die unangenehme Effekte hervorbringen können.

Als Beispiel (siehe Abbildung 5.55) betrachten wir die boolesche Schaltung, die dem Term  $(A + 0) \cdot (A \cdot 1)' = A \cdot 1' = 0$  entspricht. Der Term vereinfacht zu  $A \cdot A' = 0$ . Unabhängig von dem Input-Wert bei  $A$  sollte der Ausgang, der in der Figur mit  $Z$  bezeichnet ist, den Wert 0 behalten.

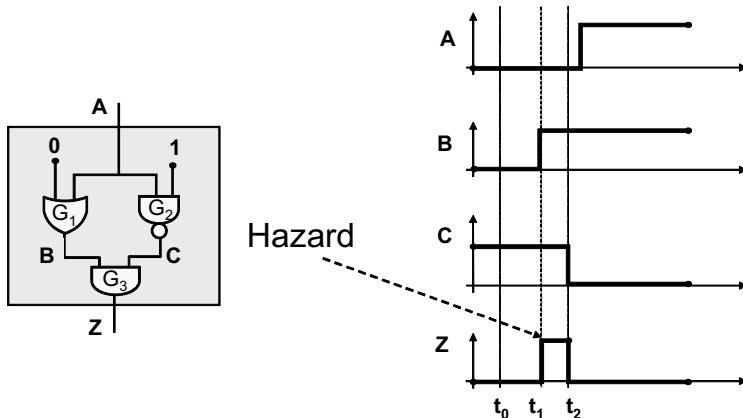


Abb. 5.55: Spannungsverläufe an verschiedenen Punkten in einer Schaltung mit Hazard

Wir betrachten nun einen Zeitpunkt  $t_0$ , zu dem  $A$  von 0 auf 1 umschaltet. Dabei wechselt der Ausgang des OR-Gliedes  $G_1$  von 0 auf 1 und der Ausgang des NAND-Gliedes  $G_2$  von 1 auf 0. Wir erhalten wieder 1 und 0 am Eingang des letzten AND-Gatters, also  $z = 0$ .

Nun nehmen wir aber an, dass Gatter  $G_1$  zum Zeitpunkt  $t_1$  den neuen Schaltzustand bereits eingenommen hat, Gatter  $G_2$  aber erst etwas später, zur Zeit  $t_2$ . In der Zwischenzeit, von  $t_1$  bis  $t_2$ , liegen beide Eingänge von  $G_3$  auf 1, und im Ausgang, der nach der Theorie konstant 0 sein sollte, ist für eine Zeitspanne  $t_2 - t_1$  ein 1-Puls entstanden.

Diesen Puls, der nur kurzzeitig während eines Schaltvorganges auftritt, nennt man auch einen *Hazard*. Ein Hazard kann in einer Schaltung mit speichernden Gliedern viel Unheil anrichten. Er könnte zum Beispiel ausreichen, um einen Flip-Flop versehentlich zu schalten. Um Hazards auszuschließen, muss man boolesche Schaltungen ggf. mit zusätzlichen Schaltgliedern ausstatten. Zu einem booleschen Term gilt es dann also, einen äquivalenten hazard-freien booleschen Term zu finden. Auf dieses Problem wollen wir hier aber nicht weiter eingehen.

## 5.5.5 Synchrone und asynchrone Schaltungen

Bis nach einem Input ein stabiler Schaltzustand eingetreten ist muss man, wie wir bereits anhand der Additionsschaltung diskutiert haben, eine bestimmte Zeitspanne warten. Diese hängt von der Komplexität der Schaltung, insbesondere ihrer Schachtelungstiefe ab. Durch die vielen Schaltkreise, die in einem konkreten Rechner ineinander greifen ist eine zeitliche Koordination ohne einen vorgegebenen Takt unmöglich. Insbesondere, wenn ein System eine sehr komplexe Schaltung beinhaltet, muss man nicht mit allen Aktionen warten, bis der langsamste Kreis garantiert geschaltet hat, sondern man kann einem langsamen Bauteil mehrere Takte geben während dessen andere Teile schon wieder nützliche Arbeit erledigen.

Eine Schaltung, bei der gewisse Aktionen nur zu vorgegebenen Taktzeiten stattfinden, heißt *synchrone Schaltung*, im Gegensatz zu den vorher diskutierten *asynchronen Schaltungen*. Wir werden uns im Folgenden mit solchen synchronen Schaltungen beschäftigen.

Zunächst gehen wir davon aus, dass wir einen Taktgeber, etwa durch einen Schwingquarz realisiert haben, der periodisch zwischen dem Signal 1 und dem Signal 0 schwankt. Idealerweise beschreibt der Signalverlauf des als CLK (für *clock*) bezeichneten Signals eine ideale Rechteckkurve. In der Praxis dauert es immer eine (wenn auch extrem kurze) Zeit, während der das CLK-Signal von 0 auf 1 steigt bzw. von 1 nach 0 fällt. Man nennt diese Teile des Signalverlaufs steigende bzw. fallende *Flanke*.

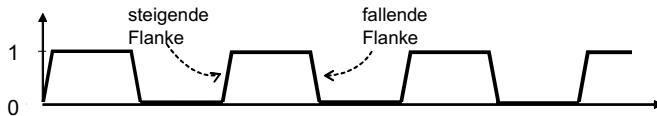


Abb. 5.56: Clock-Signal

### 5.5.6 Getaktete Flip-Flops

Getaktete Flip-Flops können ihren Zustand nur zu bestimmten Zeitpunkten ändern, wenn z.B. ein Uhrimpuls vorliegt. Einen solchen getakteten Flip-Flop haben wir bereits in Abb. 5.52 gesehen. Das von einem Taktgeber erzeugte Signal öffnet und schließt die als Schalter den R- und S-Eingängen vorgelagerten AND-Glieder. Nur solange der Taktgeber eine 1 produziert, kann ein Signal an S oder an R den Speicherzustand beeinflussen. Allerdings erfordert das korrekte Funktionieren des in Abb. 5.52 gezeigten einfachen Registers ein gutes Timing. Das Signal an D muss gehalten werden, solange das Clk-Signal 1 ist. Verändert das D-Signal vorher seinen Wert, so wird dieser veränderte Wert gespeichert.

Daher zieht man es vor, *flankengesteuerte* Flip-Flops zu benutzen. Nur in der kurzen Zeitspanne einer aufsteigenden (oder absteigenden) Flanke des CLK-Signals muss das Inputsignal vorliegen. Dieses kann dann im Flip-Flop gespeichert werden. Einen Flip-Flop mit einem solchen Verhalten kann man sich aus zwei einfachen Flip-Flops konstruieren. Die Schaltung nennt man auch *Master-Slave* Schaltung, weil der Master seinen Wert an den Slave während des Flankenwechsels weitergibt und anschließend die Schotten dicht macht: Ist das CLK-Signal 1, so wird durch S bzw. R das Q des ersten Flip-Flops gesetzt bzw. zurückgesetzt. Fällt das Clocksignal, so schließt der Master während gleichzeitig der Slave öffnet und den gespeicherten Wert übernimmt.

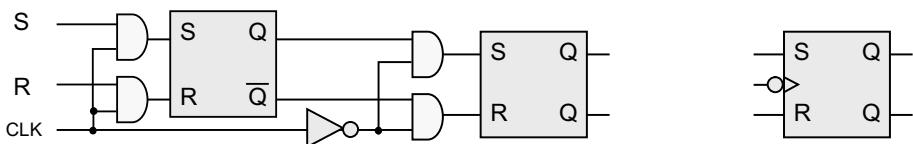


Abb. 5.57: Flankengesteuerter R-S-Flip-Flop, als Master-Slave Schaltung und zugehöriges Schaltsymbol

Der gezeigte Flip-Flop wird also von der fallenden CLK-Flanke gesteuert, was in dem Ersatzschaltbild durch dem Kreis vor dem CLK-Eingang angedeutet wird. Durch ein zusätzliches Negationsglied erreicht man eine Ansteuerung durch die steigende Flanke.

Schaltet man dem  $R$ -Eingang noch ein Negationsglied vor und fasst dann  $S$  und  $R'$  zu einem Eingang zusammen, so erhält man einen *flankengesteuerten D-Flip-Flop*, analog zu Abb. 5.52. Diese finden auch Verwendung als Verzögerungsglieder (*delay*). Ein am Eingang anliegender Wert liegt einen Takt später am Ausgang an. Wir werden sehen, dass wir jede sequentielle Schaltung mit Hilfe von booleschen Schaltgliedern und solchen als delay dienenden D-FlipFlops konstruieren können.

### 5.5.7 Zustandsautomaten

Während eine boolesche Funktion  $f$  eine Eingabe  $x = (x_1, \dots, x_n)$  direkt in eine Ausgabe  $z = f(x_1, \dots, x_n)$  verwandelt, berücksichtigt eine sequentielle Schaltung auch noch einen Zustand  $q$  der zum Beispiel durch die Inhalte einer Reihe von Registern gegeben sein kann:  $q = (q_1, \dots, q_r)$ .

Damit ist die Ausgabe  $z$  nicht nur von der externen Eingabe  $x = (x_1, \dots, x_n)$ , sondern auch von dem Inhalt der Register  $q = (q_1, \dots, q_r)$  abhängig, also

$$z = \gamma(x_1, \dots, x_n, q_1, \dots, q_r).$$

Außerdem kann sich bei jedem Uhrtakt der Zustand verändern, so dass man den neuen Zustand  $q^+ = (q_1^+, \dots, q_r^+)$  mittels einer weiteren booleschen Funktion  $\delta$  aus den Inputs und den alten Zuständen berechnen kann als

$$q^+ = \delta(x_1, \dots, x_n, q_1, \dots, q_r).$$

Den Schaltkreis, der  $\gamma$  implementiert, nennt man die *Output Logik* und den Schaltkreis der den neuen Zustand berechnet nennt man die *next-state Logik*.

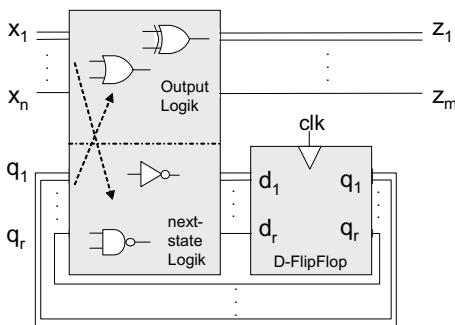


Abb. 5.58: Mealy-Automat

Einen solchen Schaltkreis nennt man auch *Mealy-Automat*. Abstrakt definiert man einen *Mealy-Automaten*  $A = (E, Q, A, \delta, \gamma)$  durch eine endliche Menge  $Q$  von Zuständen, eine Menge  $E$  möglicher Eingaben und eine Menge  $A$  möglicher Ausgaben und zwei Funktionen:

$\delta : E \times Q \rightarrow Q$ , die *Zustandsübergangsfunktion*, und

$\gamma : E \times Q \rightarrow A$ , die Ausgabefunktion.

In der Digitallogik werden die Mengen  $E$ ,  $Q$  und  $A$  durch Kombinationen von Bits dargestellt, also z.B.  $E = \{0, 1\}^n$ ,  $Q = \{0, 1\}^r$  und  $A = \{0, 1\}^m$ , so dass  $\delta : \{0, 1\}^n \times \{0, 1\}^r \rightarrow \{0, 1\}^m$  als Kombination von  $m$  vielen  $(n+r)$ -stelligen booleschen Funktionen implementiert werden muss. Analoges gilt auch für  $\gamma$ . Für die Repräsentation nicht benötigte Bitvektoren stören nicht. In den Schalttabellen kann man ihnen einen beliebigen Wert zuordnen.

Der jeweils aktuelle Zustand  $q = (q_1, \dots, q_r)$  wird durch eine Gruppe von D-FlipFlops realisiert. Der Übergang von Zustand  $q$  in den Nachfolgezustand  $q'$  geschieht bei der Flanke des Taktsignals, wenn die an den D-Flipflops einliegenden Eingabewerte  $d_1, \dots, d_r$  in die Ausgänge  $q_1, \dots, q_r$  übernommen werden.

Moore-Automaten unterscheiden sich von Mealy-Automaten nur dadurch, dass die Ausgabefunktion nicht von  $E$ , sondern nur von  $Q$  abhängt:  $\gamma : Q \rightarrow A$ . Ansonsten sind beide Automatentypen gleichwertig und gleich nützlich. (In Abschnitt 9.2.2 ab S. 701 werden wir in einem anderen Zusammenhang eine Variante von Moore-Automaten kennenlernen.)

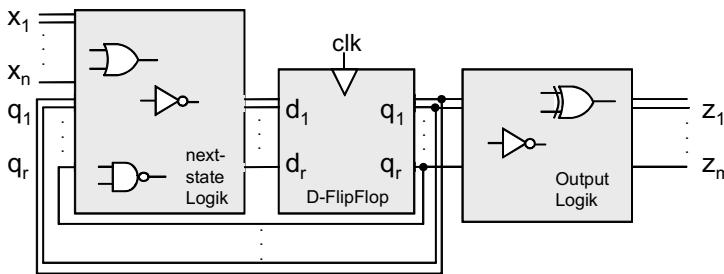
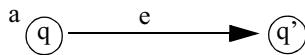


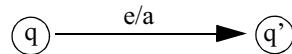
Abb. 5.59: Moore-Automat

### 5.5.8 Entwurf sequentieller Schaltungen

Automaten geraten sehr übersichtlich, wenn man sie graphisch darstellt. Für jeden Zustand  $q$  zeichnet man einen kleinen Kreis, den man mit  $q$  beschriftet. Falls  $\delta(e, q) = q'$  zeichnet man einen Pfeil von  $q$  nach  $q'$ , den man mit  $e$  beschriftet. Im Falle eines Moore-Automaten ist die Ausgabe nur vom Zustand abhängig, daher kann man direkt an jeden Zustand  $q$  den Ausgabewert  $\gamma(q)$  anheften. Die Information  $\delta(e, q) = q'$  und  $\gamma(q) = a$  wird also dargestellt durch



Im Falle des Mealy-Automaten ist die Ausgabe auch vom Input abhängig. In diesem Falle beschriftet man den von  $q$  startenden Pfeil mit Beschriftung  $e$  zusätzlich mit der Ausgabe  $\gamma(e, q)$ . Die Information  $\delta(e, q) = q'$  und  $\gamma(e, q) = a$  wird folgendermaßen dargestellt:



Auf den graphischen Entwurf des Automaten folgt seine Repräsentation durch Schaltfunktionen und schließlich deren Realisierung durch logische Gatter. Die Schaltfunktion für die Ausgabefunktion  $\gamma$  lesen wir von dem Automatendiagramm ab. Im Falle des Moore-Automaten hat die Schalttabelle für die *Output-Logik* die Eingabespalten  $q_1, \dots, q_r$  und die Ausgabespalten  $(z_1, \dots, z_m)$ , wobei jeder Ausgabewert durch einen Bitvektor  $z = (z_1, \dots, z_m)$  repräsentiert sein soll. Im Falle eines Mealy-Automaten haben wir zusätzliche Eingabespalten  $(x, \dots, x_n)$ .

Die *next-state Logik*, welche die Funktion  $\delta$  implementiert, wird in beiden Fällen durch eine Schalttabelle dargestellt, die in den Eingabespalten  $(x, \dots, x_n, q_1, \dots, q_r)$  den aktuellen Input und den aktuellen Zustand aufnimmt, und in den Ergebnisspalten  $(q_1^+, \dots, q_r^+)$  den neuen Zustand repräsentiert. Weil wir für die praktische Zustandsdarstellung D-Flipflops wählen, in deren D-Eingang der neue Zustand eingespeist wird, bezeichnen wir die Spalten für den Nachfolgezustand auch mit  $(d_1, \dots, d_r)$ .

### 5.5.9 Eine Fußgängerampel

Wir wollen Entwurf und Realisierung einer sequentiellen Schaltung an einem kleinen Beispiel illustrieren. Angenommen, wir sollen einen Fußgängerweg einrichten. Dazu stellen wir eine Ampel auf, die den Autoverkehr regelt. Durch Betätigen eines Sensors kann die anfangs grüne Autofahrerampel dazu veranlasst werden, über gelb auf rot zu springen. Sie bleibt auf rot, solange ein Fußgänger den Sensor betätigt. Ansonsten wechselt die Ampel über gelb-rot wieder auf grün, um den Autoverkehr durchzulassen.

Wir haben also ein System mit 4 Zuständen, die den möglichen Schaltsituationen der Ampel entsprechen und die wir mit Binärzahlen codieren:  $00=grün$ ,  $01=gelb$ ,  $10=gelb-rot$ , und  $11=rot$ . Als Eingabe dient nur der Sensor  $s$ , der entweder gedrückt ist (1) oder nicht gedrückt ist (0). Die folgende Abbildung skizziert den Automaten mit seinen Zustandsübergängen. Mehrere Pfeile mit gemeinsamen Start- und Endknoten fasst man zu einem Pfeil zusammen, dem man die Liste der einzelnen Marken anheftet.

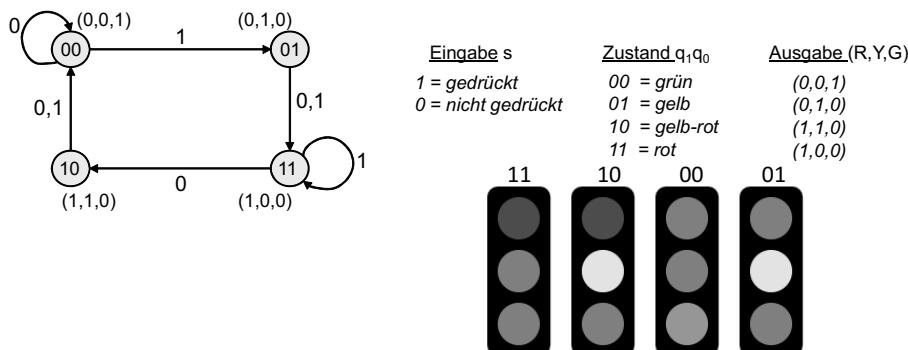


Abb. 5.60: Automat für die Fußgängerampel.

Als Ausgabe erhalten wir ein Tripel  $(R, Y, G)$  das die Schaltsituation der drei Lampen der Ampel,  $R$  (rot),  $Y$  (yellow=gelb) und  $G$  (grün) darstellt. Beispielsweise erzeugt der Zustand  $00=\text{grün}$  die Ausgabe  $(0, 0, 1)$  und der Zustand  $10=\text{gelb-rot}$  die Ausgabe  $(1, 1, 0)$ . Somit ist die Ausgabe nicht von dem Sensor abhängig, so dass wir die Situation mit einem Moore-Automaten beschreiben können. Dass jeder Zustand eine andere Ausgabe erzeugt und damit durch seine Ausgabe charakterisiert wird, ist ein Zufall unseres Beispiels.

Für die Speicherung der Zustandsbits  $q_1, q_0$  benötigen wir zwei binäre D-Flip-Flops, die durch entsprechende Eingänge  $d_1, d_0$  angesteuert werden. Als erstes konstruieren wir die Ausgabeschaltung. Sie ergibt sich sofort aus der Schalttabelle, welche die einzelnen Lampen der Ampel in den verschiedenen Zuständen beschreibt. In unserem Fall sehen wir sofort:  $R = q_1$ ,  $Y = q_1 \oplus q_0$  und  $G = (q_1 + q_0)'$ . Das Ergebnis zeigt die folgende Figur:

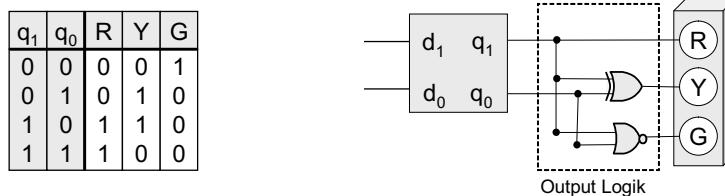


Abb. 5.61: Ausgabeschaltung für die Fußgängerampel

Jetzt fehlt nur noch die Schaltung für die Zustandsübergänge. Hier ist der jeweils nächste Zustand sowohl von dem aktuellen Zustand als auch von dem Input abhängig. Der neue Zustand wird jeweils an den D-Eingängen des D-Flip-Flops gespeichert. Die Schalttabelle können wir aus dem Automatendiagramm in Abb. 5.60 ablesen. Beispielsweise entspricht der Pfeil mit Beschriftung 1 von Zustand 00 zu Zustand 01 der Tabellenzeile 1 0 0 0 1.

Offensichtlich gilt also  $d_1 = q_0$  und mittels eines Karnaugh-Diagramms oder durch Ablesen aus der Tabelle und Vereinfachen erhalten wir  $d_0 = sq_0 + q_0q_1' + sq_1'$ , was die Schaltung in der folgenden Abbildung liefert.

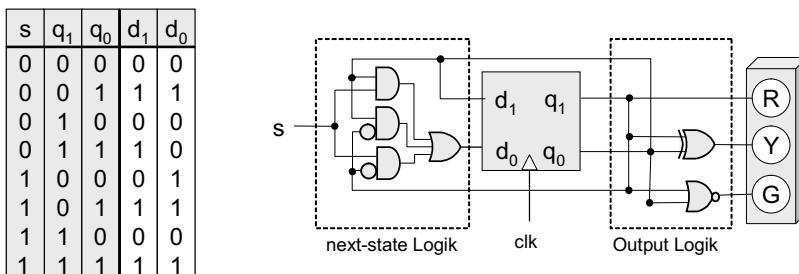


Abb. 5.62: Die fertige Fußgängerampel

### 5.5.10 Die Konstruktion der Hardwarekomponenten

Aus den einfachen booleschen Schaltgliedern AND, OR, NAND, NOR, NOT und rückgekoppelten Gliedern wie dem Flip-Flop werden wir beispielhaft alle wesentlichen Komponenten eines Rechners entwickeln. Dabei wird deutlich werden, dass das Rechenwerk selber, die Arithmetisch-Logische Einheit (engl. *Arithmetical Logical Unit*), kurz *ALU*, als rein boolesche Schaltung realisiert ist, wohingegen für die Speicherbauteile rückgekoppelte Schaltungen in Form von Flip-Flops benötigt werden. Theoretisch werden wir eine komplette Bauanleitung für einen Universalrechner beschreiben. In der Praxis sind jedoch zusätzliche Schaltungen vonnöten. Wir haben exemplarisch bereits auf einige der technischen Probleme hingewiesen: Hazards müssen vermieden werden, Schalter entprellt werden etc.

### 5.5.11 Tristate Puffer

Puffer sind Bauelemente, die ein Signal verstärken bzw. wiederherstellen sollen. Logisch realisieren sie die Identitätsfunktion, sie sind von daher überflüssige Gatter. In der Praxis benötigt man *Puffer*, wenn ein Signal an viele Abnehmer gleichzeitig fließt. In Abb. 5.26 fließt zum Beispiel das Signal des y-Eingangs an vier verschiedene Logik-Gatter. Man spricht von einem *fan-out* von 4. In solchen Fällen muss das Signal durch einen Puffer verstärkt werden. In CMOS-Technik kann man einen solchen Puffer beispielsweise durch zwei hintereinander geschaltete Inverter realisieren.

Ein zweites nicht-logisches Schaltglied ist ein sogenannter *tristate Puffer*. Er dient dazu, eine elektrische Verbindung herzustellen oder zu unterbrechen. Ist *Control=1*, so wird der Eingang zum Ausgang durchgeschaltet, ist er 0, so wird der Ausgang vom Eingang getrennt. Der Ausgang hat also weder den logischen Wert 0, noch den logischen Wert 1. In einer CMOS-Schaltung heißt das, dass der Ausgang weder mit  $V_{CC}$  noch mit *Gnd* verbunden ist. Man sagt, das der Ausgang *hochohmig* ist und gibt seinen Wert mit Z an. Von diesem dritten Zustand (neben 0 und 1) röhrt auch der Name *tristate buffer* (Puffer mit drei Zuständen).

Tristate Puffer werden u.a. benötigt, wenn mehrere Schaltglieder auf einen Bus schreiben sollen, aber immer nur eines Zugang haben soll. In diesem Falle trennt man die Glieder durch je einen tristate Puffer von dem Bus und sorgt dafür, dass von den Kontrolleingängen immer höchstens einer den Wert 1 hat. Es erhebt sich die Frage, warum man in diesem Fall die tristate Puffer nicht einfach durch AND-Glieder ersetzen könnte. Die Antwort ist einfach, wenn man sich die CMOS-Realisierung der Schaltlogik vor Augen hält: logisch 0 ist gleichbedeutend mit einer Verbindung zu *Gnd* und logisch 1 mit einer Verbindung zu  $V_{CC}$ . Verbindet man also einen Ausgang, der logisch 0 liefert mit einem Ausgang, der logisch 1 liefert, so erzeugt man einen Kurzschluss!

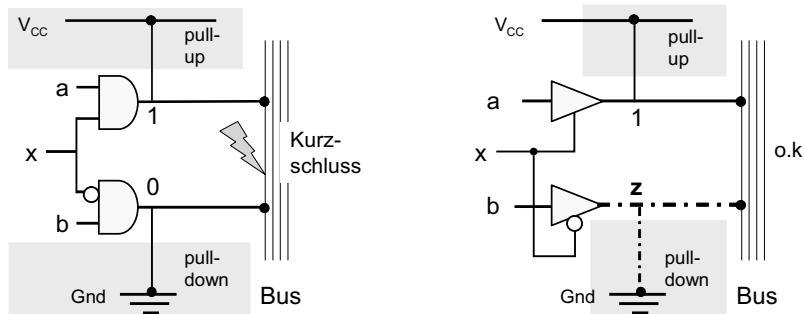


Abb. 5.63: Verbindung der Ausgänge führt zum Kurzschluss; Lösung mit tristate Puffern

In MOS-Technik lässt sich ein tristate Puffer durch einen Inverter und eine Parallelschaltung eines n-MOS mit einem p-MOS realisieren. Das Kontrollsignal geht an das Gatter des n-MOS und gleichzeitig invertiert an das Gatter des p-MOS. Da ersterer logisch 0 unverfälscht weitergibt und letzterer logisch 1, gibt die Parallelschaltung beide Pegel unverfälscht durch, sofern das Kontrollsignal 1 ist und sperrt, wenn dieses 0 ist. Meist hat man das invertierte Signal zu dem Steuersignal ohnehin vorliegen, so dass man einen gesonderten Inverter einsparen kann.

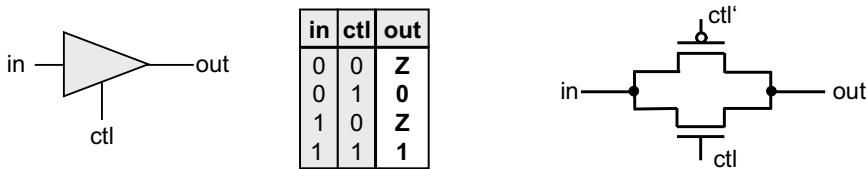


Abb. 5.64: Tristate Puffer: Schaltzeichen, Schalttabelle und MOS-Implementierung

### 5.5.12 Speicherzellen

Wir kommen nun zum Aufbau einer *Speicherzelle*. Den prinzipiellen Aufbau mithilfe eines flankengesteuerten Flip-Flops kennen wir schon. Mit einer 1 am *S*-Eingang setzen wir *Q* auf 1, mit einer 1 am *R*-Eingang setzen wir *Q* auf 0. Wir fügen jetzt noch einige wenige Schaltnetze hinzu, die dazu dienen, bestimmte Speicherzellen in einem aus vielen Zellen bestehenden Speicher zum Lesen oder zum Schreiben auszuwählen. Zunächst setzen wir AND-Glieder als Schalter vor die Eingänge *R* und *S* und hinter den *Q*-Ausgang eines Flip-Flops. Ein Eingang dieser Schalter ist jeweils mit der Leitung SELECT verbunden. Nur wenn *SELECT* = 1 ist, steht der Wert von *Q*, also der gespeicherte Wert der Speicherzelle, an der nach außen geführten Leitung OUT zur Verfügung. Die Schalter an den Eingängen erfordern zusätzlich noch, dass der Speicher zum Schreiben bereit ist. Dies wird durch die Leitung WRITE erreicht. Nur für *WRITE* = 1 und *SELECT* = 1 sind die Schalter vor den Eingängen des Flip-Flops offen. Das zu schreibende Bit liegt als 0- oder als 1-Signal an der Leitung INPUT an. Eine 1 muss den Set-Eingang aktivieren, eine 0 den RESET-Eingang. Daher

wird der INPUT-Eingang sowohl an den SET- als auch über ein Negationsglied zum RESET-Eingang geführt. (Genau genommen handelt es sich um einen trivialen Decodierer.)

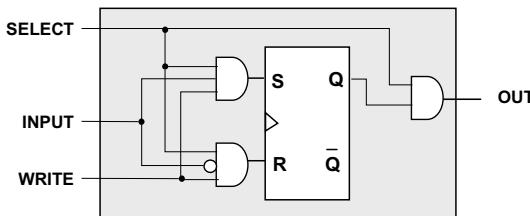


Abb. 5.65: Speicherzelle

In einem Blockschaltbild einer Speicherzelle stellen wir nur die nach außen geführten Leitungen, SELECT, INPUT und WRITE sowie OUT dar. Wir merken uns: Nur bei  $\text{SELECT} = 1$  steht der gespeicherte Wert bei OUT zur Verfügung, und nur bei  $\text{SELECT} = \text{WRITE} = 1$  kann der Wert an der INPUT-Leitung gespeichert werden.



Abb. 5.66: Vereinfachtes Schaltbild einer Speicherzelle

### 5.5.13 MOS-Implementierung von Speicherzellen

Die gezeigten Gatterimplementierungen von MUX, Flip-Flops, Speicherzellen etc. funktioniert mit jeder Technologie, in der man die Gatterbausteine AND, OR, NOT bereitstellen kann, also in MOS-Technik, TTL-Technik, bei optischen Computern, etc. Im Falle der heute vorherrschenden CMOS-Technik gibt es vereinfachte Schaltungen, die mit deutlich weniger Transistoren auskommen, als bei einem Aufbau durch Logik-Gatter notwendig wären.

#### D-RAM Speicher

Am einfachsten ist die Implementierung einer DRAM-Speicherzelle. DRAM steht für *dynamic random access memory*. Sie besteht aus einem Transistor und einem Kondensator. Das Gate wird durch die Adressleitung der Zelle angesteuert und über die Datenleitung wird der Kondensator positiv oder negativ geladen. Nachdem die Ansteuerung der Zelle wegfällt, schließt der Transistor, so dass die elektrische Verbindung einer Platte des Kondensators unterbrochen ist und dieser seine Ladung erhält. Zum Lesen wird das Gate wieder angesteuert und es wird über die Datenleitung der Ladezustand des Kondensators abgegriffen (und verstärkt). Da die Kondensatoren ihre Ladung mit der Zeit jedoch verlieren, müssen alle Speicherzellen regelmäßig, z.B. alle 50 mSec wieder aufgefrischt werden. Durch ihren einfachen Aufbau kann eine hohe Speicherdichte auf kleiner Chipfläche erzielt werden. Die Zugriffs-

zeit, um einen Wert zu speichern, hängt von der Kapazität des Kondensators ab, denn dieser muss bei jedem Schreibvorgang geladen bzw. entladen werden.

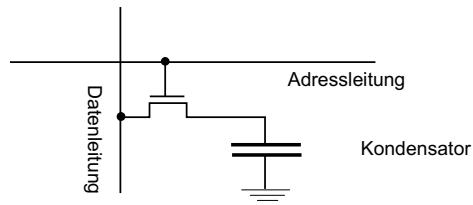


Abb. 5.67: D-RAM Speicherzelle

### S-RAM

Eine S-RAM Speicherzelle besteht aus 6 Transistoren, von denen 4 einen Flip-Flop bilden. Dieser Flip-Flop seinerseits ist aus zwei Invertern aufgebaut, deren Ausgänge jeweils das Gate des Gegners steuern.

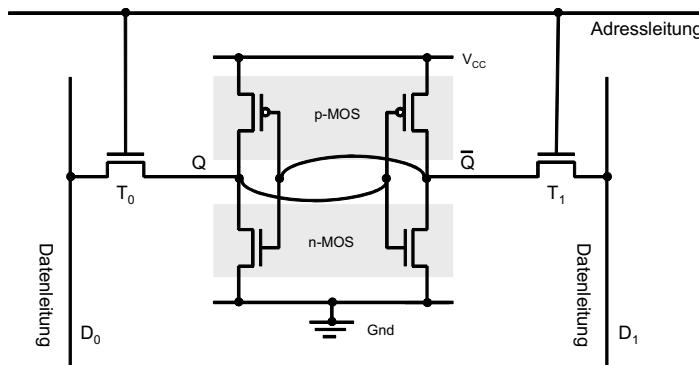


Abb. 5.68: S-RAM Speicherzelle

Offensichtlich gibt es für diese Schaltung genau zwei stabile Zustände, wobei die Logikwerte bei  $Q$  und  $\bar{Q}$  komplementär sind. Falls die Zelle nicht über die Adressleitung selektiert ist, ist der innere FlipFlop in einem der beiden stabilen Zustände. Wird die Zelle über die Adressleitung selektiert, so kann man den aktuellen Speicherzustand über die Spannung zwischen den Datenleitungen  $D_0$  und  $D_1$  abrufen. Zum Speichern eines Wertes legt man eine Spannung zwischen  $D_0$  und  $D_1$  an, um den FlipFlop in den gewünschten Zustand kippen zu lassen.

S-RAMs sind im Vergleich zu D-RAMs aufwendiger, dafür aber schneller. Aus diesem Grunde werden sie z.B. für schnellen Cache-Speicher verwendet. Sowohl S-RAM als auch D-RAM verlieren mit Abschalten des Stromes sehr schnell ihre Information.

## FLASH-Speicher

Flash-Speicherzellen haben die angenehme Eigenschaft, ihre Informationen auch ohne Stromversorgung beizubehalten. Sie sind daher die Grundlagen für viele neue Anwendungen - von den Speicherkarten in Digitalkameras bis zu den beliebten USB-Stiften. Die Speicherzellen für Flash-Speicher sind abgewandelte MOS-Transistoren. Dabei liegt zwischen dem Gate und dem Substrat ein weiteres, durch eine Oxid-Schicht komplett isoliertes „floating gate“. Über eine hohe Spannung am Steuer-Gate bringt man Elektronen auf das floating gate. Dieses steuert dann, je nach Polarität, die Source-Drain Strecke des Transistors.

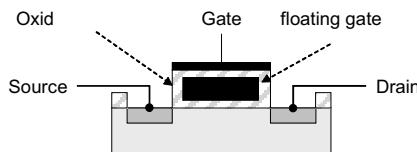


Abb. 5.69: Flash-Speicher

### 5.5.14 Register und adressierbarer Speicher

Eine Gruppe von Speicherzellen nennen wir ein *Register*. Die Anzahl der Speicherzellen in einem Register ist meist gleich der Wortgröße, also 8 Bit, 16 Bit oder 32 Bit. Da man nie einzelne Zellen eines Registers anspricht, kann man die SELECT-Eingänge wie auch die WRITE-Eingänge der einzelnen Zellen verbinden. Diese Leitungen werden dann gemeinsam nach außen geführt, was in dem folgenden Blockschaltbild für ein 4-Bit-Register angedeutet wird.

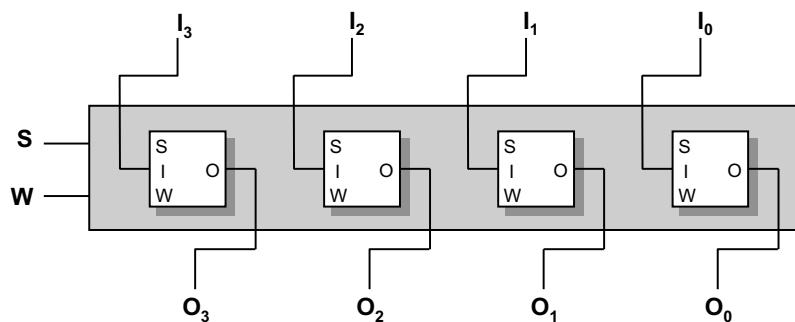


Abb. 5.70: Register

Eine Schaltung, die es gestattet, den Inhalt eines Registers  $X$  in ein anderes,  $Y$ , zu kopieren, ist jetzt konzeptionell einfach zu entwickeln: Wir verbinden die Ausgänge von  $X$  mit den entsprechenden Eingängen von  $Y$ . Dazwischen setzen wir jeweils einen Schalter. Alle diese Schalter, die jeweils durch ein AND-Glied oder einen tristate Puffer realisiert sind, werden durch ein gemeinsames 1-Signal geöffnet, so dass die Information vom  $X$ -Register zum  $Y$ -

Register fließen kann. Gleichzeitig müssen natürlich die SELECT-Eingänge beider Register sowie der WRITE-Eingang des Y-Registers *aktiviert*, also auf 1 gesetzt sein.

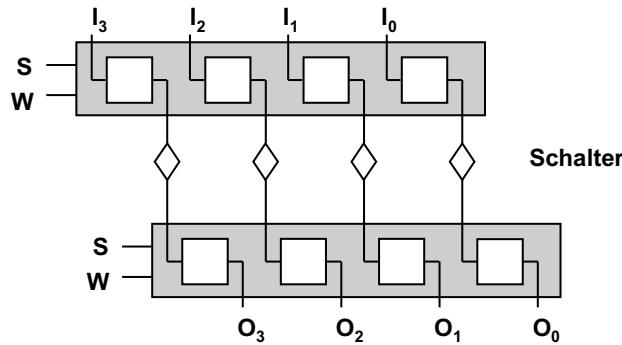


Abb. 5.71: Register-Transfer

Zu guter Letzt wollen wir die Speicherzellen zu einem adressierbaren Hauptspeicher organisieren. Der Übersichtlichkeit halber gehen wir in der Zeichnung von einer Wortlänge von 3 Bit aus und realisieren einen Speicher für 4 Worte. Jeweils 3 Zellen sind zu einem Register zusammengeschaltet. Die INPUT-Eingänge wie auch die OUT-Ausgänge der entsprechenden Bit-Zellen aller Register sind untereinander verbunden.

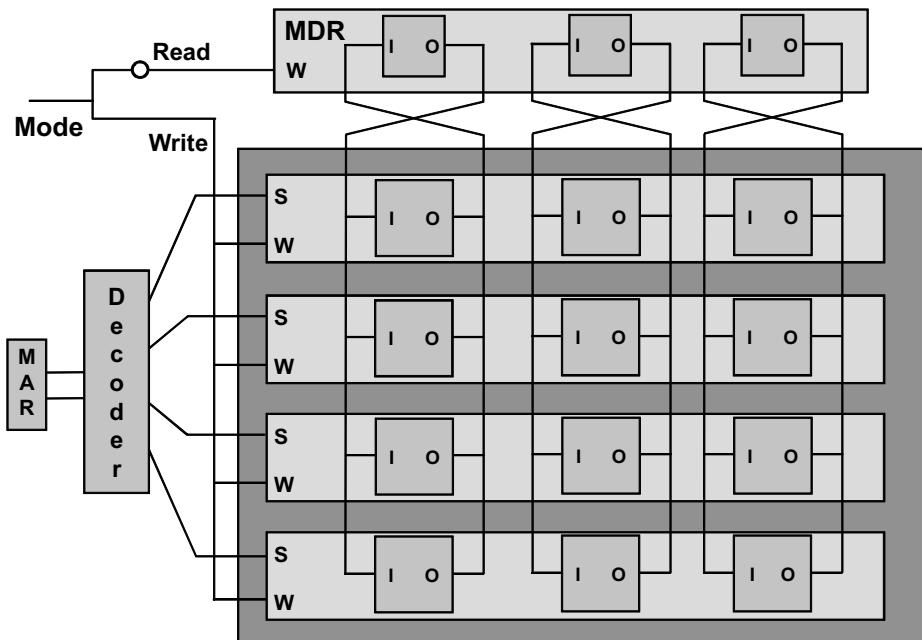


Abb. 5.72: Speicher

Dies ist möglich, da die SELECT-Ausgänge der jeweiligen Register einzeln nach außen geführt sind und nur solche Register geschrieben oder gelesen werden können, deren SELECT-Leitung gerade aktiviert ist. Dass tatsächlich immer genau ein Register selektiert ist, dafür sorgt ein *Decodierer*, siehe S. 442. Dieser setzt eine binär dargestellte Speicheradresse im Speicher-Adressregister (engl. *Memory Address Register*, kurz *MAR*) in ein 1-Signal auf der SELECT-Leitung des gewählten Speicherregisters um. Aufgrund der Funktionsweise eines Decodierers ist immer genau ein Register selektiert. Die WRITE-Eingänge sämtlicher Register sind untereinander verbunden, doch da immer nur ein Register selektiert ist, kann nur dieses verändert werden.

Die zu schreibenden Daten liegen dabei im Daten-Register, (engl. *Memory Data Register*, *MDR*), dessen Ausgänge mit den entsprechenden Eingängen sämtlicher Speicherregister verbunden sind. Um ein Wort zu speichern, bringt man dieses zunächst in das Datenregister MDR. Im Speicher-Adressregister wird die binär dargestellte Speicheradresse hinterlegt. Der Decodierer wählt das entsprechende Speicherregister aus, und wenn anschließend der WRITE-Eingang des Speichers auf 1 gesetzt wird, wird das Datum aus dem MDR in das richtige Speicherregister geschrieben.

Die Ausgänge sämtlicher Speicherregister sind über Schalter mit den entsprechenden Eingängen des Datenregisters MDR verbunden. Die Schalter werden durch ein Signal am READ-Eingang des Speichers geöffnet. Wiederum ist nur ein Speicherregister selektiert, so dass nur dessen Daten abgerufen werden. Da READ bzw. WRITE nur alternativ selektiert werden sollen, fasst man sie zu einem MODE-Input zusammen. MODE = 1 entspricht einem WRITE, MODE = 0 einem READ.

Als Schnittstelle nach außen bietet ein Speicher das Adressregister MAR, das Datenregister MDR sowie eine MODE-Leitung, über die man die Funktionen WRITE bzw. READ auswählen kann.

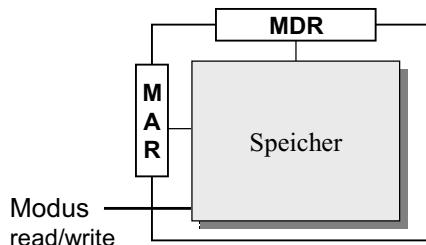


Abb. 5.73: Blockschaubild für den Speicher

### 5.5.15 Die Arithmetisch-Logische Einheit

Die *Arithmetisch-Logische Einheit* (kurz *ALU*) dient zur Realisierung der Elementaroperationen eines Rechners. Dazu gehören, wie der Name schon andeutet, sowohl arithmetische Operationen wie Addition und Subtraktion als auch logische Operationen wie AND, OR oder Prüfung auf Gleichheit. Im Allgemeinen werden zwei Eingabewerte  $X$  und  $Y$  zu einem Ergebniswert  $Z$  verknüpft. Diese Werte stehen in Registern gleichen Namens zur Verfügung. Die

Registerbreite kann 8, 16, 32 oder 64 Bit betragen. Man spricht dann von einem 8-, 16-, 32- oder 64-Bit-Rechner. Bei der Ausführung einer Operation kann es zu verschiedenen Ausnahmefällen kommen. Beispiele für solche Ausnahmefälle sind:

- *Overflow*: Bei der Addition passt das Ergebnis nicht in das Z-Register;
- *Sign*: Das Ergebnis einer Operation war negativ;
- *Zero*: Das Ergebnis einer Operation war 0.

Um solche Ausnahmefälle anzuzeigen, besitzt die ALU ein weiteres Ausgaberegister, das *Flag-Register*. Jedes Bit des Flag-Registers steht dabei für eine solche Ausnahme. Ist das Bit gesetzt, so ist die Ausnahme eingetreten, ansonsten nicht.

Da schließlich die ALU in der Lage sein soll, verschiedene Funktionen auszuführen, muss noch ein Mode-Eingang bereitgestellt werden, über den die auszuführende Operation ausgewählt wird. Schematisch stellt man eine ALU dann auf folgende Weise dar:

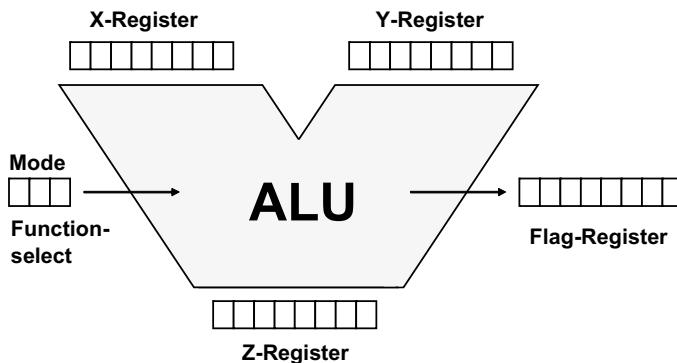


Abb. 5.74: Arithmetisch-Logische Einheit

Wir wollen nun eine geeignete ALU konstruieren. Die logischen Operationen auf Bit-Vektoren sind komponentenweise erklärt, so dass es genügt, mehrere 1-Bit-ALUs nebeneinander zu schalten. Für arithmetische Operationen ist auch ein Übertrag von einer zur nächsten Bitposition zu berücksichtigen, so dass jede 1-Bit-ALU einen zusätzlichen Eingang Carry-In und einen zusätzlichen Ausgang Carry-Out erhalten sollte.

Angenommen, wir wollen 8 verschiedene Operationen implementieren, so wird eine 1-Bit-ALU als boolesche Schaltung mit 6 Eingängen realisiert werden müssen: 3 Eingänge  $N$ ,  $S_0$  und  $S_1$ , um eine der  $2^3$  Operationen einzustellen, ein Carry-Eingang  $C_i$  sowie die Eingänge für die Eingabewerte  $X_i$  und  $Y_i$ . Wir benötigen dagegen nur 2 Ausgänge:  $Z_i$  für das Ergebnis der Operation und  $C_{i+1}$  für den neuen Übertrag. Als logisches Schaltbild erhalten wir:

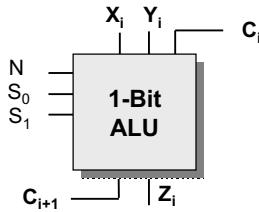


Abb. 5.75: 1-Bit-ALU

Die Funktionsweise der 1-Bit-ALU könnten wir nach Belieben durch eine Wertetabelle spezifizieren und danach die boolesche Schaltung entwickeln. Arithmetische Operationen sind dadurch gekennzeichnet, dass sie  $C_i$  berücksichtigen und  $C_{i+1}$  verändern, während logische Operationen das Carry ignorieren.

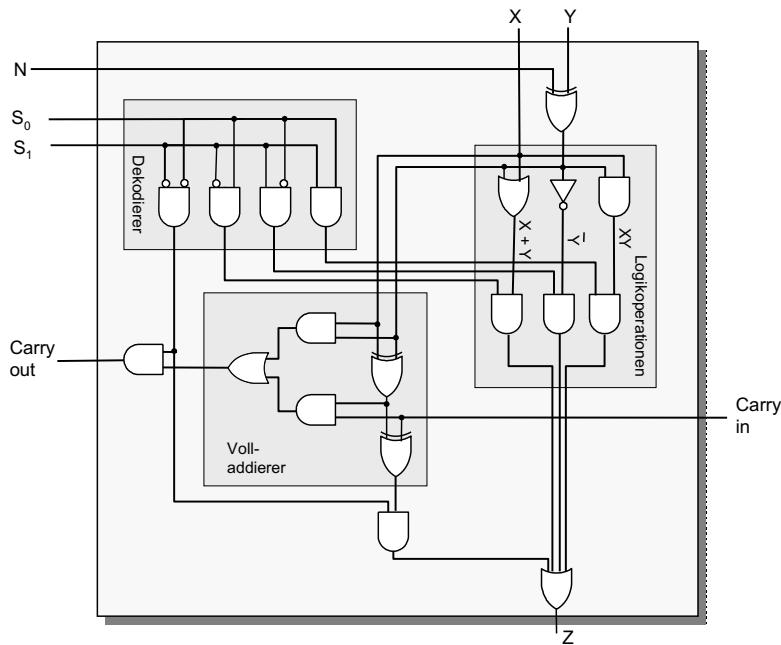


Abb. 5.76: Ein-Bit ALU

Beispielhaft zeigen wir eine 1-Bit-ALU, die addieren und subtrahieren kann und gleichzeitig die grundlegenden logischen Operationen beherrscht. Sie besteht aus drei Baugruppen, die wir schon kennen: einem Addierer, einer Gruppe von Logikoperationen und einem 2-4-Decodierer, der je nach Wert des Wortes  $S_1S_0$  bestimmte AND-Gatter ansteuert, die den Wert einer bestimmten Operation dem Ausgang zuleiten. Die Ausgänge der Operationen werden durch ein OR-Gatter zusammengeführt.

Die Werte 00, 01, 10, 11 von  $S_1S_0$  entsprechen der Reihe nach der Addition  $X + Y$ , und den logischen Operationen  $X \vee Y$ ,  $\bar{Y}$  und  $X \wedge Y$ . Wird der Negationseingang  $N$  gesetzt, so wird  $Y$  komplementiert ( $1 \oplus Y = \bar{Y}$ ). Dann wird aus der Addition die Subtraktion, sofern noch das Carry-In Bit der ersten 1-Bit-Alu gesetzt wird. Aus den logischen Operationen werden durch Setzen des Neg-Eingangs die Operationen:  $X \vee \bar{Y}$ ,  $\bar{Y}$  und  $X \wedge \bar{Y}$ .

Genauso wie wir eine Kaskade von 1-Bit-Addierern zu einem Addierer von Wortbreite zusammengefügt haben (siehe S. 444), schalten wir jetzt auch mehrere 1-Bit-ALUs zu einer ALU von Wortbreite zusammen: die Eingänge  $N$ ,  $S_0$  und  $S_1$  der einzelnen ALUs werden untereinander verbunden, so dass in jeder Komponente die gleiche Funktion berechnet wird. Der Carry-Ausgang der  $i$ -ten ALU wird mit dem Carry-Eingang der  $(i+1)$ -ten ALU verbunden. Der Carry-Eingang der 0-ten ALU, welche auf dem niedrigstwertigen Bit operiert, wird mit dem Neg-Eingang verbunden. Der Carry-Ausgang der ALU für das höchstwertige Bit wird als  $C$ -Bit nach außen geführt und ein XOR der beiden höchstwertigen Carry-Ausgänge als Overflow Bit  $O$ .

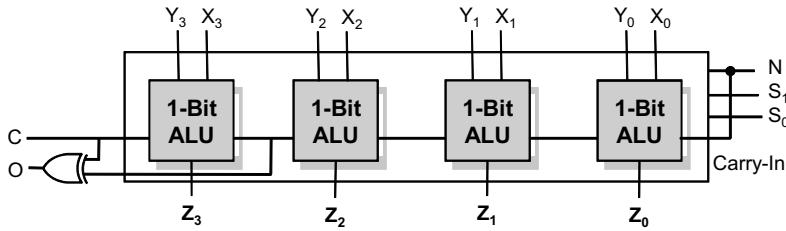


Abb. 5.77: 4-Bit-ALU

Als wichtige arithmetische Operation fehlt bisher noch die Multiplikation. Unsere ALU-Architektur ist für diese Operation noch nicht geeignet. Jede 1-Bit-ALU verknüpft nur  $X_i$  mit  $Y_i$ , d.h. die  $i$ -te Stelle von  $X$  mit der  $i$ -ten Stelle von  $Y$ . Bei der Multiplikation muss aber jede Stelle von  $X$  mit jeder Stelle von  $Y$  verknüpft werden. Im Prinzip können wir die Multiplikation zweier Binärzahlen auf Additionen und Verschiebeoperationen zurückführen. Dies wird deutlich, wenn wir zwei Binärzahlen schriftlich multiplizieren.

$$\begin{array}{r}
 0\ 0\ 1\ 0\ 1 \quad * \quad 0\ 1\ 0\ 1\ 1 \\
 \hline
 0\ 0\ 1\ 0\ 1 \\
 0\ 0\ 1\ 0\ 1 \\
 \hline
 0\ 0\ 1\ 0\ 1 \\
 \hline
 0\ 0\ 1\ 1\ 0\ 1\ 1\ 1
 \end{array}
 \qquad \boxed{5 * 11 = 55}$$

Abb. 5.78: Schriftliche Multiplikation im Binärsystem

Die Binärdarstellung von  $X$  wird jeweils um eine Stelle nach links geschoben. Falls die entsprechende Stelle von  $Y$  gerade 0 war, wird sie annulliert, ansonsten addiert. Da die Addition und die Linksverschiebung üblicherweise in der ALU vorhanden sind, kann die Multiplika-

tion durch eine Folge von ALU-Operationen implementiert werden. Als Alternative bietet sich an, eine gesonderte Multiplikationsschaltung zur ALU beizufügen.

Das *Barrel-Shifter-Multiplikationswerk* orientiert sich an der gerade besprochenen schriftlichen Multiplikation. Es besteht im Wesentlichen aus AND-Gliedern und 1-Bit-Volladdierern. Zunächst stellen wir fest, dass die bitweise Multiplikation gerade der logischen AND-Operation entspricht. Sind  $X$  und  $Y$  die Input-Register mit den Bit-Stellen  $X_{n-1}, \dots, X_0$  bzw.  $Y_{n-1}, \dots, Y_0$  stellen wir ein Gitter her, in dem jede Überkreuzungsstelle  $X_i$  mit  $Y_j$  durch ein AND-Glied verbunden wird. Mit Volladdierern summieren wir die Spalten auf. Dabei wird das Carry-Bit zeilenweise nach links durchgegeben. Ein Überlauf in einer Zeile wird zur nächsten Spalte addiert. Selbstverständlich muss man für das Ergebnis einer Multiplikation ein Register vorsehen, das doppelt so breit ist wie die Input-Register. Meist benutzt man zur Darstellung des Ergebnisses zwei reguläre Register, eines für die niederwertigen und eines für die höherwertigen Stellen.

Die folgende Zeichnung zeigt schematisch ein Barrel-Shifter-Multiplikationswerk für 4-stellige Binärzahlen. Die AND-Glieder an den Überkreuzungspunkten des Gitters sind durch kleine Karos dargestellt.

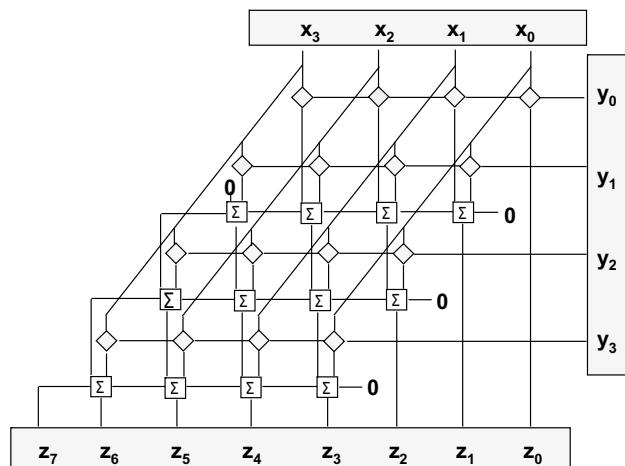


Abb. 5.79: Barrel-Shifter-Multiplikationswerk

Es ist ersichtlich, dass mit dieser Schaltung ein hohes Maß an Parallelität erreicht wird. Sämtliche Bitmultiplikationen können parallel ausgeführt werden, da alle Eingänge der AND-Karos direkt von den Inputregistern kommen. Ein Engpass ist die Addition, da das Carry-Bit von einer zur nächsten Stelle übertragen werden muss, bevor die folgende Addition stattfindet. Bei optimaler Ausnutzung der möglichen Parallelität wird die Multiplikation zweier  $n$ -Bit-Zahlen demnach die  $(2n-1)$ -fache Zeitdauer einer 1-Bit-Volladdition benötigen. Berücksichtigt man allerdings, dass auch für die Addition zweier  $n$ -Bit-Zahlen das Carry-Bit von Stelle zu Stelle übertragen werden muss, so findet man, dass schon die Addition so viel Zeit benötigt wie  $n$  1-Bit-Additionen. Letzt-

endlich dauert also die Multiplikation nur doppelt so lange wie die Addition. Allerdings gibt es Möglichkeiten, die Addition zu beschleunigen, wie wir in Abschnitt 5.5.1 gesehen haben.

## 5.6 Von den Schaltgliedern zur CPU

Die wichtigsten Einzelteile, aus denen eine CPU (Central Processing Unit) aufgebaut ist, haben wir bereits besprochen: ALU, Register und Speicher. Diese Komponenten sind durch Leitungen verbunden, welche durch Schalter geöffnet oder geschlossen werden können. Das Öffnen und Schließen dieser Schalter muss in einer zeitlichen Abfolge koordiniert werden. Daher besitzt eine CPU zunächst einen *Taktgeber*, der die Zeit in einzelne Takte zerhackt. Diese Takte sind sehr kurz, bei einem 1-GHz-Prozessor dauert ein Takt  $10^{-9}$ s, also eine Nanosekunde. Jede Operation der CPU benötigt einen Takt. Für eine einfache Operation, wie etwa die Addition zweier Registerinhalte, werden dazu drei *Phasen* benötigt:

- **Phase 1: (Hol-Phase)**  
Hole die Argumente aus den Registern und stelle sie der ALU bereit.
- **Phase 2: (Rechenphase)**  
Führe die ALU-Operation durch.
- **Phase 3: (Bring-Phase)**  
Speichere das Ergebnis in ein Register.

Für jede dieser Phasen müssen gewisse Schalter geöffnet, andere wieder geschlossen werden. Entsprechend können auch CPU-Operationen, die Datenaustausch zwischen Registern und dem Speicher betreffen, in drei Phasen zerlegt werden. Diesen Phasen entsprechen Leitungen  $P_1$ ,  $P_2$  und  $P_3$ , die abwechselnd auf 1, dann wieder auf 0 gesetzt werden. In Phase  $i$  ist  $P_i = 1$ , alle anderen  $P_i = 0$ .

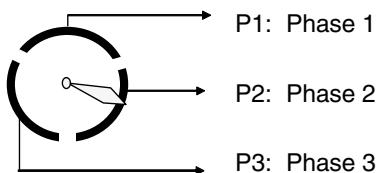


Abb. 5.80: Taktgeber – schematisch

Damit die Datenleitungen zur richtigen Zeit offen bzw. geschlossen sind, werden sie durch Schalter gesichert, die nur für eine bestimmte Phase geöffnet werden können. Schalter werden je nach Erfordernis entweder durch AND-Glieder oder durch tristate Puffer realisiert. Durch einen zusätzlichen, mit  $P_i$  verbundenen Eingang kann der Schalter nur in Phase  $i$  eingeschaltet werden, so dass nur in Phase  $i$  und bei Steuersignal  $S = 1$  Eingang  $E$  mit Ausgang  $A$  verbunden ist:

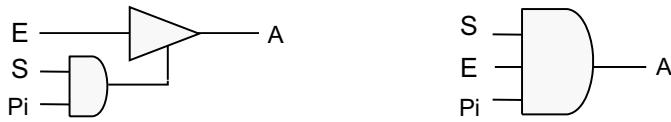


Abb. 5.81: Schalter mit Phaseneingang

### 5.6.1 Busse

Datenleitungen verbinden Register miteinander. Register enthalten Datenworte, d.h. aus mehreren Bits bestehende Daten. Wir wollen in unserer Diskussion von einer 32-Bit-Architektur ausgehen, so dass alle Register 32 Bit breit sind. In einem Registertransfer werden die entsprechenden Bits von Quell- und Zielregister durch parallele Leitungen verbunden. Die Schalter in diesen Datenleitungen sind entweder alle eingeschaltet oder alle ausgeschaltet. Wir stellen die Verbindung zwischen zwei Registern daher lediglich durch einen Pfeil mit einem kleinen Karo dar.

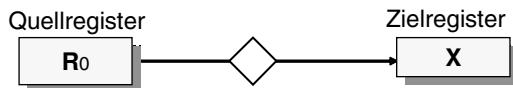


Abb. 5.82: Registertransfer

Der Pfeil steht also für eine Reihe paralleler Leitungen (eine pro Bitstelle) und das Karo für je einen Schalter in jeder dieser Leitungen, die alle an derselben Phase und demselben Steuersignal hängen. Die Pfeilspitze deutet die Richtung an, in die der Registertransfer bei geöffnetem Schalter stattfindet.

Oft hat man eine Auswahl von Registern R<sub>0</sub>, ..., R<sub>k</sub>, von denen man Daten in ein Zielregister X bringen kann, oder ein Register Z, von dem man Daten in eines der Register R<sub>0</sub>, ..., R<sub>k</sub> übertragen will. Der Zugang zu dem Register X oder der Ausgang von Z wird dann in einen parallelen Strang von Leitungen (eine pro Bitstelle) geführt. Die Register R<sub>0</sub>, ..., R<sub>k</sub> werden an diesen Strang angeschlossen. Diese Leitungsstränge nennt man auch *Busse*, in unserem Falle haben wir demnach einen X-Bus und einen Z-Bus. Wir stellen die Busse durch parallele Linien dar.

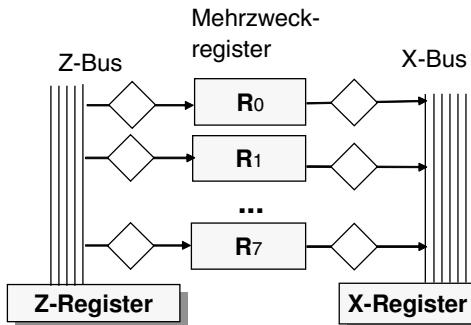


Abb. 5.83: Busse

### 5.6.2 Mikrocodegesteuerte Operationen

Wir haben bereits alle Ingredienzen, um einen Taschenrechner mit einigen Speicherzellen (Registern) zu bauen, kennen gelernt. Wir benötigen dazu zunächst eine ALU, eine Reihe von Registern (hier R<sub>0</sub>, R<sub>1</sub>, ..., R<sub>7</sub>), Busse und Schalter.

Die ALU versehen wir mit zwei Operandenregistern, X und Y, sowie einem Ergebnisregister Z. Dann verbinden wir jedes Mehrzweckregister R<sub>0</sub>, R<sub>1</sub>, ..., R<sub>7</sub> über zwei Busse, den X-Bus und den Y-Bus, mit den entsprechenden Operandenregistern der ALU. Das Ergebnisregister Z der ALU verbinden wir über den Z-Bus mit den Mehrzweckregistern.

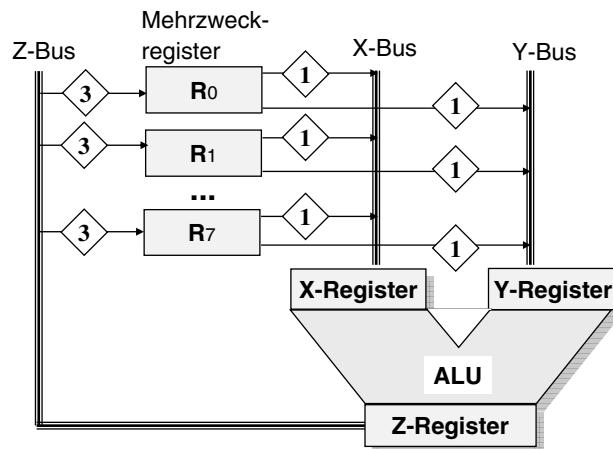


Abb. 5.84: Rechnerkern: ALU + Register + Busse

Zwischen den Registern und den Bussen sitzen Schalter, die nur in bestimmten Phasen geöffnet werden können. Entsprechend der Bedeutung der drei Phasen, Hol-Phase, Rechen-Phase und Bring-Phase, sind die Schalter zwischen den Registern und dem X- und Y-Bus nur in der

ersten Phase, der Hol-Phase, aktivierbar. Nur dann können die Daten von den Registern zu den Operandenregistern der ALU fließen. In der zweiten Phase rechnet die ALU, und in der dritten Phase steht das Ergebnis im Z-Register zur Verfügung. Nur in dieser Bring-Phase sind die Schalter zwischen Z-Bus und den Mehrzweckregistern aktivierbar, damit das Ergebnis in einem der Register abgelegt werden kann. Der komplette Aufbau ist in Abbildung 5.84 dargestellt. Die Zahlen in den Schaltern geben an, in welcher Phase sie aktiv sind.

Jeder der Schalter besitzt aber noch einen Steuereingang  $S_i$ , der auf 1 liegen muss, um den Schalter zu öffnen. Dieses Steuersignal bleibt einen kompletten Takt lang erhalten. Wegen der Phasenabhängigkeit sind die Zugänge zum X- und Y-Bus ohnehin in Phase 2 und 3 und der Zugang vom Z-Bus zu den Registern in Phase 1 und 2 geschlossen.

Die Steuersignale können wir durch eine Gruppe von 3 Bytes darstellen, jedes Byte ist für einen Bus-Zugang verantwortlich, die Bitstellen entsprechen den einzelnen Registern. Zur Illustration betrachten wir das folgende Steuersignal:

0 1 0 0 0 0 0 0	0 0 1 0 0 0 0 0	0 1 0 0 0 0 0 1
X - B u s	Y - B u s	Z - B u s

Wenn in einem Takt dieses Steuersignal vorliegt, wird in der ersten Phase der Inhalt von  $R_1$  zum X-Register und der Inhalt von  $R_2$  zum Y-Register fließen. In der Phase 2 werden alle Schalter geschlossen sein, die ALU wird aus den Operanden einen Ergebniswert berechnen, und dieser wird in Phase 3 gleichzeitig in die Register  $R_1$  und  $R_7$  geschrieben. Bei der angegebenen Schalterstellung wird also die Operation

$$R_1, R_7 := R_1 \text{ op } R_2$$

ausgeführt, wobei die durch Komma getrennten Ziele auf der linken Seite gleichzeitig das Ergebnis der Operation empfangen.

Schließlich müssen wir noch an der ALU einstellen können, welche Operation  $op$  sie berechnen soll. Wir nehmen an, dass unsere ALU ein Repertoire von 64 Operationen umfasst, so dass wir die ausgewählte Operation mit 6 Bit beschreiben können. Neben den grundlegenden arithmetischen, logischen und vergleichenden Operationen sind auch konstante Operationen nicht vergessen worden. Diese dienen lediglich dazu, eine feste Konstante im Z-Register bereitzustellen. Die ALU-Funktionscodes (ALU-FC) sind in der folgenden Tabelle dargestellt. Die Zuordnung der Codes zu einer Operation ist frei wählbar. Die ALU als boolesche Schaltung ist nach beliebigen Vorgaben konstruierbar. In der Operations-Spalte wird angegeben, wie sich der Wert im Z-Register aus den Werten in den X- und Y-Registern ergibt. Ggf. werden noch die Inhalte von X- und Y-Register gegenseitig ersetzt oder vertauscht, was durch  $Y \rightarrow X$  bzw. durch  $X \leftrightarrow Y$  angedeutet wird.

Auf Basis der hier vorgestellten Architektur hat M. Perner seinen mehrfach preisgekrönten CPU-Simulator *MikroSim* entwickelt. Mit diesem Windows-Programm lässt sich die Funktionsweise der in diesem Abschnitt beschriebenen CPU in allen Details und auf verschiedenen Abstraktionsstufen experimentell und visuell nachvollziehen. Eine funktionsfähige Demo-Version kann von der Seite des Autors heruntergeladen werden: <http://mikrocodesimulator.de/>.

Wir gehen von der folgenden Zuordnung von Operationen zu Funktionscodes aus:

ALU-FC	Operation	ALU-FC	Operation
0	Z := Z (Keine Operation)	8	Z := X, Y → X
1	Z := -Z	9	Z := X + 1
2	Z := X	10	Z := X - 1
3	Z := -X	11	Z := X + Y
4	Z := Y	12	Z := X - Y
5	Z := -Y	13	Z := X * Y
6	Z := Y, X ↔ Y	14	Z := X DIV Y
7	Z := X, X ↔ Y	15	Z := X MOD Y

Abb. 5.85: Arithmetische ALU-Operationen

Um eine komplette CPU-Operation auszuführen, wie z.B. die Addition zweier Register und die Speicherung des Ergebnisses

$$R_1 := R_1 + R_2,$$

muss demzufolge der ALU-Code Nr. 11 = (001011)<sub>2</sub> eingestellt sein, die Schaltersignale für die X-Schalter sind 01000000, für die Y-Schalter 00100000 und für die Z-Schalter 01000000. Man kann folglich den ALU-Code mit den Schaltersignalen zusammenfassen und dies als Mikrocode für die CPU-Operation  $R_1 := R_1 + R_2$  darstellen:

CPU-Operation	Mikrocode
$R_1 := R_1 + R_2$	$\Leftrightarrow \underbrace{001011}_{\text{ALU-FC}} \quad \underbrace{01000000}_{\text{X-Bus}} \quad \underbrace{00100000}_{\text{Y-Bus}} \quad \underbrace{01000000}_{\text{Z-Bus}}$

### 5.6.3 Der Zugang zum Hauptspeicher

Der *Hauptspeicher* (engl. *Random Access Memory* kurz *RAM*) hat als Interface zwei Register und einen einstellbaren Modus. Bei den Registern handelt es sich um das *Adressregister* (*MAR* = *Memory Address Register*) und das *Datenregister* (*MDR* = *Memory Data Register*). Wie bereits bei der Behandlung des linearen Speichers (S. 470) besprochen, steht im Adressregister eine Speicheradresse und im Datenregister ein Wert, der an der angegebenen Adresse geschrieben werden soll oder von der angegebenen Adresse gelesen wurde.

Der Zugang zum Adressregister geschieht über den Z-Bus. Im Allgemeinen wurde vorher von der ALU ein Wert berechnet, der dann im Z-Register vorliegt. Über den Z-Bus gelangt er zum MAR. Das Datenregister MDR wird im Unterschied zum MAR sowohl geschrieben als auch gelesen. Daher gibt es einerseits eine Verbindung vom Z-Bus zum MDR, andererseits auch eine Verbindung vom MDR zum Y-Bus. Soll ein Datenwert in den Speicher geschrieben wer-

den, so ist er i.A. gerade berechnet worden und liegt nach der Rechenphase im Z-Register vor. In der 3. Phase kann er über den Z-Bus zum MDR gelangen. Entsprechend geschieht der Zugang vom MDR zum Y-Bus in der 1. Phase. Es bleibt noch die Breite der Register MAR und MDR zu diskutieren. Die Größe von MAR bestimmt den ansprechbaren Adressraum. Im Simulator ist MAR 12 Bit breit, so dass wir  $2^{12} = 4096$  Adressen darstellen können. Jede Adresse bezeichnet ein Byte. Von den über den 32 Bit breiten Z-Bus in das Adressregister gelangenden Daten werden die höherwertigen 20 Bits abgeschnitten.

Das Datenregister ist ebenso groß wie die Mehrzweckregister  $R_0, R_1, \dots, R_7$ , also 32 Bit. Wird ein 32 Bit großes Wort in den Speicher geschrieben, so verteilt es sich auf die 4 Bytes an den Speicheradressen [MAR], [MAR+1], [MAR+2], und [MAR+3]. Entsprechend wird beim Lesen der Inhalt des MDR aus diesen 4 Bytes zusammengefügt. Unser Speicher sieht aber ebenfalls die Möglichkeit vor, 16-Bit-Größen bzw. 8-Bit-Größen zu lesen und zu schreiben. Beim Schreiben einer 16-Bit-Größe werden nur die zwei niederwertigen Bytes aus dem MDR an die Stellen [MAR] und [MAR]+1 geschrieben. Beim Lesen werden die vorderen 16 Stellen des MDR durch Nullen aufgefüllt.

Entsprechend geht man bei 8-Bit-Speicheroperationen vor. Um eine komplette Speicheroperation beschreiben zu können, benötigt man sowohl einen Mode (lesend, schreibend oder wartend) als auch ein Datenformat (8 Bit, 16 Bit oder 32 Bit). Mit je zwei Ziffern lässt sich der Modus beschreiben (00: wartend, 01: lesend, 10: schreibend) und mit zwei weiteren Bits das Datenformat (00: 1 Byte, 01: 2 Byte, 11: 4 Byte). Wollen wir Speicheroperationen beschreiben, benötigen wir also vier weitere Bits, um eine ALU-Operation zu spezifizieren. Zusätzlich benötigen wir einige Bits, um weitere Schalter zu beschreiben, die die Datenwege zwischen MAR bzw. MDR und den Bussen steuern.

Im erwähnten CPU-Simulator stehen sechs Speicher-Zugänge zur Verfügung:

```
Z --> MAR
Z --> MDR
MDR --> Z
MDR --> Y
MDR --> COP
MAR --> Z
```

Die letzten beiden werden wir erst später benötigen. Der dritte ist eigentlich nicht notwendig, da wir einen Datentransfer vom Speicher zum Z-Bus und damit in ein Mehrzweckregister auch ohne die direkte Verbindung  $MDR \dashrightarrow Z$  in nur einem Takt schaffen:

Phase1:	$MDR \dashrightarrow Y$
Phase 2:	$Z := Y$ (Alufunktion 000100)
Phase 3:	$Z \dashrightarrow Register$ .

Allerdings stehen die Daten dann erst in Phase 3 im Z-Register zur Verfügung. Das macht aber nichts, da sie erst in dieser Phase in ein Register übertragen werden können. Der Hauptvorteil des direkten Zugangs  $MDR \dashrightarrow Z$  ist: Die ALU ist frei für zusätzliche Aktivitäten.

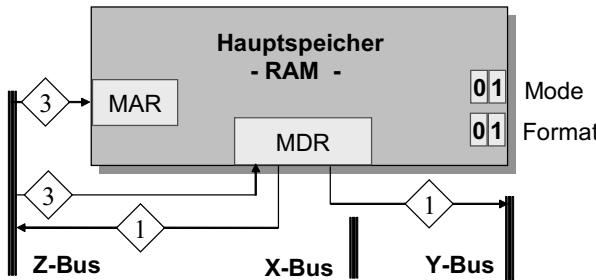
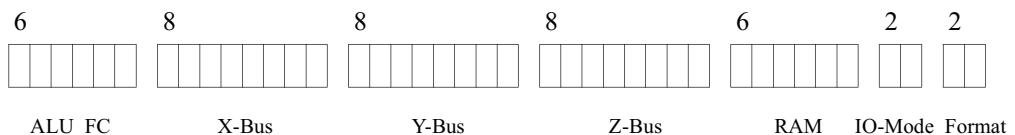


Abb. 5.86: Hauptspeicherzugang

Unsere Mikrobefehle sind nun bereits 40 Bit lang geworden:



Die sechs mit RAM bezeichneten Schalter bedienen die Datenwege zwischen MAR, MDR und den Z- und Y-Bussen, wie in der vorhergehenden Tabelle aufgelistet.

Zur Illustration zeigen wir, wie in zwei Takten der Inhalt von  $R_1$  zu dem 16-Bit-Wort an Speicherstelle 5 addiert werden kann ( $[5] := R_1 + [5]$ ). Wir entwickeln zunächst die beiden Mikrobefehle in ihren einzelnen Phasen:

Takt 1	Phase 1 : keine Operation	
	Phase 2 : $Z := 5$	ALU-FC: 100101
	Phase 3 : $Z \rightarrow MAR$	RAM: 100000
	(IO-Mode = Lesen, Format = 2 Byte)	Modus: 01, Format: 01
	(jetzt liegt der Inhalt von [5h] im MDR)	
Takt 2	Phase 1 : $MDR \rightarrow Y, R_1 \rightarrow X$	XBUS: 01000000
	Phase 2 : $Z := X + Y$	ALU-FC: 001011
	Phase 3 : $Z \rightarrow MDR$	RAM: 010100
	(IO-Mode = Schreiben, Format = 2 Byte).	Modus: 10, Format: 01

Die beiden nacheinander auszuführenden *Mikrobefehle* sind also:

AluFC	XBus	YBus	ZBus	RAM	IO	Fmt
100101	00000000	00000000	00000000	100000	01	01
001011	01000000	00000000	00000000	010100	10	01.

### 5.6.4 Der Mikrobefehlsspeicher – das ROM

*Mikrobefehle* sind Bitfolgen, die wie andere Daten auch in einem Speicher abgelegt werden können. Ein solcher *Mikrobefehlsspeicher* ist Teil der CPU. Er ist als *ROM (Read-Only-Memory)* ausgeführt, d.h. er kann nur gelesen, nicht aber verändert werden. Ansonsten ist das ROM wie jeder andere Speicher aufgebaut, insbesondere besitzt es ein Adressregister, in dem die Adresse eines Speicherwertes abgelegt wird, und ein Datenregister, in dem der dort befindliche Datenwert zurückgegeben wird. Weil die im ROM gespeicherten Daten als Mikrocode interpretiert werden, bezeichnen wir das Adressregister mit *CAR (Code Address Register)* und das Datenregister mit *CDR (Code Data Register)*. In unserem CPU-Modell beabsichtigen wir, bis zu 1024 Mikrobefehle im ROM speichern zu können, daher benötigen wir ein 10 Bit breites CAR. Mikrobefehle sind bis jetzt 40 Bit, d.h. 5 Byte lang. Wir müssen aber noch ein weiteres Byte vorsehen, um Sprünge realisieren zu können. Daher wird im Endeffekt jeder Mikrobefehl 6 Byte lang sein. Damit wird das auf der CPU befindliche ROM eine Größe von 6 kByte besitzen. Wir können aber nicht jedes Byte adressieren wie im RAM, sondern nur jeden Mikrobefehl, d.h. jedes 6. Byte.

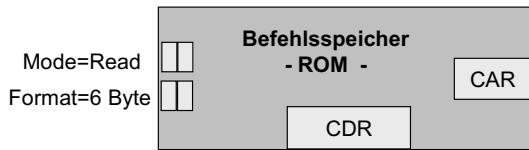


Abb. 5.87: Mikrobefehlsspeicher (ROM)

### 5.6.5 Sprünge

Die im ROM befindlichen Befehle könnte man der Reihe nach abarbeiten. Das wäre aber sehr eintönig und sinnlos, denn dann würde immer dasselbe Programm ablaufen, da der Inhalt des ROM ja unveränderbar ist. Indem beliebige Adressen in das CAR geschrieben werden können, sind wir in der Lage, beliebige Sprünge zu realisieren. Jeder Mikrobefehl besitzt daher ein weiteres Byte, das *Sprungbyte*, das bestimmt, welcher Befehl als Nächster auszuführen ist. Es sind zwei Fälle zu unterscheiden: Entweder steht das Sprungziel von vornherein fest, oder das Sprungziel ergibt sich als Wert einer Berechnung der ALU, aus dem Inhalt des RAMs oder aufgrund einer Bedingung, die aus dem Flag-Register der ALU ablesbar ist. Die erste Art von Sprüngen bezeichnen wir als einen *festen Sprung*, die letzteren Arten heißen *berechnete Sprünge*. Die ersten beiden Bits des Sprungbytes legen den Sprungmodus fest, d.h. um welche Art von Sprung es sich handeln soll. Wir reservieren die Kombination 11 für berechnete und bedingte Sprünge und die Kombinationen 00, 01 und 10 für feste Sprünge. Betrachten wir zunächst die festen Sprünge, so bleiben uns vom Sprungbyte noch 6 Bit übrig, um das Sprungziel festzulegen. Diese 6-Bit-Binärzahl heißt CN für *Code Next*. Wir müssen uns also mit  $2^6 = 64$  möglichen Sprungzielen begnügen. Um diese etwas besser über den Speicher zu verteilen, multiplizieren wir CN noch mit 4, indem wir zwei Nullen anhängen.

Auf diese Weise bietet sich eine logische Gruppierung von je 4 aufeinanderfolgenden Mikrocodeadressen zu einem *Segment* an. Das  $k$ -te Segment besteht aus den Adressen  $4k$ ,  $4k+1$ ,  $4k+2$  und  $4k+3$ . Somit sind die 1024 möglichen Mikrocodeadressen in 256 Segmente gruppiert.

Im *Sprungmodus 00* ergibt sich nun die tatsächliche Sprungadresse (also der Inhalt des CAR) zu  $4 \times CN$ . Ein solcher Sprung heißt auch *absoluter Sprung*. Mit absoluten Sprüngen ist der obere Teil des ROM-Speichers nicht erreichbar, denn die höchste erreichbare Adresse ist 252, d.h. der Beginn des 63. Segmentes. Auf jeden Fall landet ein absoluter Sprung immer auf dem Beginn eines Segmentes.

Im *Sprungmodus 01* wird ein Vorwärtssprung *relativ zum gegenwärtigen CAR* ausgeführt. Die Adresse des neuen Befehls ergibt sich zu

$$CAR := CAR + 1 + 4 \times CN.$$

Entsprechend bewirkt Sprungmodus 10 einen Rückwärtssprung:

$$CAR := CAR + 1 - 4 \times CN.$$

Würde in den obigen Fällen die +1 fehlen, so könnte man immer nur die Befehle am Anfang eines Segmentes erreichen. Außerdem führt ein relativer Sprung, gestartet vom letzten Befehl eines Segmentes, immer wieder auf den Anfang eines anderen Segmentes. Insbesondere führt Sprungmodus 01 mit  $CN = 0$ , d.h. das Sprungbyte 01 000000, zum jeweils nächsten Befehl.

## 5.6.6 Berechnete Sprünge

Auch mit den bisher behandelten Möglichkeiten, Sprünge zu programmieren, ist das Mikroprogramm noch nicht von außen beeinflussbar. Diese Möglichkeit schaffen wir uns jetzt dadurch, dass wir Sprünge von dem Inhalt des RAM oder von Ergebnissen von Operationen beeinflussen lassen. Beides ist im Sprungmodus 11 möglich. In diesem Falle soll die Adresse des Sprunges nicht mehr in CN stehen. Die 6 Bits von CN können also anders genutzt werden. Zunächst sei COP<sup>1</sup> ein Register, in das die Adresse des berechneten Sprungs von außen hingeschrieben werden soll. Für diesen Zweck gibt es einen Datenpfad vom Datenregister des RAM zum COP, wir haben ihn vorher bereits ohne Erklärung als MDR  $\dashrightarrow$  COP erwähnt. Auf diese Weise können im RAM abgelegte Sprungadressen übernommen werden. Falls CN mit den Bits 00 beginnt, geschieht dies und die neue Adresse lautet:

$$CAR := 4 \times COP$$

Beginnt CN *nicht* mit 00, dann sollen die Flags der ALU in Betracht gezogen werden. Die ALU signalisiert bestimmte Ereignisse bei der Berechnung durch Setzen einiger Bits in einem Statusregister, das man oft auch Flag-Register nennt. Wichtige Ereignisse solcher Art sind insbesondere: *Overflow* einer arithmetischen Operation, Ergebnis war 0 oder Ergebnis war

---

1. Im Abschnitt über die Interpretation von Maschinensprachen auf Seite S. 489 wird deutlich, warum die Benennung COP gewählt wurde: C kennzeichnet es als ein Code-Register und OP erinnert daran, dass mit diesem Registers der Anfang einer Routine zur Abarbeitung eines OpCodes angesteuert werden kann.

negativ. Die entsprechenden Flags heißen *overflow flag*, *zero flag* oder *sign flag*. Ein weiteres Flag zeigt an, ob das Ergebnis positiv war. Wie man diese Flags bei den verschiedenen Zahendarstellungen heranziehen kann, um Größenvergleiche von Argumenten durchzuführen, wird im Kapitel über Assembler näher erläutert.

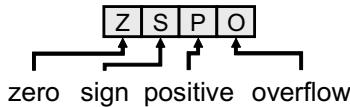


Abb. 5.88: Flag-Register der ALU

Um den Inhalt des Flag-Registers in die Sprungberechnung mit einzubeziehen, gibt man in den letzten 4 Bits von CN eine Maske an. Dies ist eine 4-stellige Binärzahl, die über ein logisches AND mit dem Inhalt des Flag-Registers verknüpft wird. Ist das Ergebnis  $\neq 0000$ , so wird der Sprung zur Adresse  $4 \times \text{COP}$  ausgeführt, ansonsten gilt

$$\text{CAR} := \text{CAR} + 1,$$

d.h. es geht mit dem nächsten Befehl weiter.

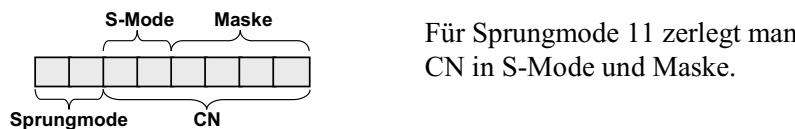


Abb. 5.89: Das Sprungbyte

Wollen wir beispielsweise einen Sprung nur ausführen, falls das Ergebnis einer Berechnung 0 oder Overflow war, so wählen wir die Maske 1001. Nur falls im Flag-Register das erste oder das letzte Bit gesetzt war, ergibt ein AND mit dieser Maske ein Ergebnis  $\neq 0000$ .

Zusammenfassend eine Übersicht über die Interpretation des Sprungbytes

Sprungmode	Sprungziel
00	$\text{CAR} := 4 \times \text{CN}$
01	$\text{CAR} := \text{CAR} + 1 + 4 \times \text{CN}$
10	$\text{CAR} := \text{CAR} + 1 - 4 \times \text{CN}$
11	Zerlege CN in S-Mode (2 Bit) und Maske (4 Bit)

Für Sprungmode = 11 gilt folgende Tabelle:

S-Mode	Sprungziel
= 00	$\text{CAR} := 4 \times \text{COP}$

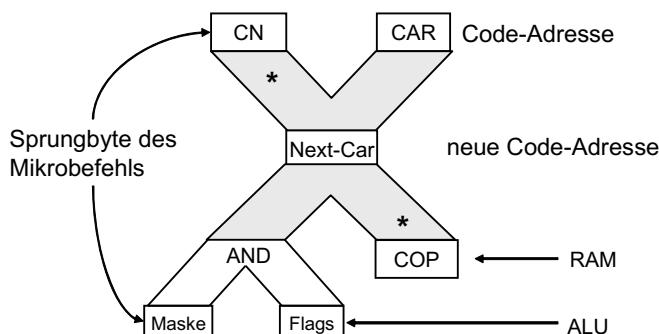
S-Mode	Sprungziel
$\neq 00$	CAR := $4 \times \text{COP}$ , falls (Maske AND Flags) $\neq 0000$
	CAR := CAR+1, falls (Maske AND Flags) = 0000

## 5.6.7 Der Adressrechner

Zur Adressberechnung des nächsten Mikrocodebefehls, wie oben dargestellt, wird eine Adressberechnungseinheit benötigt, die ein einfaches Repertoire an Operationen (AND,  $\times 4$ , +1) besitzen muss. Zweck der Einheit ist, in CAR die Adresse des nächsten Mikrobefehles bereitzustellen. Bei der Berechnung von CAR werden berücksichtigt:

- der gegenwärtige Inhalt von CAR,
  - das Sprungbyte des gegenwärtigen Mikrobefehls,
  - das von außen zugängliche Operationsregister COP,
  - das Flag-Register der ALU.

Nun ist die CPU, bestehend aus Registern  $R_0, \dots, R_7, X, Y, Z$ -Bussen, ALU, RAM, ROM und Adressberechnungseinheit, komplett. Die nächste Aufgabe ist, ein geeignetes Programm für das ROM zu überlegen, so dass die CPU extern (über das RAM) programmierbar wird.



*Abb. 5.90:* Mikroprogramm-Adressrechner

### 5.6.8 Ein Mikroprogramm

Zum Abschluss präsentieren wir ein Mikroprogramm, das die Summe aller Zahlen von 1 bis  $N$  berechnet, wobei  $N$  eine Zahl ist, die im RAM an der Stelle 00H gespeichert ist. Wir erstellen das Programm mit dem bereits erwähnten CPU-Simulator. Nachdem wir mit Datei/Neu eine neue ROM-Datei `Gauss.rom` eröffnet haben, drücken wir zunächst den *Reset-Button* des Simulators, um anschließend mit dem *ROM-Button* den *ROM-Editor* aufzurufen. Dort können wir durch anklicken die einzelnen Teile der Mikrobefehle zusammensetzen, sie mit Kommentaren versehen und nach dem Beenden und Speichern das Programm austesten. Vorher schreiben wir mit dem *RAM-Editor* noch Testdaten in das RAM.

Wir verwenden in unserer Programmbeschreibung die ROM-Adresse des Mikrobefehls als Befehlsnummer. Nach einer allgemeinen Befehlsbeschreibung erklären wir die Aktionen in den einzelnen Phasen und stellen zum Schluss den fertigen Befehl dar. Dabei ist das erste Byte das Sprungbyte, es folgen der ALU-Funktionscode (6 Bit), die Zugänge zu den Bussen (3 Byte), die Datenwege zum Speicher (6 Bit), Speichermodus (2 Bit) und Speicherformat (2 Bit).

---

- 00: Initialisiere R<sub>0</sub>, ..., R<sub>7</sub> und MAR mit 0  
Phase 1: keine Aktion  
Phase 2: Z := 0  
Phase 3: Z --> R<sub>0</sub>, ..., R<sub>7</sub>, Z --> MAR  
01 00 0000 100000 00000000 00000000 11111111 100000 00 00
- 01: Lies N aus [00H] und speichere N in R<sub>0</sub>  
Phase 1: Speicher liest, MDR --> Y  
Phase 2: Z := Y  
Phase 3: Z --> R<sub>0</sub>  
01 00 0000 000100 00000000 00000000 10000000 000100 01 00
- 02: 1 --> MDR (Sprungvorbereitung)  
Phase 2: Z := 1  
Phase 3: Z --> MDR  
01 00 0000 100001 00000000 00000000 00000000 010000 00 00
- 03: MDR--> COP (noch Sprungvorbereitung)  
Phase 1: MDR --> COP  
01 00 0000 000000 00000000 00000000 00000000 000010 00 00
- 04: Addiere R<sub>0</sub> zu R<sub>2</sub>  
Phase 1: R0 --> X, R2 --> Y  
Phase 2: Z := X+Y  
Phase 3: Z --> R2  
01 00 0000 001011 00100000 10000000 00100000 000000 00 00
- 05: Dekrementiere R<sub>0</sub> und springe (an 4\*COP=4), falls Ergebnis > 0  
Phase 1: R0 --> X  
Phase 2: Z := X-1  
Phase 3: Z --> R0  
11 01 0100 001010 10000000 00000000 10000000 000000 00 00
-

### 5.6.9 Maschinenbefehle

Die Vorstellung, größere Programme in Mikrocode programmieren zu müssen, ist abschreckend. Als Programmierer sollte man sich nicht damit plagen müssen, Schalter in Datenwegen zu betätigen, Daten mühsam via Adress- und Datenregister aus dem Speicher zu lesen, Code-Adressen in Code-Adress-Register zu schreiben oder ähnliche lästige Dinge festzulegen. Die Details der Benutzung der Busse und der zeitlichen Abfolge der Teilschritte in den einzelnen Phasen sollen dem Programmierer ebenfalls verborgen bleiben. Eine abstraktere Sicht der CPU ist notwendig.

Diese abstrakte Sicht der CPU zeigt immer noch Speicherzellen und Register, verschwunden sind aber Busse, ALU, Adressrechner, Phasen und Schalter. Stattdessen gibt es Befehle, um Operationen direkt auf Registerinhalten durchzuführen und Daten zwischen Registern und Speicher zu verschieben. Außerdem gibt es Befehle, die direkt Sprünge zu besonders gekennzeichneten Code-Stellen bewirken, anstatt dass mühsam aus Sprungmode und Masken Programmverzweigungen hergestellt werden müssen.

Unser abstraktes Bild der CPU zeigt jetzt nicht mehr acht identische Register  $R_0, \dots, R_7$ , sondern eine Sammlung von Registern, von denen jedes seine spezielle Aufgabe hat und daher auch nur bestimmte Operationen ausführen kann. In unserem Modell wählen wir wieder acht Register (es hätten auch mehr oder weniger sein können), die wir mit  $A, B, X, I, DP, SP, IP, I0$  bezeichnen. Die Register  $A$  und  $B$  nennen wir auch Akkumulatoren. Mit ihnen können arithmetische und Verschiebe-Operationen durchgeführt werden, z.B.:

**ADD A, B**

wobei der Inhalt von  $B$  zu dem Inhalt von  $A$  addiert wird, oder

**MOV A, [61h]**

wobei der Inhalt von Speicherzelle 61h in Register  $A$  kopiert wird.

$X$  dient als Hilfsregister, um Werte kurzfristig zwischenzuspeichern,  $I$  als Index für Schleifen.  $DP, SP$  und  $IP$  stehen für Data Pointer, Stack Pointer und Instruction Pointer. Sie können nicht in arithmetischen Operationen oder in Datenverschiebeoperationen verwendet werden, sondern nur durch spezialisierte Befehle. PUSH und POP z.B. verändern  $SP$ , Sprungbefehle verändern  $IP$ , doch kann man  $IP$  nicht mit einem Datenverschiebebefehl (à la  $MOV IP, [61h]$ ) verändern.  $I/O$  Werte werden aus dem  $I0$  Register in einen Port geschrieben.

Sprungbefehle bewirken eine Verzweigung zu einer gewünschten Stelle des Programms. Diese Stelle kann durch eine Zeilennummer oder eine Maske gekennzeichnet sein. Statt mühsam eine Maske zu erstellen, tragen die Befehle verständliche Namen wie **JMP** (Springe auf jeden Fall, Jump) oder **JNZ** (Springe, falls das letzte arithmetische Ergebnis  $\neq 0$ , Jump if Not Zero).

Statt an dieser Stelle in Details von Maschinensprache einzudringen, verweisen wir auf S. 491 ff, wo Maschinensprache und Assembler von PCs ausführlich behandelt werden. Ein kleines Programm in Maschinensprache zur Addition der Zahlen 1 ... N, wobei  $N$  der anfängliche Inhalt von Speicherzelle 1 ist, mag einen ersten Eindruck vermitteln. Rechts neben jedem Befehl steht als Kommentar eine kurze Erklärung.

MOV A, [0h]	<i>; Inhalt von [0h] nach Register A</i>
MOV B, 0	<i>; Initialisiere B mit 0</i>
nochmal:	<i>; ein Sprungziel (label)</i>
ADD B, A	<i>; Addiere A zu B</i>
DEC A	<i>; Erniedrigre A</i>
JNZ nochmal	<i>; Falls letzte Operation ≠ 0, springe zu nochmal</i>

Zunächst wird hier der Inhalt von Speicheradresse 0 in Register A geladen und B initialisiert. Ab dem mit der Marke *nochmal* gekennzeichneten Befehl wird A zu B addiert und A erniedrigt. War das Ergebnis  $\neq 0$ , so wird mit JNZ (jump if not zero) erneut zur Marke *nochmal* gesprungen.

Genau genommen handelt es sich bei dem obigen Programm um eine lesbare Form der Maschinensprache, die auch Assemblersprache genannt wird. In reiner Maschinensprache hat jeder Befehl eine Nummer, *OpCode* genannt. Die Abkürzungen ADD, MOV, JNZ etc. sind aber leichter zu merken als entsprechende Befehlsnummern.

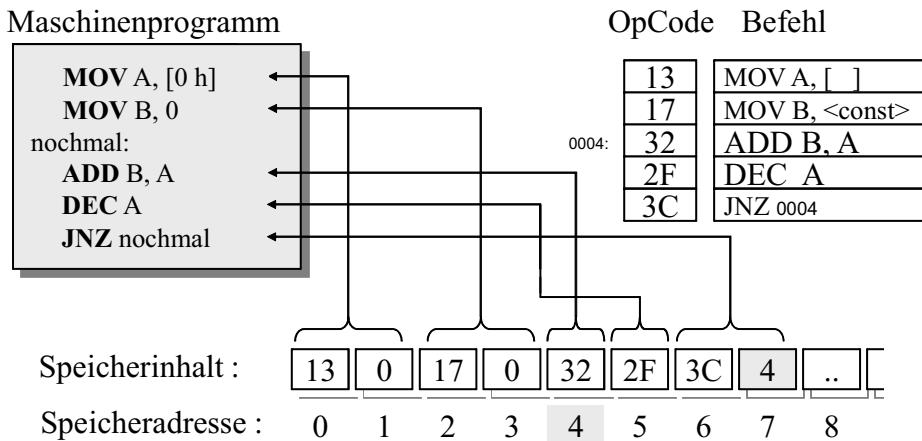


Abb. 5.91: Ein Maschinenprogramm im RAM-Speicher

Einige Befehle sind allein durch ihren OpCode gekennzeichnet. Dazu gehören Befehle wie PUSH, POP, INC A, INC B, ADD A, B etc. Andere Befehle wie ADD A, [061h] oder MOV [01h], B besitzen noch Argumente. Für jeden OpCode steht die Anzahl der Argumente fest. Ein Programm in Maschinensprache besteht aus einer Folge von OpCodes, gegebenenfalls begleitet von ihren Argumenten. Das Programm wird im Hauptspeicher (RAM) abgelegt. Die Aufgabe des Mikrobefehlsspeichers (ROM) auf dem Chip ist es nun, ein solches im RAM abgelegtes Programm auszuführen.

## 5.6.10 Der Maschinenspracheinterpretierer

Das ROM beinhaltet ein Mikroprogramm, dessen Aufgabe es ist, ein Programm in Maschinensprache, das als Folge von OpCodes mit Argumenten im Hauptspeicher vorliegt, auszu-

führen. Zunächst müssen den Registern, auf die sich die Befehle beziehen, tatsächliche Register der CPU zugeordnet werden. Wir ordnen den Registern  $R_0, \dots, R_7$  in dieser Reihenfolge die Register  $IP, A, B, I, X, IO, DP, SP$  zu. Anschließend muss jeder Maschinenbefehl durch ein kleines Stück Mikroprogramm implementiert werden.

Der Befehl INC A, der Register A um 1 erhöhen soll, kann z.B. durch einen einzigen Mikrobefehl implementiert werden. Andere Befehle, wie MOV A, [Speicheradresse], benötigen mehrere Mikrobefehle. Wir legen fest, dass die Routine für den Maschinenbefehl mit OpCode  $n$  an der Stelle  $4 \times n$ , also im  $n$ -ten Segment des ROM liegen soll. Dies vereinfacht das Aufsuchen des Befehls, denn man muss lediglich den OpCode in das Register COP des Adressrechners schaffen und einen absoluten Sprung an  $4 \times COP$  ausführen. Die Dimensionierung des ROM auf 1024 Adressen, d.h. 256 Segmente, lässt danach maximal 256 Maschinenbefehle zu. Wenn sich einige davon über mehr als ein Segment erstrecken, werden es eventuell noch weniger.

Zusätzlich muss am Anfang des ROM nach einer kurzen Initialisierungsroutine noch eine Interpreterschleife eingebaut werden, die

- den nächsten OpCode aus dem RAM liest,
- den Programmzähler erhöht,
- zur Mikroroutine, die den OpCode implementiert, verzweigt.

Dies nennt man den *Load-Increment-Execute-Zyklus*. Dieser ist mit nur zwei Mikroinstruktionen implementierbar! Der Interpreter für Maschinensprache kann in einer Art Pseudocode folgendermaßen beschrieben werden:

---

Segment 0:	Initialisiere alle Register zu 0, initialisiere $R_7$ (SP) zu 7FFh (maximale Stackgröße).
Segment 1:	Lies den OpCode, auf den der Programmzeiger ( $IP = R_0$ ) zeigt, aus dem RAM ins MDR.  Befördere MDR in Register COP des Adressrechners, erhöhe IP und springe nach $4 \times COP$ , d.h. führe den Befehl, dessen OpCode in COP steht, aus.  Springe zum Anfang von Segment 1.
Segmente 2-255:	Implementierung der Maschinenbefehle mit den OpCodes 2-255. Jeder Befehl endet mit einem Sprung zu Segment 1.

---

Der Load-Increment-Execute-Zyklus, also Segment 1, besteht aus den folgenden zwei Befehlen:

---

Befehl 1:	Lade Programmzeiger in MAR:
Phase 1:	$R_0 \xrightarrow{} X$
Phase 2:	$Z := X$
Phase 3:	$Z \xrightarrow{} MAR$

---

---

Befehl 2:	Lies OpCode, erhöhe IP	und verzweige nach $4 * \text{OpCode}$ .
Phase 1:		$[\text{MAR}] \longrightarrow \text{MDR}$ (Modus = Lesen, Format 1 Byte)
Phase 2:		$\text{MDR} \longrightarrow \text{COP}$ $4 * \text{COP} \longrightarrow \text{CAR}$ (Sprungadresse = Segment des OpCodes) $Z := X + 1$
Phase 3:		$Z \longrightarrow R_0$ $Z \longrightarrow \text{MAR}$ (Zeige auf nächst. OpCode oder Argument).

---

Jede Routine des Mikroprogramms muss die folgende Invariante der *while*-Schleife respektieren:

*IP zeigt immer auf den als Nächstes auszuführenden OpCode.*

### 5.6.11 Argumente

*Maschinenbefehle* können ein oder mehrere Argumente beinhalten. Der Befehl MOV <memory>, A besitzt ein Argument, nämlich die zwei Byte umfassende Adresse des Hauptspeichers (*main memory*), an der der Inhalt von A gespeichert werden soll. Nehmen wir an, der entsprechende MOV-Befehl habe OpCode 2C, so würde der komplette Befehl MOV [3FCh], A aus den 3 aufeinanderfolgenden Bytes

2C, 03, FC

bestehen. Anschließend folgt der OpCode für den nächsten Befehl. Der Load-Increment-Execute-Zyklus hat dafür gesorgt, dass *IP* um 1 erhöht wurde, bevor zur Adresse  $4 \times 02C$  gesprungen wurde. *IP* zeigt jetzt auf das Argument des Befehls. In der Implementierung von OpCode 2C kann man daher ausnutzen, dass der Programmzähler *IP* bereits auf das erste Byte des Argumentes zeigt, und man muss dafür sorgen, dass am Ende *IP* um 2 erhöht wird, damit die oben erwähnte Invariante nicht verletzt wird.

## 5.7 Assemblerprogrammierung

*Maschinensprache* ist eine Sammlung von Befehlen, die dem Programmierer zum direkten Zugriff auf die CPU zur Verfügung steht. Im Grunde ist es unangemessen, diese Befehlsammlung als *Sprache* zu bezeichnen, fehlen doch die grundlegenden Strukturierungsmittel höherer Programmiersprachen. Dafür gestattet Maschinensprache den unmittelbaren Zugang zur gesamten Hardware: der CPU, dem Speicher, Bildschirm, Tastatur, seriellen und parallelen Eingängen, Laufwerke, Maus etc. Ein zweiter Grund, Maschinensprache statt einer höheren Programmiersprache zu benutzen, ist, dass man nur in Maschinensprache eine genaue Kontrolle über die Ausführungszeiten der Befehle hat. Man kann zeitkritische Programmteile

sehr effizient in Aktionen der CPU umsetzen. Allerdings gehört viel Übung dazu, Konstrukte höherer Programmiersprachen besser in Maschinensprache zu übersetzen, als dies ein guter optimisierender Compiler kann. Bei einem RISC Prozessor mit mehreren Pipelines kann unter Umständen ein Befehl, der mehr Takte benötigt günstiger sein, als ein Befehl mit weniger Takten, der sich aber schlechter mit anderen Befehlen in der Pipeline verträgt. Eine empfehlenswerte Vorgehensweise ist in jedem Fall, zunächst ein Programm in einer Hochsprache zu entwickeln, anschließend die zeitkritischen Stellen oder die Stellen, die spezielle Hardwarezugriffe erfordern, zu identifizieren und sie gezielt in Maschinensprache umzuschreiben.

Der Nachteil von Programmen in Maschinensprache ist, dass sie nur auf dem Prozessortyp lauffähig sind, für den sie geschrieben wurden. Immerhin bemühen sich die Hardwarehersteller, neue Prozessorgenerationen abwärts kompatibel zu halten, so das auch alte Programme auf der neuen Hardware laufen.

Früher wurden viele zeitkritische Programme in Maschinensprache erstellt. Heute ist mit der schnelleren Hardware die Bedeutung von Maschinensprache zurückgedrängt worden. Maschinensprache wird vor allem als Bindeglied zwischen Hardware und Betriebssystem oder als Zielsprache für einen Compiler verwendet. Für einen neuen Chip werden zunächst ein Betriebssystemkern und ein C-Compiler, also ein Übersetzer, in Maschinensprache geschrieben. Mit einem Cross-Compiler wird dann ein vorhandenes Betriebssystem auf die neue Architektur portiert. C besitzt Anweisungen, die sehr maschinennah sind, dennoch ist C als Hochsprache auf allen gängigen CPUs verfügbar. Daher kann man in C implementierte Betriebssysteme leicht auf andere Architekturen portieren. Auch andere Hochsprachen haben Schnittstellen zur Maschinensprache – etwa in Form von *Inline Assembler*, das sind Programmteile, die in Assembler geschrieben sind.

### 5.7.1 Maschinensprache und Assembler

Jeder Maschinenbefehl besteht zunächst aus einer Bitfolge. Davon identifizieren einige Bits den Typ des Befehls, andere sind Teile von Operanden. Die Bedeutung der einzelnen Bits müsste man im Grunde immer in einer Tabelle nachschlagen. Es gibt daher lesbare Darstellungen von Maschinensprachbefehlen, so genannte *Mnemonics*. So verwendet man z.B. für den Sprungbefehl, der nur ausgeführt wird, wenn das Zero-Flag gesetzt ist, das Mnemonic *JZ* (für *jump on zero*). Programme, die mit solchen lesbaren Abkürzungen, formuliert sind, nennt man Assemblerprogramme. Als *Assemblierer* oder *Assembler* bezeichnet man ein Programm, das Assemblerprogramme in Maschinenprogramme umwandelt (engl: *to assemble = zusammenstellen*).

Ein *Disassemblierer* (oder *Disassembler*) leistet in eingeschränktem Maße die umgekehrte Übersetzung. Aus einem Maschinenspracheprogramm versucht er das ursprüngliche Assemblerprogramm zu rekonstruieren.

Glücklicherweise besitzt ein Assembler noch mehr Fähigkeiten als zu einem Assemblierbefehl den zugehörigen Maschinenbefehl aus einer Tabelle herauszusuchen. Der Assembler erlaubt auch, symbolische Namen für Speicherplätze (Variablen), symbolische Sprungadressen (*Labels*) und Daten (Konstanten) zu verwenden. Außerdem steht ein einfaches Prozedurkon-

zept zur Verfügung. *Makros* dienen dazu, den Code lesbarer und übersichtlicher zu gestalten, und natürlich sind auch Kommentare erlaubt.

Kommerziell verfügbare Assembler für die 80x86 Prozessorfamilie waren zum Beispiel *MASM* (Macro Assembler) der Firma Microsoft sowie *TASM* (Turbo Assembler) von Borland. Derzeit verfügbar sind freie Weiterentwicklungen wie z.B. *masm32*, *goASM* und *fasm*. Letztere können ausführbare Dateien sowohl für Linux als auch für Windows erzeugen und zwar sowohl für den 32-Bit als auch für den 64-Bit Modus. Die Maschinenbefehle und deren Schreibweise in Assemblersprache werden zunächst vom Hersteller der CPU definiert, so dass sich verschiedene Assembler nur in Komfort und Sprachzusätzen unterscheiden. Im Falle der Intel Prozessoren hat sich neben der dominierenden Intel Syntax auch eine AT&T Syntax etabliert, in der, neben anderen Unterschieden, die Assemblerbefehle die generelle Struktur

*op quelle, ziel*

haben, wie z.B. in `movl $100, %ebx`.

### 5.7.2 Register der 80x86-Familie

Die ersten Prozessoren des IBM-PC, der 8088, 8086 und der 80286, waren 16-Bit Prozessoren. Aus dieser Zeit hat der heutige Pentium noch die bekannten 16-Bit-Register geerbt. Es handelt sich um die *Allzweck-Register AX, BX, CX, DX, SI, DI, BP und SP*, die *Segment-Register CS, DS, SS und ES*, den *Befehlszähler IP* sowie das *Flag-Register*. Die niederwertigen Byte (low byte) bzw. die höherwertigen Byte (high byte) der Register AX, BX, CX und DX sind als 8-Bit-Register AL, BL, CL, DL bzw. AH, BH, CH, DH gesondert ansprechbar. Die Registernamen stehen für folgende Abkürzungen: AX=Accumulator, BX=Base, CX=Counter, DX=Destination, SI=Source Index, DI=Destination Index, BP=Base Pointer, SP=Stack Pointer, IP=Instruction Pointer, CS=Code Segment, DS=Data Segment, SS=Stack Segment und ES=Extra Segment.

Seit dem 80386 verarbeitet der Intel-Prozessor 32-Bit-Daten, braucht also auch 32-Bit breite Register. Darum hat man einfach die bestehenden Register auf 32 Bit breite Register *EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP* und *EIP* erweitert. Der Präfix „E“ steht für „extended“. Die alten Register sind immer noch adressierbar, physikalisch stellen sie die niederwertigen zwei Byte der neuen 32-Bit Register dar. Die Segmentregister *CS, DS, SS und ES* behielten ihre Größe von 16 Byte, wurden aber um zwei neue Register, *FS* und *GS* ergänzt. So blieb der Pentium abwärts kompatibel zu den früheren 80x86 Prozessoren. Für die Speicherverwaltung, bei der die Segmentregister eine besondere Rolle spielen, gilt dies aber nur, wenn der Prozessor im so genannten *Real-Mode* betrieben ist. Moderne Betriebssysteme betreiben den Prozessor aber fast durchweg im so genannten *Protected-Mode*.

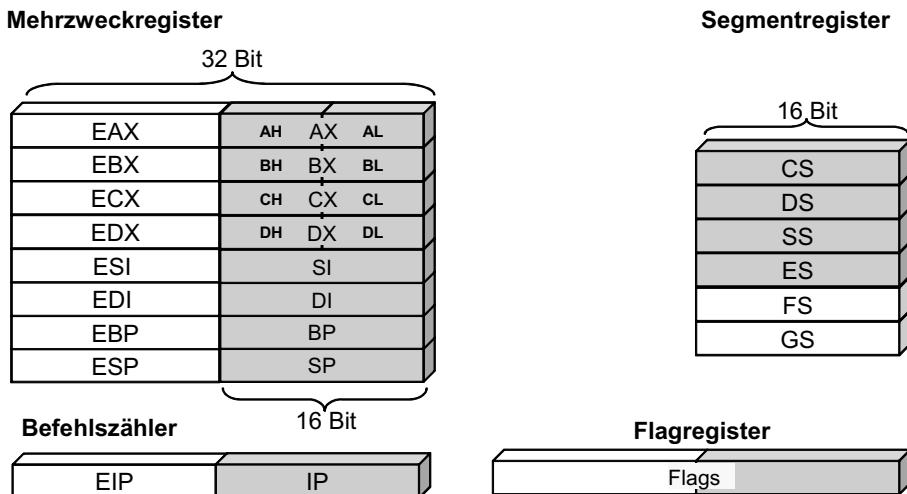


Abb. 5.92: Register von PC-Prozessoren im 32-Bit Betrieb

Die Abbildung zeigt die wichtigsten Register für die Ganzzahlarithmetik, die Speicheradressierung und die Programmlogik. In diesem Bild nicht gezeigt sind die 80-Bit breiten Spezialregister zur Verarbeitung von Gleitkommazahlen, die 64-Bit breiten MMX-Register für schnelle graphische Operationen und die 128-Bit breiten so genannten XMM SIMD Register.

Beim Übergang auf die 64-Bit Architektur, zunächst durch AMD und später auch durch Intel wurden die Mehrzweckregister, das Flag Register und das Befehlszählerregister erneut erweitert und durch das Buchstabenpräfix „R“ gekennzeichnet. Im 64-Bit Modus stehen dem Programmierer die Register RAX, RBX, ... RSP zur Verfügung. Die niederwertigen Teile davon können mit den älteren Bezeichnungen EAX, AX, AH, AL, etc. angesprochen werden. Neben diesen Registern stehen acht neue 64-Bit Mehrzweckregister mit den Bezeichnungen R8, .. , R15 zur Verfügung.

Alle neueren Intel und AMD Prozessoren können wahlweise als 64-Bit Rechner oder im älteren 32-Bit Modus betrieben werden. Im 64-Bit Modus werden ältere 32-Bit Programme in einem Kompatibilitätsmodus ausgeführt. Von dieser Möglichkeit macht auch das Betriebssystem Windows 7 Gebrauch.

### 5.7.3 Assemblerbefehle

In Intel Syntax haben die meisten Assemblerbefehle, die Operationen beschreiben, die Form *Op Ziel, Quelle*.

*Ziel* und *Quelle* werden mit der Operation *Op* verknüpft und das Ergebnis in *Ziel* gespeichert. In Java-Notation entspricht dies einer Zuweisung: *Ziel = Ziel Op Quelle*. Je nach Befehl

können Ziel und Quelle Register oder Speicherplätze sein. Als Quelle kommen auch konstante Werte in Frage. Beispiele solcher Befehle sind `add RAX, R12`, `add EAX, EBX` oder `sub AX, 5`.

Daten liegen entweder als Konstanten oder als Inhalte von Registern und Speicherzellen vor. Die Interpretation der dort gespeicherten Bitfolgen bleibt dem Programmierer überlassen, es gibt keine Typüberprüfung. Der Assemblierer kann lediglich feststellen, ob die Breiten der verknüpften Register zueinander passen. Die meisten Operationen sind mit 64-, 32-, 16- oder 8-Bit-Registern durchführbar, doch muss die Datenbreite von Quelle und Ziel stets übereinstimmen.

In Assembler schreibt man stets einen Befehl pro Zeile. Zwischen Groß- bzw. Kleinschreibung wird nicht unterschieden. Ein Semikolon beginnt einen Kommentar, der sich bis zum Zeilenende erstreckt. Das folgende Assemblerfragment benutzt die arithmetischen Operationen `ADD`, `SUB`, `INC`, `DEC` und `NEG` auf den Mehrzweckregistern EAX bis EDX sowie deren 16-Bit und 8-Bit-Teilregistern AX bis DX, AH bis DH und AL bis DL. Der Befehl `MOV` transportiert einen Wert von Quelle nach Ziel. Die Wirkung jedes Befehls wird in Java-ähnlicher Notation in einem Kommentar erklärt.

```

add AH, AL ; AH = AH+AL
mov AL, CL ; AL = CL
dec CL ; CL = CL-1
add EAX, 3E8h ; EAX = EAX+1000
inc ECX ; ECX = ECX+1
neg ECX ; ECX = -ECX

```

Während das Ziel einer arithmetischen Operation immer ein Speicherplatz oder ein Register sein muss, bezeichnet die Quelle immer einen Wert: den Inhalt eines Registers oder Speicherplatzes oder auch eine Konstante. Diese kann dezimal oder in hexadezimaler Notation (kurz Hex) angegeben sein. Hex-Notation erreicht man durch ein nachgestelltes `h` oder eine vorangestellte `0`.

#### 5.7.4 Mehrzweckregister und Spezialregister

Alle Mehrzweckregister können als Ziel von arithmetischen Operationen (dazu gehört auch der `mov`-Befehl) dienen. Dennoch erfüllen ESI, EDI, EBP, ESP noch besondere Aufgaben, so dass es sinnvoll und üblich ist, sich für arithmetische Berechnungen auf EAX - EDX zu beschränken. Die Spezialregister, dazu gehören die *Segmentregister* SS, DS, CS, ES, FS und GS sowie der *Instruction Pointer EIP* und das *Flag-Register*, können nicht oder nur eingeschränkt Ziel arithmetischer Operationen sein. In ein Segmentregister kann man nicht unmittelbar konstante Werte übertragen. Um etwa `100h` nach DS zu bringen, muss man den Umweg über ein Mehrzweckregister in Kauf nehmen:

```

mov AX, 100h
mov DS, AX.

```

Das Register *EIP* enthält stets die Adresse des nächsten Befehls. Es ist demnach nicht möglich, dort Daten zu speichern. *EIP* wird entweder automatisch erhöht, oder durch Sprungbefehle, dazu gehören auch Funktionsaufrufe und -rücksprünge, verändert.

### 5.7.5 Flag-Register

Das *Flag-Register* ändert sich nach arithmetischen Operationen. Es dient dazu, spezielle Situationen bei der Durchführung einer ALU-Operation, anzuzeigen. Es ist eigentlich ein Ausgaberegister, dennoch kann es auf dem Umweg über den später zu besprechenden Stack gezielt verändert werden.

Im Flag-Register hat jedes einzelne Bit seine eigene Bedeutung. Von den 32 Bits des Flag-Registers werden nur 14 verwendet, uns interessieren hier nur 9 davon. Deren Namen sind in der folgenden Tabelle mit einem Kurzkommentar aufgelistet. Die Flags *C, A, O, S, Z, P* beziehen sich immer auf das Ergebnis einer gerade durchgeföhrten Operation.

Die Flags *D, I* und *T* dienen als Schalter. Sie bleiben unverändert, bis man sie durch Spezialbefehle verändert. Die so genannte *direction flag, D*, beeinflusst die Wirkungsweise von String-Befehlen. Ist *D* gesetzt, so werden Strings von links nach rechts, andernfalls von rechts nach links abgearbeitet. Die Befehle STD (set direction) und CLD (clear direction) setzen die *D-Flag* auf 1 bzw. auf 0. Entsprechend bestimmt die *interrupt-enable flag, I*, ob der Prozessor auf gewisse Unterbrechungen (z.B. auf die Tastatureingabe) reagieren soll oder nicht.

Schließlich wird die *trap flag* von Programmen wie einem *Debugger* verwendet. Ein Debugger erlaubt die schrittweise Ausführung eines Maschinenprogramms zu Testzwecken. Damit ergibt sich eine Zweiteilung der Flags in solche, die eigentlich als Schalter dienen, und solche, die Ergebnisse von Operationen erläutern:

<i>C</i>	<i>Carry</i>	Bereichsüberschreitung für vorzeichenlose Zahlen,
<i>A</i>	<i>Aux. Carry</i>	Bereichsüberschreitung für vorzeichenlose Nibbles,
<i>O</i>	<i>Overflow</i>	Bereichsüberschreitung bei arithmetischer Operation auf Zahlen mit Vorzeichen,
<i>S</i>	<i>Sign</i>	Ergebnis negativ,
<i>Z</i>	<i>Zero</i>	Ergebnis 0,
<i>P</i>	<i>Parity</i>	Ergebnis hat gerade Anzahl von Einsen,
<i>D</i>	<i>Direction</i>	bestimmt Richtung von String Befehlen,
<i>I</i>	<i>Interrupt</i>	bestimmt, ob Interrupts zugelassen werden,
<i>T</i>	<i>Trap</i>	erlaubt single step modus. Vom Debugger verwendet.

Von den Ergebnisflags sind sowohl *Z* als auch *P* offensichtlich zu interpretieren. *Z = 1* bedeutet, dass das Ergebnis der letzten arithmetischen Operation 0 war, und *P = 1* bedeutet, dass das Ergebnis der letzten Operation gerade viele Einsen hatte. Diese Information ist für die Datenübertragung manchmal nützlich, wenn zu übertragende Daten mit einem zusätzlichen *Prijsbit* so aufgefüllt werden, dass das zu übertragende Datum eine gerade Anzahl von Einsen hat. Wird ein Wort mit einer ungeraden Anzahl von Einsen empfangen, so erkennt man, dass ein Fehler eingetreten ist.

Nicht alle Operationen beeinflussen alle Flags, so dass evtl. auch später noch durch eine frühere Operation erzeugte Spezialbedingungen aus dem Flag-Register ablesbar sind. In diesem Zusammenhang muss man aber wissen, ob die seither durchgeföhrten anderen Operationen die fraglichen Flags nicht beeinflusst haben. Eine fundierte Kenntnis von Maschinensprache beinhaltet daher auch das Wissen, welche Operation welche Flag beeinflusst.

### 5.7.6 Arithmetische Flags

Die Flags, *C*, *A*, *O*, und *S* beziehen sich auf die Interpretation der beteiligten Daten als ganze Zahlen. Der Prozessor weiß nicht, ob die Inhalte von Registern als natürliche Zahl (*unsigned number*) oder als ganze Zahl in Zweierkomplement-Darstellung (*signed number*) gemeint sind (siehe dazu das Kapitel über Zahlendarstellungen, S. 17ff.). Für die einfachen arithmetischen Operationen ist diese Information auch nicht notwendig, da das Ergebnis in beiden Fällen durch die gleiche Bitfolge dargestellt wird. Allerdings kann in der einen Interpretation das Resultat ungültig sein und in der anderen Interpretation nicht. Für beide Fälle werden vorsorglich die richtigen Flags gesetzt. Für die Deutung als vorzeichenlose Zahl zeigt das *Carry-Bit*, ob ein Übertrag aus der höchsten Bitposition entstanden ist bzw. ob das Ergebnis negativ und daher als vorzeichenlose Zahl ungültig ist. Für die Interpretation als vorzeichenbehaftete Zweierkomplementzahl zeigt das *overflow flag* eine Bereichsüberschreitung oder -unterschreitung an. *Zero flag* und *sign flag* zeigen, ob das Ergebnis der letzten Operation 0 oder negativ war.

Wir demonstrieren beide Sichtweisen an einigen konkreten Beispielen. Im ersten Fall werden die Register AL und BL mit den Bytes 0FD und 0FF gefüllt und addiert. Als Ergebnis entsteht im Register AL das Byte 0FC. Als Zweierkomplementzahlen interpretiert, haben wir  $-1$  zu  $-3$  in Register AL addiert. Das Ergebnis  $-4$  ist korrekt, da wir den Bereich  $-128 \dots + 127$  der 8-Bit-Zweierkomplementzahlen nicht verlassen haben. Konsequenterweise ist das O-Flag nicht gesetzt. Interpretieren wir dieselben Daten als vorzeichenlose natürliche Zahlen, so wird zu 253 in AL die Zahl 255 aus BL addiert. Von dem Ergebnis,  $253 + 255 = 508$ , passen nur die niedrigsten 8 Bit in das Zielregister AL, also  $508 \bmod 256 = 252$ . Das Carry-Bit zeigt an, dass ein Übertrag aus der höchsten Bitposition entstanden ist. Als Addition von natürlichen Zahlen ist das Ergebnis also ungültig. Aus den Befehlsgruppen im folgenden Bild erzeugt der Assembler in der Tat jeweils identische Maschinenbefehle. Die richtige Interpretation muss der Programmierer liefern und dafür die entsprechenden Flags beachten.

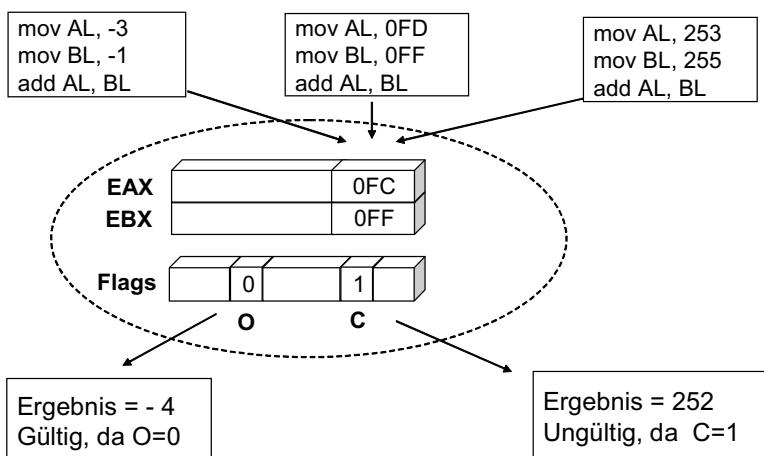


Abb. 5.93: Maschinenbefehle und ihre Wirkung auf das Flag-Register (1)

Als zweites Beispiel betrachten wir die Befehle:

```
mov AL, 100
add AL, AL
```

Diese Befehlsfolge ist identisch mit:

```
mov AL, 064
add AL, AL
```

In jedem Fall befindet sich am Ende 0C8 im Register AL. Interpretiert man die Berechnung als vorzeichenbehaftete 8-Bit-Addition, so findet man das Ergebnis -56 in AL und das Overflow-Bit gesetzt. Es hat eine Bereichsüberschreitung stattgefunden. Als natürliche Zahl betrachtet stellt 0C8h gerade 200 dar. Es hat während der Addition keine Bereichsüberschreitung stattgefunden, weswegen das C-Bit nicht gesetzt wurde.

Dieselben Überlegungen gelten auch für die Subtraktion. Beispielsweise ist

```
mov AH, 02
mov BH, 0FF
sub AH, BB
```

sowohl als vorzeichenbehaftete Subtraktion  $2 - (-1) = 3$  als auch als Subtraktion von ganzen Zahlen  $2 - 255$  interpretierbar. In jedem Falle enthält AH den Wert 3, das Carry-Bit zeigt aber an, dass die Operation für vorzeichenlose Zahlen ungültig war.

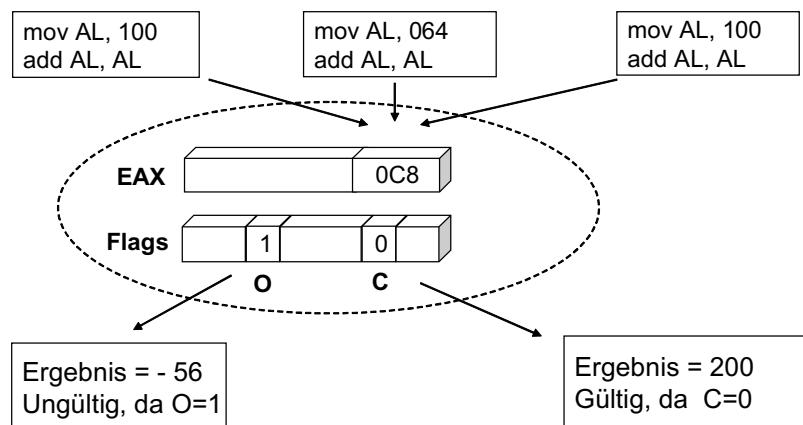


Abb. 5.94: Maschinenbefehle und ihre Wirkung auf das Flag-Register (2)

### 5.7.7 Größenvergleiche

Die Subtraktion mit anschließender Flag-Prüfung kann man verwenden, um Registerwerte der Größe nach zu vergleichen. Es stellt sich aber ein kleines Problem: Ist der Hex-Wert 02 kleiner oder größer als 0FF? Als vorzeichenlose Zahl gilt  $02 = 2$  und  $0FF = 255$ , also  $02 < 0FF$ . Als Zweierkomplementzahl gilt  $02 = 2$  und  $0FF = -1$ , also  $02 > 0FF$ .

Die Frage, ob für zwei Registerinhalte  $X$  und  $Y$  die Relation  $X < Y$  gilt, hängt also davon ab, wie wir  $X$  und  $Y$  interpretieren. Demgemäß unterscheidet man auf Hex-Zahlen zwei Ordnungen: *below* und *less*. Für beliebige Registerinhalte  $X$  und  $Y$  sagt man

- $X \text{ below } Y \Leftrightarrow X < Y$  als vorzeichenlose natürliche Zahl,
- $X \text{ less } Y \Leftrightarrow X < Y$  als Zweierkomplementzahl.

Die entsprechenden inversen Relationen heißen *above* bzw. *greater*. Vorzeichenlose Zahlen muss man also mit *above/below* und vorzeichenbehaftete Zahlen mit *greater/less* vergleichen. Die Ordnungen stimmen überein, wenn man kleine positive Zahlen vergleicht, also 8-Bit Zahlen kleiner als 128, 16-Bit Zahlen kleiner als 32768 oder 32-Bit-Zahlen kleiner als  $2^{31}$ .

Ob eine der Ordnungsrelationen zutrifft, erkennt man direkt nach einer Subtraktion  $X - Y$  an den Flags:

- $X \text{ below } Y \Leftrightarrow$  Bei der Subtraktion ist ein Übertrag aufgetreten, also  $C = 1$ .
- $X = Y \Leftrightarrow$  Z-Flag gesetzt, also  $Z = 1$ .
- $X \text{ above } Y \Leftrightarrow$  sonst, also  $C = Z = 0$ .

Für die Interpretation als vorzeichenbehaftete Zahlen gilt entsprechend:

- $X \text{ less } Y \Leftrightarrow X - Y$  negativ und  $O = 0$  oder  $X - Y$  positiv und  $O = 1$ , kurz:  $S \neq O$ .
- $X = Y \Leftrightarrow$  Z-Flag gesetzt, also  $Z = 1$ .
- $X \text{ greater } Y \Leftrightarrow$  sonst, also  $Z = 0$  und  $S = 0$ .

Bei einer Subtraktion von  $X = 0FFh$  und  $Y = 06h$  in einem 8-Bit-Register werden die folgenden Flags setzt:  $C = 0$ ,  $Z = 0$ ,  $O = 0$ ,  $S = 1$ . Damit gilt:  $X \text{ above } Y$  und gleichzeitig  $X \text{ less } Y$ . In der Tat gilt vorzeichenlos:  $X = 255$ ,  $Y = 6$  und somit  $X > Y$ . Als 8-Bit-Zweierkomplementzahlen gilt dagegen:  $X = -1$ ,  $Y = +6$  und  $X < Y$ . Findet die gleiche Subtraktion in einem 16-Bit-Register statt, so gilt  $X \text{ above } Y$  und  $X \text{ greater } Y$ , da  $X$  als 16-Bit-Zweierkomplementzahl  $+255$  darstellt.

Da für einen Vergleich nur die Flags nach der Subtraktion eine Rolle spielen, nicht aber das Ergebnis, gibt es eine Operation *cmp*, die genau das Nötige leistet. Die Operation

`cmp Ziel, Quelle`

setzt die Flags wie die entsprechende *SUB*-Operation ohne den Inhalt von *Ziel* zu verändern. Je nach Ausgang einer Vergleichsoperation kann z.B. verzweigt werden.

Die arithmetischen Operationen, *INC* und *DEC* dienen zum Inkrementieren bzw. Dekrementieren eines Speicher- oder Registerinhaltes um 1. Die *INC*- und *DEC*-Versionen sind schneller und lesbarer als die entsprechenden *ADD*- und *SUB*-Befehle und werden oft in Schleifen benötigt. *INC* und *DEC* verändern jedoch nicht das Carry-Flag. Das ist insbesondere deswegen nicht von Nachteil, weil man die Bedingung auch anders testen kann: *ADD Ziel, 1* setzt genau dann das Carry-Flag, wenn das Ergebnis 0 ist, d.h. wenn auch das Z-Flag gesetzt wird. *SUB Ziel, 1* setzt genau dann das Carry-Flag, wenn vorher *Ziel = 0* war. Auch dies ist leicht feststellbar. Somit verzichten *INC* und *DEC* auf das Setzen des Carry-Flags, was für die Programmierung von Schleifen oft von Vorteil ist.

### 5.7.8 Logische Operationen

Die logischen Operationen *AND*, *OR*, *XOR*, *NOT* funktionieren prinzipiell wie die arithmetischen und auch mit denselben Registern. Die meisten logischen Operationen setzen das Carry-Flag auf 0. Die Flags *S*, *Z* und *P* werden je nach Ergebniswert gesetzt.

Die Bedeutung dieser Operationen bedarf kaum einer Erläuterung. Sie werden bitweise ausgeführt. Beispielsweise ist

$$\text{AND } 7 \ 13 = \text{AND } 00000111b \ 00001101b = 0101b = 5.$$

Häufig werden logische Operationen für Zwecke benutzt, die nicht unmittelbar klar sind:

```
; Setze Register AX auf 0
xor AX, AX
; Vertausche den Inhalt der Register AX und BX
xor AX, BX
xor BX, AX
xor AX, BX
```

Mit *AND* und *OR* kann man einzelne Bits in einem Wort löschen oder setzen. Dazu benutzt man als zweiten Operanden eine *Maske*. Das ist eine konstante Bitfolge, die an den aus- oder einzublendenen Bits eine 1 besitzt. Die Maske wird gern als Binärzahl, erkenntlich an dem nachgestellten *b*, geschrieben.

```
; Setze Bit 2 und Bit 7 von AL auf 1
or AL, 0100 0010b
; Setze alle Bits außer Bit 2 und Bit 7 von BL auf 0
and BL, 0100 0010b
```

Die Operation *TEST* setzt alle Flags wie der entsprechende *AND*-Befehl, lässt aber die Operanden intakt. Somit verhält sich *TEST* zu *AND* wie *CMP* zu *SUB*.

```
; Prüfe, ob Bit 2 oder Bit 7 in AH gesetzt sind
test AH, 0100 0010b
; jetzt sollte man die Z-Flag überprüfen
```

Zum Verständnis von Assemblerbefehlen gehört also nicht nur das Wissen um das Ergebnis einer ausgeführten Operation, sondern auch um deren Einfluss auf die Flags. Die folgende Tabelle fasst dies noch einmal zusammen. Auf die Operationen *mul* und *div* werden wir später noch eingehen.

---

<i>add, sub, neg</i>	beeinflussen	<i>O, S, Z, C, P, A</i>
<i>inc, dec</i>	beeinflussen	<i>O, S, Z, P, A</i>
<i>mul, div</i>	beeinflussen	<i>O, C</i>
<i>and, or, xor</i>	beeinflussen	<i>S, Z, P</i> und setzen <i>C = 0</i>
<i>cmp</i>	setzt die Flags wie	<i>sub</i>
<i>test</i>	setzt die Flags wie	<i>and</i>

---

### 5.7.9 Sprünge

Assemblerbefehle werden in der Reihenfolge ausgeführt, in der sie im Text erscheinen, es sei denn, es handelt sich um einen *Sprungbefehl*. Ein solcher bewirkt die Fortsetzung des Programms an einer beliebigen, durch einen Namen markierten Stelle. Die Sprungmarke (engl. *label*) ist das Argument des Sprungbefehles.

Es gibt unbedingte Sprünge, bei denen der Sprung auf jeden Fall stattfindet, und *bedingte Sprünge*, bei denen er nur erfolgt, falls eine bestimmte Bedingung erfüllt ist. Die Bedingung wird immer anhand des Flag-Registers überprüft.

*JMP* ist der unbedingte Sprungbefehl (Jump). *JZ* steht für *Jump on zero*, der nur ausgeführt wird, falls das Zero-Flag gesetzt ist. *JNZ* (Jump if not zero) wird ausgeführt, falls das Zero-Flag nicht gesetzt ist. Meist folgt ein solcher Sprungbefehl auf einen Vergleich oder auf eine arithmetische oder logische Operation. Die Befehle *JE* (jump on equal) und *JNE* (jump on not equal) sind identisch zu *JZ* und *JNZ*. Der Assembler erzeugt jeweils identischen Maschinencode.

```
; Berechne den ggT von AX und BX
mov ax, 504      ; Anfangswerte
mov bx, 210      ; -- " --
schleife:
    cmp ax, bx      ; Vergleich
    jz ausgabe      ; Bedingter Sprung
    jb AX_below_BX ; Bedingter Sprung
    sub ax,bx      ; AX = AX - BX
    jmp schleife   ; unbedingter Sprung

AX_below_BX:
    sub bx, ax      ; BX = BX-AX
    jmp schleife   ; unbedingter Sprung

ausgabe:
```

Abb. 5.95: Schleife mit bedingtem Sprung

Wichtig sind die auf einem Größenvergleich basierenden Sprünge. Auch sie erfolgen üblicherweise im Anschluss an einen CMP-Befehl. Je nachdem, ob Registerinhalte als vorzeichenlose oder vorzeichenbehaftete Zahlen aufgefasst werden sollen, muss eine der Ordnungen *above/below* oder *greater/less* geprüft werden. Die mnemonischen Formen *JA* (jump above), *JB* (jump below), *JG* (jump greater), *JL* (jump less) entledigen den Programmierer der Mühe, sich genau zu überlegen, welche Flags zu überprüfen sind.

Sprungbefehl	Bedeutung	Flag-Bedingung
JA	Jump Above	C = 0 and Z = 0
JAE	Jump Above or Equal	C = 0

Sprungbefehl	Bedeutung	Flag-Bedingung
JB	Jump Below	C = 1
JBE	Jump Below or Equal	C = 1 or Z = 1

Abb. 5.96: Sprünge, basierend auf dem Vergleich vorzeichenloser Zahlen

Sprungbefehl	Bedeutung	Flag-Bedingung
JG	Jump Greater	S = O and Z = 0
JGE	Jump Greater or Equal	S = O
JL	Jump Less	S ≠ O
JLE	Jump Less or Equal	S ≠ O or Z = 1

Abb. 5.97: Sprünge, basierend auf dem Vergleich vorzeichenbehafteter Zahlen (O ist O-Flag)

### 5.7.10 Struktur eines vollständigen Assemblerprogrammes

Mit den elementaren arithmetischen Operationen und Sprüngen können wir erste sinnvolle Assemblerprogramme schreiben. Ein lauffähiges Assemblerprogramm benötigt noch zusätzliche Hinweise (Direktiven), deren genaue Syntax von dem gewählten Assemblierer abhängen. Im Falle des Freeware-Systems *masm32* sind dies u.a.:

```
.386
.model flat, stdcall
option casemap :none
```

Es soll hier Code für einen 386-Prozessor (oder später) erzeugt werden. Man geht von einem linearen (flachen) Speichermodell aus, wobei Code und Daten in dem gleichen Speichersegment liegen. Funktionsaufrufe erwarten ihre Parameter in umgekehrter Reihenfolge auf dem Stack (*stdcall*) und Sprungmarken sowie Funktionsnamen sind case-sensitiv. Soll das Programm unter Windows lauffähig sein und Windows Ressourcen anfordern, so müssen die benötigten Datentypen und Prozeduren des Betriebssystems dem Assembler bekannt gemacht werden. Dies geschieht durch *include*-Direktiven

```
include \masm32\include\windows.inc
include \masm32\include\kernel32.inc
includelib \masm32\lib\kernel32.lib
```

Oft werden noch weitere nützliche Bibliotheksprogramme auf diese Weise geladen, denn auch Assemblerprogrammierer wollen das Rad nicht neu erfinden.

Das Programm selber besteht aus *Segmenten*, in denen Speicherplatz für Daten reserviert und strukturiert wird und aus Segmenten, die den Code enthalten. Die entsprechenden Teile werden jeweils durch die Schlüsselworte *.data* bzw. *.code* eingeleitet. Nicht jedes Programm benötigt ein Datensegment. Das Codesegment muss mindestens eine Marke besitzen, bei der

die Programmausführung beginnen soll. Diese wird dadurch gekennzeichnet, dass sie nach dem Schlüsselwort `end` wiederholt wird.

Nach seiner Beendigung soll das Programm die Kontrolle wieder an das Betriebssystem zurückgeben. Zu diesem Zweck ruft es die Bibliotheksfunktion `ExitProcess` auf, deren Parameter 0 vorher mit `push 0` auf dem Stack abgelegt wurde.

Das Codesegment eines mit `masm32` erstellten und unter Windows lauffähigen Assemblerprogramms sieht dann folgendermaßen aus:

```
.code
main:
    ; ... hier kommt der Programmcode hin ...
    push 0           ; Argument 0
    call ExitProcess ; Funktionsaufruf - zurück zu Windows
end main
```

Nachdem das Programm mit einem Editor erstellt und in einer Datei `ggT.asm` abgespeichert wurde, kann es assembled werden. Es entsteht zunächst eine Objekt-Datei, unter Windows mit der Endung `.obj`. Diese muss noch durch einen so genannten *linker* mit den nötigen Bibliotheksfunktionen zu einer ausführbaren `exe`-Datei verbunden werden.

### 5.7.11 Ein Beispielprogramm

Im *Datensegment* des folgenden Beispielprogramms, das mit dem Schlüsselwort `.data` beginnt, werden die Variablen `Rahmentxt` und `Fenstertxt` als Bytefolgen erklärt und mit Anfangswerten vorbelegt:

```
.data
Rahmentxt    db "Gruss von Windows", 0
Fenstertxt   db "Das Ergebnis ist : "
ergebnis      db 5 DUP (0)
```

Für `ergebnis` werden 5 Byte mit Inhalt 0 reserviert. `db` steht hierbei für *define byte*. Analog gibt es `dw`, `dd`, `dq` für *word*, *double word* und *quad word*. Eine Direktive `n DUP (x)` veranlasst den Assembler, *n* viele Speicherplätze zu reservieren und mit dem Wert *x* vorzubereiten. Der Speicherplatz wird hintereinander im Speicher angelegt. Die eingeführten Namen sind genau genommen als Marken im Datensegment zu verstehen, d.h. als Adressen relativ zum Anfang des Datensegments. Im obigen Fall haben wir also  
`Rahmentxt: 0`, `Fenstertxt: 18=12h`, `ergebnis: 37=25h`.

Die Angabe, ob es sich um Byte, Word, DoubleWord oder QuadWord Formate handelt, dient nur zur Vermeidung logischer Fehler. So wird sich der Assembler weigern, ein `mov Rahmentxt, ax` oder ein `mov Rahmentxt, eax` zu assemblen, weil (E)AX eine 16(32)-Bit Größe enthält, nicht ein Byte. Sollte der Programmierer aber darauf bestehen, muss er es mit der Direktive *Word Ptr* bzw. *DWord Ptr* klarstellen, also etwa  
`mov DWORD PTR Rahmentxt, eax`.

Im *Code-Segment*, das mit dem Schlüsselwort `.code` beginnt, wird zuerst in Register EAX der ggT von 504 und 210 berechnet und dann das Ergebnis, das bei Beendigung der Schleife in EAX als Hex-Wert (2Ah) vorliegt, mit Hilfe der Funktion `dtoa` (double word to ascii) aus der *masm32*-Bibliothek in eine Dezimalzahl (42) umgerechnet und als Folge von ASCII-Zeichen „4“, „2“ in `ergebnis` abgelegt. Anschließend rufen wir die Betriebssystemfunktion `MessageBox` auf, die die Startadressen zweier Strings verlangt.

```
invoke dwtoa, eax, addr ergebnis
invoke MessageBox, 0, addr Fenstertxt, addr Rahmentxt, MB_OK
```

Strings enden automatisch mit dem ersten NULL-Byte (00h), weshalb der erste String explizit mit 0 beendet wurde. `Fenstertxt` wurde nicht mit 0 abgeschlossen, daher endet der dort beginnende String mit der ersten 0, die in den 5 Bytes von `ergebnis` gefunden wird, was in der `MessageBox` die Ausgabe „Das Ergebnis ist : 42“ bewirkt.

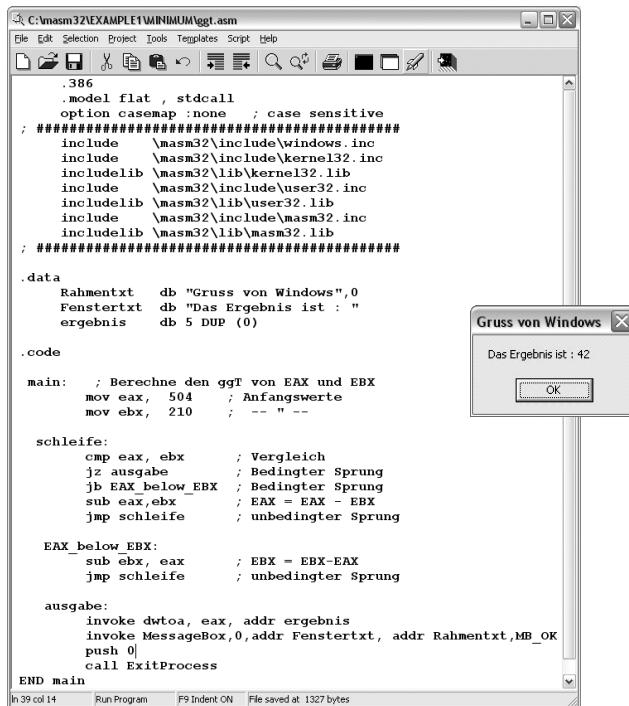


Abb. 5.98: Komplettes Assemblerprogramm und Ergebnis des Aufrufs unter Windows XP

Heutige Benutzeroberflächen, so auch der zu *masm32* gehörende *Quick Editor*, verbinden das Assemblieren und Linken zu einem einzigen Menübefehl. Die Abbildung zeigt ein komplettes Programm, das den ggT zweier Zahlen berechnet und das Ergebnis in einer Windows MessageBox ausgibt. Der Menüpunkt *Project>Build All* assembliert und verlinkt das Programm zu einer ausführbaren *exe*-Datei, die mit *Projekt>Run* sofort gestartet werden kann.

Zur Ausführung von `ggT.exe` wird die Datei vom Betriebssystem in den Speicher geladen. Die physikalische Adresse der genannten Variablen ergibt sich dann durch Addition mit der Adresse an der Anfang des Datensegments im Speicher zu liegen kommt.

### 5.7.12 Testen von Assemblerprogrammen

Auch sorgfältig programmierte Assemblerprogramme funktionieren selten auf Anhieb. Syntaxfehler werden bereits vom Assembler erkannt. Logische Fehler, dazu gehören auch Endlosschleifen, stellen sich erst zur Laufzeit heraus. Oft hilft es, gewisse Programmteile schrittweise durchzugehen und dabei die Wirkung der einzelnen Instruktionen auf die Register und auf den Speicher zu verfolgen. Diesen Zweck erfüllen *Debugger*. Kommerzielle Debugger, wie z.B. *SoftIce* sind nicht ganz billig, für den Anfang tut der frei erhältliche *Turbo Debugger 32* von Borland gute Dienste.

Die folgende Abbildung zeigt *TD32* bei der Inspektion von `ggT.exe`. Nachdem das Programm in den Debugger geladen wurde, kann es mit der Taste *F8* schrittweise ausgeführt werden. In der Mitte erkennt man die Darstellung des Programmcodes, im rechten Fenster die Register mit ihren Inhalten und am rechten Rand die wichtigsten Flags. Der Programmzähler steht gerade bei `jb`, der fünften Instruktion. Die Register `EAX` und `EBX` enthalten die Werte `54h` bzw. `D2h`, weswegen der Vergleich `cmp eax, ebx` soeben das Vorzeichen-Flag `s` (sign) und das Übertrag-Flag `c` (carry) gesetzt hat. Als Nächstes steht der Sprung `jb` an, der aufgrund der gesetzten `c`-Flag auch ausgeführt werden wird.

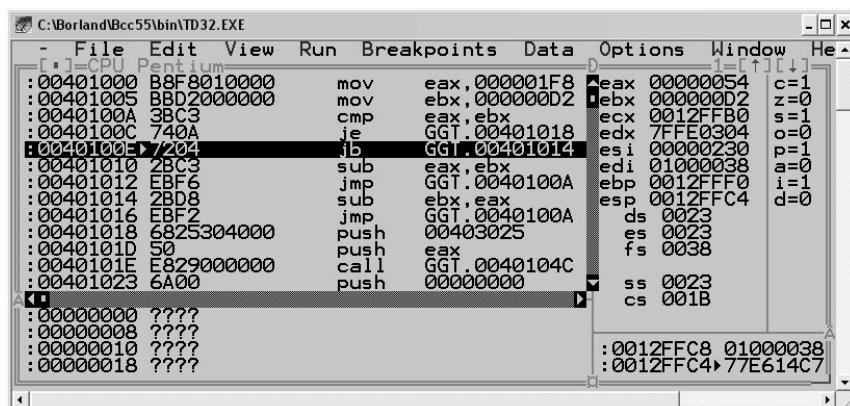


Abb. 5.99: Das *ggT*-Programm im Debugger

Der Maschinencode für den Sprung besteht aus den beiden Bytes `72h` und `04h`. Davon ist `72h` der eigentliche Sprungbefehl und `04h` die Sprungweite. Es wird also um 4 Byte nach vorne gesprungen werden, gezählt vom Beginn der folgenden Instruktion. Analog wurde der unbedingte Sprung `jmp schleife` des ursprünglichen Assemblerprogramms übersetzt in `EB F2`. Hier steht `EBh` für den unbedingten Sprung und `F2h` für `-14`, also einen Sprung um 14

Byte zurück. In der vordersten Spalte des linken Fensters erkennt man auch die Speicheradressen, an denen die einzelnen Maschinencodes gespeichert sind.

Mit einem Debugger kann man jedes ausführbare Programm schrittweise mitverfolgen, gegebenenfalls auch verändern und zurückschreiben. Letzteren Prozess nennt man auch *disassemblieren*. Einfachere Programme für die Erkundung und Veränderung fremder Programme heißen auch *Disassembler*. Da beim Assemblieren die Marken und die Namen der aufgerufenen Routinen verlorengehen, ersetzt der Debugger diese durch automatisch generierte Namen. Man kann das Quellprogramm durch geeignete Parameter auch so übersetzen, dass die vom Programmierer definierten Namen und Marken in einer *Symboleitabelle* aufbewahrt werden und vom Debugger verwendet werden können, was eine Fehlersuche im Debugger deutlich erleichtert.

### 5.7.13 Speicheradressierung

Unter frühen PC-Betriebssystemen war es kein Problem, direkt auf eine bestimmte Zelle im Hauptspeicher lesend oder schreibend zuzugreifen. Unter Windows oder Linux sind solche direkten Speicherplatzzugriffe nicht erlaubt, da sie andere Programme oder das Betriebssystem stören könnten. Jedes Benutzerprogramm erhält einen eigenen Adressraum, in dem es Daten schreiben und lesen kann. Wie dieser Adressraum aber auf den physikalisch vorhandenen (oder nicht vorhandenen) Speicher abgebildet wird, ist Sache des Betriebssystems. Ein Benutzerprogramm kann im 32-Bit Betrieb einen Adressraum von bis zu 4 GB erhalten, im 64-Bit Betrieb je nach CPU Typ 64 GB oder mehr. Um physikalische Geräte oder Ports anzusteuern, muss man sich der Funktionen des Betriebssystems bedienen. Das ist ohnehin der bequemere Weg.

Natürlich muss man auch unter Windows den Hauptspeicher verwenden, allerdings bestimmt das Betriebssystem, in welchen physikalischen Speicherzellen die Benutzerdaten abgelegt werden. Der Benutzer verwaltet einen virtuellen Hauptspeicherbereich, welchen er durch die Datendefinitionen *db*, *dw*, *dd*, *dq* strukturiert.

Angenommen, eine Bank definiert Daten für Konten und Transaktionsnummern (TAN)

```
.data
KontoNr    dd  123987
TAN        dw  6734, 1067, 2945, 1981, 5511,
```

durch welche KontoNr als 32-Bit Wert mit Inhalt 123987 und TAN als Liste von fünf 16-Bit Werten definiert werden. In dem Datensegment bezeichnet dann KontoNr die Adresse 0 und TAN die Adresse 4. Die folgenden Daten der TAN-Liste beginnen an den Adressen 6, 8, 10 und 12. Obwohl KontoNr und TAN Adressen sind, dürfen wir mit ihnen fast so umgehen wie mit Variablen in höheren Programmiersprachen:

```
mov eax, KontoNr
```

lädt die KontoNr in Register EAX, wobei der Assemblierer überprüft, dass die Variablengröße *double* mit der Länge des EAX-Registers (32 Bit) übereinstimmt.

```
mov TAN, ax
```

ersetzt die erste TAN durch den Inhalt von Register AX. Die folgenden TAN können wir durch ihre Speicheradressen TAN+2, TAN+4, ... ,TAN+8 ansprechen. Alternativ ist auch die Notation TAN [2],TAN [4], ... ,TAN [8] zugelassen.

Um beispielsweise die dritte und die fünfte TAN zu vertauschen, könnte man schreiben:

```
mov ax, TAN [4]
mov bx, TAN [8]
mov TAN+8, ax
mov TAN+4, bx
```

Bei dem MOV-Befehl dürfen nicht sowohl Quelle als auch Ziel Speicheradressen sein – mov TAN [4], TAN [8] ist also nicht erlaubt. Allerdings würde eine vermutlich fehlerhafte Anweisung wie mov TAN [5], ax vom Assembler akzeptiert, was den dritten und den vierten TAN-Wert der Liste verändern würde. Selbst mov KontoNr [4], eax würde klaglos akzeptiert, obwohl durch den Befehl ein Teil der TAN-Liste überschrieben würde.

In dem gerade betrachteten Fall waren die korrekten relativen Speicheradressen bereits zur Assemblierzeit bekannt. Meist werden die Adressen aber erst zur Laufzeit berechnet, so beispielsweise, wenn wir alle Werte der TAN-Liste um eins erhöhen. Wir können das durch eine kleine Schleife erledigen. Dazu benötigen wir allerdings die so genannte *indirekte Adressierung*, wobei der Index der TAN-Liste in dem Register EBX (mnemonisch für *base index*) berechnet wird. Das Register EBX spielt hierbei eine Sonderrolle. Neben einfachen Indexangaben wie TAN [bx] oder alternativ [TAN+bx] sind auch einfache konstante Ausdrücke wie z.B. [TAN+2\*ebx-2] zugelassen. Zur Illustration folgt ein kurzes Programm, das jede unserer TAN-Nummern um eins erhöht:

```
mov ebx, 5
naechste:
    mov ax, TAN [2*ebx-2]
    inc ax
    mov TAN [2*ebx-2], ax
    dec ebx
    jne naechste
```

### 5.7.14 Operationen auf Speicherblöcken

Die 80x86-Familie besitzt spezielle Operationen und Schleifenmechanismen, um ganze Datenblöcke im Speicher zu verschieben oder zu vergleichen. Die Register *ESI* (extended Source Index) und *EDI* (extended Destination Index) müssen zur Adressierung von Quelle und Ziel der Datenbewegung verwendet werden. Die Befehle zur Stringverschiebung (*MOV*S<sub>x</sub>) und zum Stringvergleich (*CMP*S<sub>x</sub>) dienen zur direkten Bewegung bzw. zum Vergleich von Daten ohne Umweg über ein Register. *x* steht hier für die Datengröße, muss also durch B, W, D oder Q ersetzt werden. So kopiert beispielsweise der Befehl *MOVS*B ein Byte von der Adresse in ESI zur Adresse in EDI. Je nachdem, ob das *direction flag* D gesetzt ist oder nicht, werden automatisch ESI und EDI erniedrigt oder erhöht, so dass ein erneuter Befehl *MOVS*B das nächste Byte kopiert.

Die String-Befehle erlauben zusätzlich noch *Wiederholungspräfixe*. REP wiederholt den folgenden Befehl, bis das ECX Register 0 ist und dekrementiert jedesmal ECX. Auf diese Weise

kann man sehr effizient Datenblöcke verschieben. Im folgenden Beispiel wird ein Block von 6 Byte, der an Adresse MyOS beginnt, an Adresse Rahmentxt+10 verschoben. *Offset* ist eine Assembler-Direktive, die die Adresse einer Marke im Datensegment berechnet.

```
.data
    Rahmentxt    db "Gruss von Windows", 0
    MyOS         db "Linux", 0
.code
beispiel:
    mov esi, offset MyOS           ; Quelladresse setzen
    mov edi, offset Rahmentxt+10   ; Zieladresse setzen
    cld                          ; Richtung: aufsteigend
    mov ecx, 6                   ; Anz. d. Wiederholungen
    rep movsb                   ; While cx>0 MOVSB
end Beispiel
```

Für die Vergleichsoperationen CMPSx sind die Wiederholungspräfixe REPZ bzw. REPNZ nützlich, die den Vergleich solange ausführen, wie CX nicht 0 ist und das Zero-Flag gesetzt bzw. nicht gesetzt ist. Die Richtung von Datenbewegungen oder Vergleichen lässt sich mit CLD (clear direction flag) und STD (set direction flag) festlegen.

### 5.7.15 Multiplikation und Division

Wir kehren nun zur Besprechung der beiden noch fehlenden arithmetischen Grundoperationen zurück, der Multiplikation und der Division. Von dem Format *Op Ziel, Quelle* muss man hier abweichen, da das Ergebnis im Allgemeinen nicht in das Zielregister passen würde. Daher dienen, je nach Größe der Faktoren, spezielle Register zur Aufnahme des Produktes. Für die Multiplikation von 8-Bit-Zahlen wird ein Operand im Register AL erwartet, der andere Operand in einem 8-Bit-Mehrzweckregister oder Speicher. Das Ergebnis von „*MUL Operand*“ steht dann als 16-Bit-Größe in AX. Umgekehrt kann man eine 16-Bit-Zahl in AX durch einen 8-Bit-Operanden dividieren: „*DIV Operand*“. Der Quotient liegt danach in AL, der Rest in AH.

```
mov al, 17
mov dl, 30
mul dl    ; ax := al * dl
```

```
mov ax, 37
mov bl, 3
div bl
; al := ax div bl
; ah := ax mod bl
```

Für die Multiplikation von 16(32)-Bit-Zahlen wird ein Operand in (E)AX erwartet, der andere 16(32)-Bit-Operand in einem Mehrzweckregister oder dem Speicher. Das Ergebnis von *mul Operand* steht dann als 32(64)-Bit-Größe in (E)DX:(E)AX, d.h. die höherwertigen Bits in (E)DX, die niedrigwertigen in (E)AX. Umgekehrt kann man eine solche 32-Bit-Zahl in (E)DX:(E)AX durch einen 16(32)-Bit Operanden dividieren. Der Quotient liegt danach in (E)AX, der Rest in (E)DX.

```
mov ax, 1001
mov cx, 30
mul cx
;   dx:ax = 30030
```

```
mov ax, 1001
mov dx, 0
mov cx, 15
div cx
;   ax = 66
;   dx = 11
```

Im Gegensatz zu Addition und Subtraktion funktionieren Multiplikation und Division bei vorzeichenlosen Zahlen anders als bei vorzeichenbehafteten Zahlen. MUL und DIV arbeiten auf vorzeichenlosen, d.h. natürlichen Zahlen. Für vorzeichenbehaftete oder ganze Zahlen muss man die entsprechenden Befehle IMUL (integer multiply) bzw. IDIV (integer divide) verwenden.

### 5.7.16 Shift-Operationen

Multiplikation und Division sind relativ aufwändige Operationen. Für spezielle Fälle, etwa Multiplikation mit einer Zweierpotenz, kann man auch die Shift- bzw. Rotate-Operationen benutzen. Diese Operationen verschieben den Inhalt eines Registers um eine oder mehrere Bitposition nach links oder nach rechts. Dabei fällt rechts bzw. links ein Bit aus dem Register, an dem anderen Ende entsteht eine Lücke, die mit irgendeinem Bit aufgefüllt werden muss.

In der Art, wie diese herausfallenden bzw. zu füllenden Bitpositionen zu behandeln sind, unterscheiden sich die verschiedenen Shift- bzw. Rotate-Versionen. Bei den Shift-Operationen gelangt ein herausfallendes Bit in das Carry-Flag. Die Shift-Instruktionen sind:

<b>SHR</b>	<i>; Shift unsigned right,</i>
<b>SHL</b>	<i>; Shift unsigned left,</i>
<b>SAR</b>	<i>; Shift arithmetic right,</i>
<b>SAL</b>	<i>; Shift arithmetic left.</i>

Bei den ersten beiden Operationen, SHR und SHL, wird die jeweils entstehende Lücke mit einer 0 gefüllt. Diese Operationen führen also eine Halbierung bzw. Verdopplung ihres Argumentes durch. Sei beispielsweise in AL die Zahl  $183 = (1011011)_2$  gespeichert.

```
SHR AL, 1
```

ändert den Inhalt von AL zu 01011011b, was als vorzeichenlose Zahl  $(01011011)_2 = 91$  darstellt. Am Inhalt des Carry-Flag, C = 1, erkennt man, dass ein Rest bei der Division durch 2 entstanden ist. Ein

```
SHL AL, 1
```

hätte AL zu 01101110b gesetzt. Die am weitesten links stehende 1 wäre in die Carry-Flag gewandert, welche so angezeigt hätte, dass die Multiplikation mit 2 den zulässigen Bereich überschritten hat.

Teilt man eine negative ganze Zahl durch 2, so ist zu berücksichtigen, dass die am weitesten links stehende Bitposition als Vorzeichen dient. Bei einem Rechts-Shift sollte sie nicht einfach durch 0 aufgefüllt werden, sondern ihren alten Wert behalten. Daher gibt es die Variante für vorzeichenbehaftete Zahlen, SAR, shift arithmetic right.

```
SAR AX, 1
```

teilt eine ganze Zahl in AX durch 2. Die Operation SAL ist identisch mit SHL.

Der zweite Operand einer Shift-Operation gibt die Anzahl der auszuführenden Shifts an. Er muss entweder eine konstante Zahl oder das Register CL sein. Entsprechendes gilt für die Rotate-Operationen. Bei diesen wird das herausgeschobene Bit benutzt, um die am anderen Ende entstandene Lücke zu füllen. Die Rotate-Operationen sind:

<b>ROR</b>	<i>; rotate right,</i>
<b>ROL</b>	<i>; rotate left,</i>
<b>RCR</b>	<i>; rotate through carry right,</i>
<b>RCL</b>	<i>; rotate through carry left.</i>

In den letzten beiden Versionen, RCL und RCR, wird das Carry-Bit in die Rotation einbezogen. Das Carry-Bit füllt die Lücke, und das herausfallende Bit wandert in das Carry.

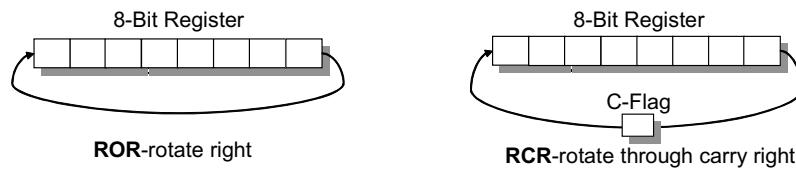


Abb. 5.100: Rotate-Befehl und Carry-Flag

Der folgende Code multipliziert eine positive 32-Bit-Zahl in DX : AX mit 2:

```
SHL AX, 1
RCL DX, 1
```

Die Shift- und Rotate-Operationen sind um ein Vielfaches schneller als die entsprechenden Multiplikationen oder Divisionen.

### 5.7.17 LOOP-Befehle

Mit den Sprungbefehlen kann man beliebige *while*-, *repeat*- und *for*- Schleifen nachbilden. Die 80x86-Prozessorfamilie besitzt aber zusätzliche Operationen, um dies effizienter und leserbarer zu gestalten. In allen diesen Befehlen wird das CX Register als Schleifenzähler benutzt. Es enthält die Anzahl der verbleibenden Iterationen.

Man kann sich C als Abkürzung für *counter* einprägen. Der Befehl LOOP dekrementiert CX und springt an den Anfang der Schleife, falls CX ≠ 0 ist. Es wird keine Flag verändert, insbesondere auch nicht die Z-Flag gesetzt. Um im Falle, dass vor Beginn der Schleife schon CX = 0 ist, gleich an deren Ende zu springen, gibt es den Sprungbefehl JCXZ (jump if CX is zero). Abgesehen von den Flags ist folgende *for*-Schleife

```
MOV CX, k
JCXZ Fertig
```

```
Schleife:  
    Befehl1  
    . . .  
    Befehln  
    LOOP Schleife  
Fertig: . . .
```

äquivalent zu dem etwas umständlicheren Code:

```
MOV CX, k  
CMP CX, 0  
JZ Fertig  
Schleife:  
    Befehl1  
    . . .  
    Befehln  
    DEC CX  
    JNZ Schleife  
Fertig:
```

Es gibt weitere LOOP-Befehle, die wir hier nicht weiter besprechen wollen. LOOPZ und LOOPNZ terminieren, wenn entweder CX = 0 oder die Zero-Flag 1 bzw. 0 ist.

### 5.7.18 Der Stack

Der Stack ist hauptsächlich für die Ausführung von Unterprogrammen erforderlich. Bei jedem Aufruf wächst der Stack, bei jedem Rücksprung schrumpft er wieder. Im Allgemeinen muss der Benutzer den Stack nicht explizit manipulieren.

Dennoch gibt es Situationen, in denen die Stack-Operationen PUSH und POP auch dem Programmierer nützlich sind. Eine typische Situation tritt auf, wenn Register für eine Zwischenrechnung gebraucht werden, ihre alten Inhalte aber aufbewahrt werden müssen. Es ist z.B. eine Konvention der Win32-API-Programmierung, dass die Register EAX, ECX und EDX in den Bibliotheksfunktionen verändert werden können, während EBX, ESI und EDI erhalten werden sollen. Wird also eine solche API-Funktion aufgerufen und soll aber der gegenwärtige Inhalt von EAX und ECX gerettet werden, so empfiehlt es sich, den Inhalt dieser Register auf dem Stack zu retten:

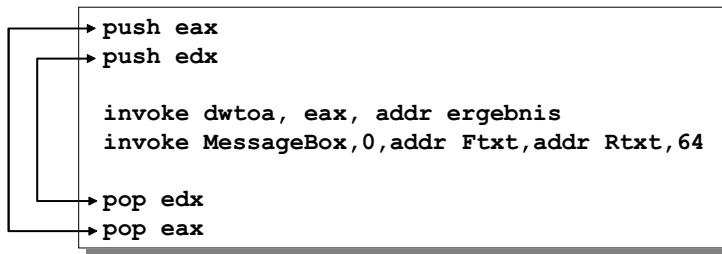


Abb. 5.101: Verwendung des Stack

Ganz analog wird der Programmierer einer Bibliotheksfunktion vorgehen. Falls er etwa das Register EBX oder BX benötigt, wird er den alten Wert mit `push ebx` speichern und ihn vor Ende der Funktion wieder mit `pop ebx` restaurieren.

### 5.7.19 Einfache Unterprogramme

Auch in Assembler kann man strukturiert programmieren. Ein wesentliches Hilfsmittel dazu bietet der Prozedur-Mechanismus. In der einfachsten Ausprägung besteht dieser aus zwei Assembler-Instruktionen: CALL und RET. Ein Unterprogramm ist dann Assembler-Code, der mit einer Sprungmarke *marke*: beginnt und der Anweisung RET endet. Der Aufruf des Unterprogramms geschieht mit dem Befehl CALL *marke*.

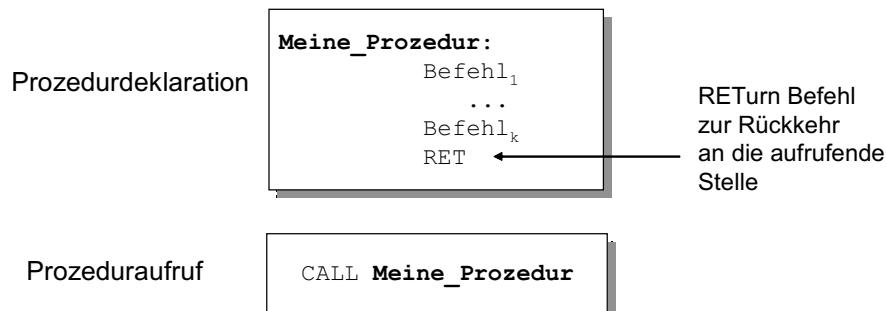


Abb. 5.102: Prozedurdeklaration und -aufruf

In unserem vorigen ggT-Beispiel hatten wir eine mysteriöse Funktion dwtoa aufgerufen, die den Wert des EAX-Registers als ASCII-String in der Variablen *ergebnis* ablegte. Wir wollen ein ähnliches Unterprogramm *toAscii* selber programmieren. Wir benötigen ein weiteres Unterprogramm *letzteZiffer*, um die letzte Dezimalziffer von EAX zu berechnen und als ASCII-Zeichen in DL zu speichern. Der Quotient EAX/10 liegt danach wieder in EAX.

```

letzteZiffer:           ; erwartet Zahl in eax und liefert
                        ; ASCII der letzten Dezimalziffer in dl
                        ; Quotient in eax
xor edx, edx            ; 32-Bit Division vorbereiten
mov ecx,10              ; Quotient
div ecx                 ; eax := edx:eax / ecx
add dl,'0'              ; '0' = 48
ret

toAscii:                ; schreibt Dezimalwert von eax als ASCII-
                        ; String der Länge 5 in "ergebnis"
                        ; String mit 0 terminieren
mov ebx,5
mov ergebnis[ebx],0
dec ebx
vorigeZiffer:
call letzteZiffer       ; letzte Ziffer berechnen
mov ergebnis[ebx],dl
dec ebx                 ; schreiben
dec ebx                 ; zurück
jge vorigeZiffer        ; nochmal
ret

```

Abb. 5.103: Assemblerprogramm mit Prozeduren

Der Unterprogramm-Mechanismus ist technisch erstaunlich einfach zu realisieren. Der Aufruf des Unterprogramms

```
call letzteZiffer
```

führt zu zwei Aktionen: zunächst wird der Programmzeiger IP auf den Stack gelegt und anschließend mit einem unbedingten Sprung jmp letzteZiffer verzweigt. Das Unterprogramm selbst endet mit dem Befehl:

```
ret
```

Dieser bewirkt ein POP des obersten Stackwertes in den Programmzähler IP. Dies hat zur Folge, dass die Berechnung mit der Instruktion fortgesetzt wird, die auf das zuletzt ausgeführte CALL-Kommando folgt.

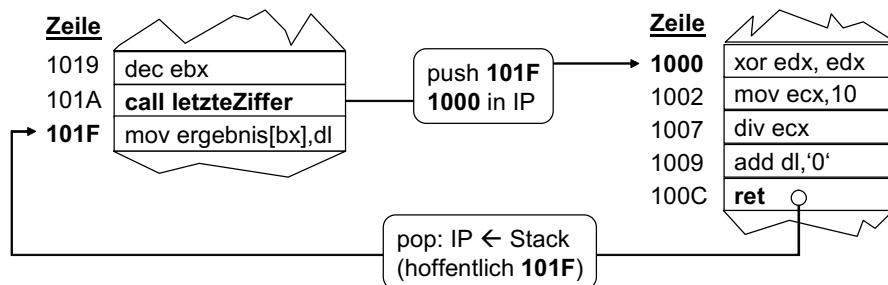


Abb. 5.104: Der CALL-RET-Mechanismus

Der Mechanismus funktioniert auch bei verschachtelten Prozeduraufrufen dank des last-in-first-out Mechanismus des Stacks: der zuletzt abgelegte Wert wird als erster wieder entfernt.

### 5.7.20 Parameterübergabe und Stack

Bisher haben wir nur parameterlose Unterprogramme gezeigt. Benötigt ein solches aber Parameter, so gibt es mehrere Möglichkeiten, diese zu übergeben. Am einfachsten ist es, sie in bestimmte Register zu schreiben und dann das Unterprogramm aufzurufen. Dieses kann die Argumente dann den entsprechenden Registern entnehmen. Analoges gilt für Rückgabewerte. So hatten wir es bei der Funktion `toAscii` praktiziert, die ihr Argument in EAX erwartet und das Ergebnis in DL ab liefert.

Für längere Parameterlisten oder für Array-Parameter wäre dies zu unübersichtlich oder gar unmöglich. Statt dessen bietet es sich an, den Stack zu nutzen. Vor dem Aufruf des Unterprogramms werden die Argumente auf den Stack gelegt. Danach kommt der Aufruf. So kommt die Rücksprungadresse über den Argumenten zuoberst auf dem Stack zu liegen. Um an die Parameter heranzukommen, muss das Unterprogramm daher in den Stack hineinschauen können, ohne diesen mit POP zu verändern. Genau für diese Zwecke gibt es das *EBP*-Register. Es dient zum indizierten Zugriff auf Daten im Stacksegment, ähnlich wie EBX einen indizierten Zugriff im Datensegment ermöglicht. Da das Register *ESP* (extended stack pointer) stets auf den aktuellen top des Stacks zeigt, kann man EBP folgendermaßen initialisieren:

```
mov EBP, ESP
```

Sodann findet man bei [EBP] die Rücksprungadresse, bei [EBP + 2] das zuletzt abgelegte Argument, bei [EBP + 4] das vorletzte etc., wenn wir der Einfachheit halber 16 Bit große Parameter annehmen.

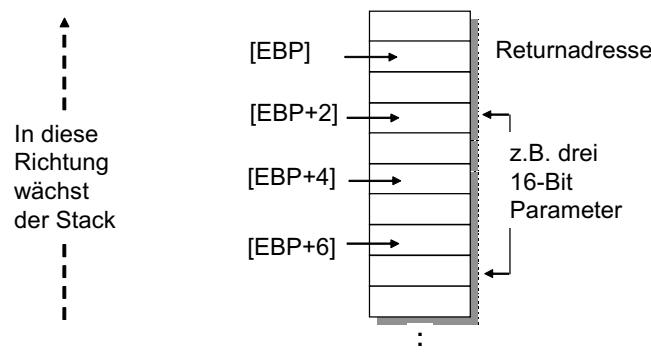


Abb. 5.105: Parameterübergabe mittels Stack

Nach Beendigung des Aufrufes müssen alle Argumente wieder von dem Stack entfernt werden. Dies erreicht man am bequemsten mit dem Befehl `RET k`, wobei *k* die Anzahl der Bytes angibt, die zusätzlich zur Rücksprungadresse vom Stack entfernt werden müssen. Selbstverständlich bedeutet *Entfernen vom Stack* lediglich, SP neu zu berechnen.

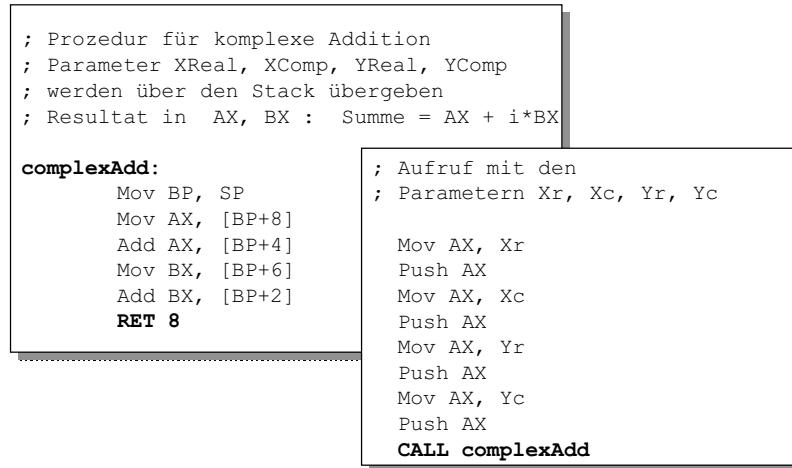


Abb. 5.106: Prozedur mit Parametern – Definition und Aufruf

### 5.7.21 Prozeduren und Funktionen

Es ist klar, dass der Zugriff auf die Parameter knifflig und fehlerträchtig ist. Daher haben alle Assembler einen *Prozedurmechanismus*, der die Sache erleichtert. Die Parameter bekommen einen Namen, über den sie im Programm referenziert werden können, und eine Länge. Beim Assemblieren werden die Namen durch entsprechende Stackadressen ersetzt. Das vorige Beispiel wird so deutlich übersichtlicher:

```

complexAdd PROC XReal:WORD, XComp:WORD, YReal:WORD, YComp:WORD
    MOV AX, XReal
    ADD AX, YReal
    MOV BX, XComp
    ADD BX, YComp
    RET
complexAdd ENDP

```

*PROC* und *ENDP* sind so genannte *Pseudooperationen*. Man kann sich vorstellen, dass *PROC <Argumente>* zunächst in elementaren Assemblercode expandiert und dieser danach in Maschinensprache übersetzt wird. Zusätzlich zu den Parametern kann man mit der Direktive *LOCAL* auch noch lokale Variablen deklarieren. Zur Ausführungszeit der Prozedur befinden diese sich dann ebenfalls auf dem Stack.

Prozeduren dürfen selber den Stack benutzen, sie müssen diesen aber am Ende so verlassen, wie er aufgefunden wurde. Der *PROC*-Mechanismus sorgt dafür, dass am Ende alle Argumente wieder vom Stack verschwunden sind, obwohl der Benutzer die Funktion mit einem einfachen *RET* beendet.

### 5.7.22 Makros

Die einfachste Form eines *Makros* (engl.: *macro*) ist eine Abkürzung eines Textteiles durch ein Schlüsselwort. Jedes spätere Erscheinen des Schlüsselwortes wird von dem Assembler vor der Übersetzung automatisch *expandiert*, d.h. durch den ungekürzten Text ersetzt. Im Allgemeinen lassen Makros auch Parameter zu. Sie haben dann eine große Ähnlichkeit zu Prozeduren. Allerdings existiert der Code für eine Prozedur nur einmal in dem Programm. Bei jedem Aufruf wird an die Stelle, an der sich der Code befindet, verzweigt. Bei Makros wird dagegen jeder Aufruf durch eine Kopie des Makro-Textes ersetzt. Die Expansion von Makros kann ein mehrstufiger Prozess sein, weil Makro-Aufrufe auch geschachtelt sein können.

Ein Beispiel eines in Masm32 schon vordefinierten Makros ist *invoke*, der es erlaubt, Funktionen mit Parametern fast wie in Hochsprachen aufzurufen. Angenommen, wir hätten unsere Funktion `toAscii` als Prozedur mit folgenden Parametern deklariert:

```
toAscii PROC Wert:DWORD, laenge:BYTE
```

Vor einem Aufruf müssen zuerst die Parameter auf den Stack gebracht werden:

```
push 5
push eax
call toAscii
```

Wir bauen nun einen Makro, um beliebige Funktionen mit zwei Argumenten bequemer aufrufen zu können:

```
rufe2 MACRO Funktion, Arg1, Arg2
    push Arg2
    push Arg1
    call Funktion
ENDM
```

Ohne an den Stack zu denken, können wir nun Funktionen mit zwei Parametern bequem in einer Zeile aufrufen:

```
rufe2 toAscii, eax, 5
```

Der Aufruf bewirkt, dass vor dem Assemblieren der Makro `rufe2` expandiert wird, wobei `Funktion`, `Arg1` und `Arg2` durch `toAscii`, `eax` und `5` ersetzt werden.

Die Möglichkeiten von *Masm32*, Makros zu erstellen, sind vielfältig, man kann sogar von einer Makro-Sprache reden, deren Darstellung unseren Rahmen sprengen würde. Nicht umsonst steht *masm* für „Macro Assembler“. Eine Reihe von Makros, wie z.B. `.if - .elseif - .else` oder `.while - .endw` sind bereits vordefiniert, so dass sich auch Assemblerprogramme sehr übersichtlich gestalten lassen. Makros können lokale Variablen und lokale Daten- und Codesegmente haben. Auch *invoke* ist ein solcher vordefinierter Makro, der, anders als unser bescheidener `rufe2`, beliebig viele Parameter zulässt.

## 5.8 RISC-Architekturen

Zu Beginn der 80er Jahre wurde der Begriff *RISC* geprägt. Diese Abkürzung steht für ein CPU-Konzept mit einem *reduzierten Befehlssatz (Reduced Instruction Set Computer)*. Mit dieser Begriffsbildung wurden gleichzeitig die bis dahin verwendeten Konzepte für die Konstruktion von CPUs mit dem gegenteiligen Begriff *CISC (Complex Instruction Set Computers)* belegt. Das Ziel der RISC-Philosophie war es, die CPU-Architektur an neuere Entwicklungen der zugrunde liegenden Hardware-Technik anzupassen. Die Grundidee war, einen Maschinenbefehl nicht durch ein Mikroprogramm zu implementieren, sondern ihn direkt durch einen einzigen Mikrobefehl ausführen zu lassen.

Anfang der 80er Jahre entstanden Prototypen zur RISC-Technologie (Stanford MIPS, Berkeley RISC und IBM 801). Die ersten darauf aufbauenden kommerziellen Produkte waren nicht besonders erfolgreich (Beispiel: IBM R/6000 Serie). Ende der 80er Jahre begann dann die Blütezeit der RISC-Technologie. In den letzten Jahren gab es mehrere kommerziell und technisch erfolgreiche Produktreihen, die auf RISC-Prozessoren aufbauen:

- IBM RS/6000, PowerPC,
- DEC  $\alpha$ ,
- SUN Sparc, Ultra Sparc,
- SGI Iris Indigo, Crimson, Indy 2, O2, Challenger, Origin,
- HP PA.

### 5.8.1 CISC

Als besonders markante Vertreter von Computerfamilien, die auf CISC-CPUs aufbauen, gelten die Familien:

- /360, /370, ES9000 von IBM,
- VAX von DEC.

Aber auch die Mikroprozessoren der x86-Serie von Intel und der 680x0-Serie von Motorola müssen diesem CPU-Konzept zugeordnet werden, auch wenn die Hersteller Wert darauf legen, bei neueren Modellen weitgehend RISC-Konzepte zu berücksichtigen. Dies trifft z.B. auf die im nächsten Abschnitt diskutierten Prozessoren von Intel und AMD zu. Die den CISC-Prozessoren zugrunde liegenden Ideen charakterisieren die Situation Anfang der 60er Jahre und gehen aus von:

- einer relativ schnellen Arbeitsweise der CPU,
- einem relativ langsamen Arbeitsspeicher,
- einem sehr kleinen Arbeitsspeicher,
- einem sehr teuren Arbeitsspeicher.

Während die ersten beiden Punkte sich bis heute nicht wesentlich verändert haben, aber im Gegensatz zu früher durch Cache-Speicher im Wesentlichen kompensiert werden können, treffen die beiden letzten Punkte heute nicht mehr zu.

Aus damaliger Sicht war es jedoch erstrebenswert, möglichst wenige, dafür komplexe Maschinenbefehle zu verwenden, mit dem Ziel, Programme zu verkürzen und die Zahl der Speicherzugriffe zum Laden von Instruktionen zu minimieren.

Ermöglicht werden komplexe Maschinenbefehle durch die im vorletzten Abschnitt erläuterte *Mikroprogrammtechnik*. Nur durch die Einführung dieser zusätzlichen Abstraktionsschicht erhält man die Chance, eine große Zahl komplexer Befehle fehlerfrei in einer CPU zu realisieren. Die komplexen Maschinenbefehle werden als Einsprungpunkte in ein Mikroprogramm aufgefasst und von dem dort anzutreffenden Mikroprogramm gesteuert, durch eine relativ einfache CPU-Logik ausgeführt. Das Mikroprogramm kann vor der Konstruktion der CPU entworfen werden und mithilfe von Simulationsprogrammen ausgetestet werden.

Eine andere Motivation für die Verwendung von Mikroprogrammen erwuchs aus der Absicht, mit unterschiedlichen Mikroprogrammen für verschiedene, mehr oder weniger aufwändige CPU-Konstruktionen dieselbe Hard- bzw. Software-Schnittstelle in Form einer definierten Maschinenarchitektur anzubieten.

Ein Charakteristikum von CPUs in CISC-Architektur ist meist die Verwendung von relativ wenigen Registern (typisches Beispiel ist die x86-Familie von Intel), dafür aber die Möglichkeit zu direkter Speicheradressierung in praktisch allen Befehlen. Operanden können entweder aus einem Register oder direkt aus dem Speicher stammen. Diese Speicheradressierung und die große Menge komplexer Instruktionen werden häufig damit begründet, dass auf Basis eines solchen Designs die Generierung von Maschinencode durch einen Compiler einfacher wird.

### 5.8.2 Von CISC zu RISC

Einer der Ausgangspunkte des Übergangs zu RISC-Architekturen war die Untersuchung gängiger Compiler. Es wurden Statistiken bekannt, denen zufolge gängige Compiler einen großen Teil der komplexen Instruktionen überhaupt nicht verwendeten. Und auch dort, wo sie verwendet wurden, trugen die komplexen Instruktionen nur ca. 20 % zur Laufzeit des generierten Codes bei. Die übrigen Instruktionen sind dagegen so einfach, dass sie auch zur RISC-Philosophie passen.

Hinzu kam die Beobachtung, dass immer mehr Speicher innerhalb der CPU und im Arbeitsspeicher zur Verfügung stehen, so dass keine Notwendigkeit besteht, Instruktionen zusammenzustauen. Cache-Speicher verringern den zusätzlichen Zeitaufwand zum Laden von *mehreren* Instruktionen.

Eine der Maßnahmen zur Reduzierung des Platzbedarfs von CISC-Instruktionen war die Definition zahlloser Befehlsformate: So kann die Befehlslänge bei der x86-Familie von 1 bis 32 Byte variieren, wobei fast jeder Zwischenschritt möglich ist. Bei dem weitgehend unbekannten Prozessor iAPX 432 aus dem Jahr 1982 variierten Befehlsanfang und Befehlslänge nicht nur auf Byte-, sondern sogar auf Bitebene.

### 5.8.3 RISC-Prozessoren

RISC-Prozessoren sind gekennzeichnet durch:

- wenige einfache Befehle, die möglichst in einem Maschinentakt ausgeführt werden,
- wenige Befehlsformate, möglichst mit nur einer festen Befehlslänge,
- viele Mehrzweckregister,
- Speicherzugriffe nur über Load- bzw. Store-Befehle.

Letzteres bedeutet, dass Quelle und Ziel von Operationen nur Register, nie Hauptspeicher sein können. Werden Operanden aus dem Speicher benötigt, so müssen sie vorher durch einen gesonderten Load-Befehl in einem Register bereitgestellt werden.

Als typisches Beispiel hat der MIPS-Prozessor R3000 64 Maschinenbefehle, der Operationscode wird mit 6 Bit codiert. Es gibt drei Befehlsformate, eine Befehlslänge und 32 Mehrzweckregister. Die Anzahl der Register war bei den ersten Prototypen der RISC-Architekturen sehr verschieden:

- Der Stanford MIPS (Vorläufer der MIPS R Serien) hatte nur 16 Register.
- Der Berkeley RISC (Vorläufer der SPARC-Prozessoren) hatte 138 Register.
- Der IBM 801 (Vorläufer der IBM-POWER-Prozessoren) hatte 32 Register.

Heute gilt die Zahl von 32 Registern als guter Kompromiss. Die Effekte, die man mit einer größeren Anzahl von Registern erzielen wollte, insbesondere die Verringerung von Speicherzugriffen, erreicht man heute besser mit einem On-Chip-Cache.

Strittig ist noch die Frage, wie viele Befehle ein RISC-Prozessor haben soll. Die oben genannte Zahl von 64 Befehlen erscheint aus heutiger Sicht sehr einengend. Möglicherweise ist eine Codierung des Operationscodes durch 8 Bits sinnvoller. Diese würde bis zu 256 Befehle zulassen und damit eigentlich der ursprünglichen RISC-Philosophie widersprechen. Andererseits gingen ursprüngliche RISC-Entwürfe von einer Aufteilung der Funktionen auf mehrere Chips bzw. Prozessoren und Coprozessoren aus. Mit der heutigen Integrationsdichte erscheint diese Vorgehensweise nicht mehr zeitgemäß. Es ist daher sinnvoller, Gleitpunktoperationen durch eigene Maschinenbefehle anzusprechen und nicht über eine Coprozessor-schnittstelle abzuwickeln.

Nach wie vor umstritten ist die Reduktion des Speicherzugriffs auf Load- bzw. Store-Befehle. Daten können nur manipuliert werden, wenn sie sich in Registern befinden:

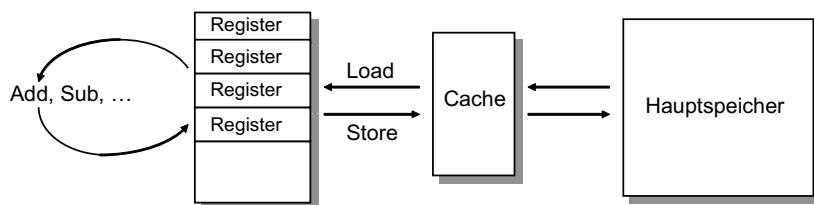


Abb. 5.107: Speicherhierarchien bei RISC-Prozessoren

Häufig vorkommende Befehle wie Load, Store, Add, Sub etc. werden möglichst schnell ausgeführt, d.h. in einem Maschinentakt und ohne Hilfe eines Mikroprogramms. Einer besonderen Optimierung bedarf es auch bei Sprungbefehlen, da diese sehr häufig vorkommen und wegen eines eventuell notwendigen Speicherzugriffs zum Laden der Zieladresse nicht in einem Takt erledigt werden können. Häufig findet man daher das Konzept des verzögerten Sprungs. Ein Sprungbefehl wird in einem Takt *abgearbeitet*, *springt* aber erst einen Takt später. Der Programmierer/Compiler hat Gelegenheit dies zu nutzen, indem er die Befehle so umsortiert, dass unmittelbar nach dem Sprungbefehl noch ein Befehl abgearbeitet wird, der logisch gesehen vor dem Sprung ausgeführt werden soll und die Sprungbedingung nicht beeinflusst. Falls ein solcher Befehl nicht gefunden werden kann, muss ein Noop-Befehl eingefügt werden, der nichts tut (No Operation).

Dieses Konzept der verzögerten Wirkung kann auch auf Load- und Store-Befehle angewendet werden, falls sich die Taktzeit auf diese Weise weiter reduzieren lässt.

Die ursprüngliche RISC-Philosophie forderte den gänzlichen Verzicht auf Mikroprogramme. Solange nur ganz wenige Befehle mehrere Takte benötigen (z.B. die Multiplikation und vor allem die Division), ließ sich das auch durchhalten. Heute werden wegen der hohen Integrationsdichte wieder zunehmend komplexere Befehle in der CPU ausgeführt, z.B. Gleitpunktbefehle, Bitblockbefehle, Multimediabefehle, so dass man von dem ursprünglichen Konzept wieder Abstand nimmt und lediglich fordert, dass nur *wenige* Befehle mithilfe von Mikroprogrammen abgewickelt werden. Der zusätzlichen Leistungssteigerung dienen heute zwei weitere Konzepte:

- Anwendung der Fließbandtechnik (Pipelining),
- Parallelisierung (Superskalar-Technik).

Diese Techniken steigern die Leistung der Prozessoren und nutzen die ungeheure Zahl von Transistorfunktionen, die in einem heutigen Chip potentiell zur Verfügung stehen. Im Jahr 2010 haben die höchstintegrierten Chips (8 GBit DRAMs) ca. 12.800.000.000 Transistorfunktionen. Die meisten bekannten CPU-Chips verwenden bisher aber weniger als 1.000.000.000. Der bereits erwähnte Prozessor des Core i7 980X wird seit März 2010 gefertigt und verfügt über 1,17 Milliarden Transistorfunktionen.

#### 5.8.4 Pipelining

Eine *Pipeline* ist eine Warteschlange, in der sich die als Nächstes abzuarbeitenden Befehle befinden. Jeder Befehl besteht aus einer Reihe von Phasen. Während noch die letzten Phasen der vorderen Befehle in der Pipeline abgearbeitet werden, kann bereits mit den ersten Phasen der nächsten Befehle begonnen werden.

Mithilfe der Pipeline-Technik lassen sich die Taktzeiten einer CPU weiter reduzieren, wobei angestrebt wird, dass die durchschnittliche Ausführungszeit eines Befehls nahe bei einem Takt liegt. Der Befehl wird in mehrere Phasen aufgeteilt, die nacheinander, aber gleichzeitig mit anderen Phasen anderer Befehle, in einer Pipeline ausgeführt werden. Während eine Phase eines Befehls bearbeitet wird, erledigt die Pipeline schon andere Phasen weiterer Befehle. Heute sind 5- bis 35-stufige Pipelines üblich. Bei einer 5-stufigen Pipeline könnte

die Phasen-Aufteilung für einen Register/Register-Befehl etwa folgendermaßen aussehen:

- 
- |     |                                |
|-----|--------------------------------|
| S1: | Befehlsbereitstellung          |
| S2: | Dekodieren des Befehls         |
| S3: | Lesen der beteiligten Register |
| S4: | ALU-Operation                  |
| S5: | Schreiben in das Ziel-Register |
- 

Dabei werden bis zu fünf Befehle gleichzeitig überlappend bearbeitet. Während die Ergebnisse des 1. Befehls noch in ein Register übertragen werden, wird bereits der 5. Befehl bereitgestellt, der 4. Befehl dekodiert usw.

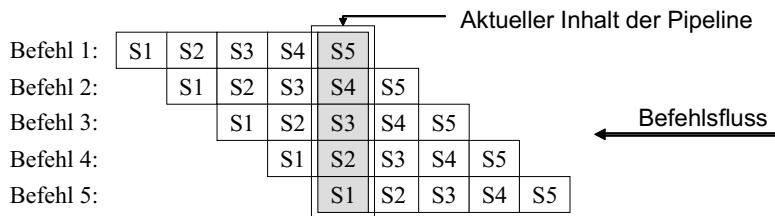


Abb. 5.108: Befehlsfluss in einer Pipeline

Die Verwendung einer Pipeline setzt voraus, dass zwischen den 5 beteiligten Befehlen keine störenden Zwischenbeziehungen existieren. Beispiel: Da der 2. Befehl seine Register bereits gelesen hat, dürfen diese nicht mit dem Register übereinstimmen, das der 1. Befehl noch schreiben will. Es gibt Techniken, solche Zwischenbeziehungen auf Hardwareebene zu entdecken. In einem solchen Fall muss die Pipeline zwischen den beteiligten Befehlen so lange angehalten werden bis Konsistenz vorliegt.

### 5.8.5 Superskalare Architekturen

Bei einer *superskalaren Architektur* kommen mehrere Pipelines parallel zum Einsatz. Bei heutigen CPUs sind dies 2 bis 10 – in Zukunft könnten es noch mehr werden.

Oft wird jeweils eine Pipeline für Integeroperationen und für Gleitpunktoperationen implementiert. Der Befehlsfluss wird zerlegt und, soweit das ohne Störung der Konsistenz der Daten möglich ist, auf die Pipelines verteilt:

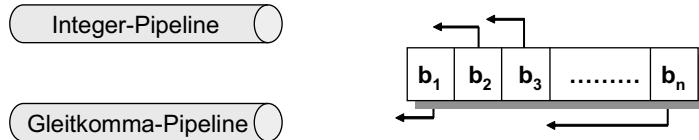


Abb. 5.109: Verteilung des Befehlsflusses auf mehrere Pipelines

### 5.8.6 Cache-Speicher

Ob CISC oder RISC, alle Rechnerarchitekturen sind heute mit aufwändigen Zwischenspeichern (*Caches*) ausgestattet. Der Grund dafür sind die immer höheren Prozessortaktaten bei wenig verbesserten Hauptspeicherzugriffszeiten. Typisch waren, bis vor wenigen Jahren, Zugriffszeiten von 50 bis 60 ns. Erst der Einsatz von SD-RAM-Bausteinen hat diese Zeiten in neueren Rechnern auf ca. 1 ns reduziert. Diesem Speichertyp stehen jetzt aber wiederum schnellere Prozessoren gegenüber. Daher ist es üblich geworden, eine Hierarchie von schnelleren Cache-Speichern zwischen den Prozessor und den Hauptspeicher zu schalten. Man spricht von einem *L1-Cache*, wenn dieser in das Prozessordesign voll integriert ist. Heute sind L1-Caches mit bis zu 64 kByte üblich. Zusätzlich wird meist ein weiterer Cache, der so genannte *L2-Cache*, verwendet. Dieser ist entweder direkt auf dem Prozessorchip untergebracht oder auf einer Platine als Einheit direkt mit dem Prozessor verbunden und hat eine typische Größe von 256 kByte bis 8 MByte. Einige Prozessoren besitzen dann noch einen *L3-Cache* unabhängig vom CPU-Chip auf dem Motherboard.

### 5.8.7 Leistungsvergleiche

Alle Hersteller behaupten, die jeweils schnellsten Prozessoren anzubieten. Objektive Leistungsvergleiche sind aber kaum möglich, weil die Charakteristika der Rechner zu unterschiedlich sind. Früher wurde die Leistung von Prozessoren häufig in der fragwürdigen Einheit *MIPS (million instruction per second)* angegeben. Diese Zahl kann aber, gerade bei Verwendung von Cache-Speichern, sehr unterschiedlich ausfallen, je nachdem ob man Befehle zählt, die aus dem L1-Cache (Level-1 Cache), dem L2-Cache (Level-2 Cache), dem L3-Cache (Level-3 Cache) oder dem Hauptspeicher geladen werden können bzw. müssen.

Wichtig ist auch, dass Leistungsvergleiche (engl.: *benchmark*) von Gremien durchgeführt werden, die unabhängig von den Herstellern sind. Im Folgenden beziehen wir uns auf die Benchmarksuite einer solchen Organisation namens *SPEC* (System Performance Evaluation Cooperative), die 1988 von einer kleinen Gruppe von Workstation-Herstellern gegründet wurde. Es wurden Testprogramme entwickelt und in verschiedenen Varianten zur Verfügung gestellt ([www.specbench.org](http://www.specbench.org)). Die aktuelle Version heißt *SPEC CPU2006* und ist unterteilt in *CINT2006* für Ganzzahlarithmetik und *CFP2006* für Gleitpunktoperationen.

Die Ergebnisse dieser Tests messen die Leistungsfähigkeit einzelner CPU Kerne. Sie schwanken leicht je nachdem auf welchem PC-System getestet wurde. Für den bereits erwähnten Prozessor Core i7-980X findet man CINT2006=37,2 und CFP2006=40,2. Um Sys-

teme mit mehreren CPU-Kernen zu bewerten, gibt es zwei weitere Benchmarks: *CINT2006rate* und *CFP2006rate*. Der gerade genannte Prozessor hat 6 CPU-Kerne und bringt es bei diesen Tests auf CINT2006rate=173 und CFP2006rate=115.

Leider sind die neueren Testprogramme CPU2006 mit den älteren CPU2000, CPU95 und CPU92 nicht direkt vergleichbar. Es gibt auch wenige Messungen der älteren und neueren Werte auf jeweils gleichen Rechnern. Optimal wäre es, die Anwenderprogramme, die für einen Interessenten relevant sind, auf verschiedenen Rechnern mit identischen Lastdaten laufen zu lassen. Da aber Anwenderprogramme meist nicht auf allen Plattformen ablauffähig sind, kann man auf diese Weise meist nur verschiedene Rechner einer Plattform vergleichen.

### 5.8.8 Konkrete RISC-Architekturen

**Die SUN-SPARC-Architektur:** Die SPARC-Architektur von SUN Microsystems baut auf dem Konzept des Berkeley RISC-Prototyps auf und wurde zwischen 1984 und 1987 entworfen. Sie sieht ein festes Befehlsformat, 69 verschiedene Befehle und sehr viele (bis zu 520) Register vor. Die Verarbeitung erfolgt in einer mehrstufigen Pipeline. Die ursprüngliche SPARC-Architektur war als 32-Bit-Architektur definiert. 2007 wurde von SUN der Prozessor Ultra-Sparc-T2 auf den Markt gebracht, der eine erweiterte 64-Bit-Architektur (V9) implementiert. Seitdem wurde von SUN kein neuerer Sparc-Prozessor angeboten. SUN wurde zwischenzeitlich von Oracle übernommen. Ob in dieser Konstellation neuere Sparc Generationen geplant sind, ist den Autoren nicht bekannt. Allerdings gibt es eine OpenSPARC Initiative, deren erklärtes Ziel es ist, die SPARC Architektur weiterzuentwickeln.

**Die IBM-POWER-Architektur:** Von Anfang an war IBM an der Entwicklung von RISC-Prozessoren beteiligt. Zunächst mit dem Prototypen IBM 801, dann mit einer ersten Workstation-Produktfamilie, die allerdings wegen ihres schlechten Preis-/Leistungsverhältnisses ein Misserfolg war, und seit Ende 1989 mit einer weiterentwickelten RISC-Architektur. Diese wird von IBM als POWER (*Performance Optimized With Enhanced RISC*) bezeichnet. Auf der Basis einer POWER-Architektur mit einem Chipsatz von bis zu 7 Millionen Transistoren bietet IBM die Workstationfamilie RS 6000 an. Im Rahmen einer Kooperation mit Apple und Motorola entwickelte IBM eine neue, leicht modifizierte Architektur namens PowerPC. Ziel war dabei, die POWER-Architektur auf einem einzigen, preisgünstigen Chip zu realisieren. Der erste derartige Chip wurde 1993 unter dem Namen MPC 601, alternativ PowerPC 601, vorgestellt. Bemerkenswert für diesen sind die geringe Chipfläche, der günstige Herstellungspreis und der geringe Energieverbrauch. Neuere Modelle mit höherer Leistung haben allerdings eine deutlich erhöhte Leistungsaufnahme. Die Modelle MPC 7450 und 7457 werden in Apple Macintosh Systemen mit der Bezeichnung G4 eingesetzt. Der Nachfolger dieses Rechners wurde PowerPC 970 benannt, baut auf dem Power4 Design auf, und wurde von Apple als G5 vermarktet.

	Chipfläche in mm <sup>2</sup>	Transistorzahl in Millionen	Energiever- brauch in Watt	Maximale Takt- frequenz
PowerPC 601	121	2.8	$\leq 10$	80 MHz
MPC 750	67	6.35	$\leq 6$	266 MHz
MPC 7450	106	33	$\leq 10$	867 MHz
PowerPC 970 G5	118	52	$\leq 42$	1,8 GHz
PowerPC 970GX	79	92,3	$\leq 85$	3 GHz

Der ursprüngliche PowerPC baut auf einem Befehlssatz mit ca. 180 Befehlen in einem einheitlichen Format auf. Der PowerPC 7457 verfügt über 32 Integer-Register mit jeweils 32 Bit, 32 Gleitpunktregister mit jeweils 64 Bit, einen L1-Cache von je 32 kByte für Daten und Befehle und einen L2-Cache mit 512 kByte, die beide im dem Prozessorgehäuse integriert sind. Ein weiterer L3-Cache wird unterstützt. Die Verarbeitung erfolgt in sechs Pipelines, die in bis zu 4 Stufen aufgeteilt sind. Diese werden wie folgt verwendet:

- eine Pipeline für die Bereitstellung folgender Instruktionen,
- eine Pipeline für Lade- und Speicherbefehle,
- zwei Pipelines für Integer-Arithmetik,
- eine Pipeline für Gleitpunktarithmetik.

Die Analyse der zu bearbeitenden Befehle erfolgt in einer mehrstufigen Queue. Die Befehle können unabhängig von ihrer tatsächlichen Reihenfolge in den Pipelines bearbeitet werden, wenn diese frei sind und zwischen den Befehlen keine Datenabhängigkeiten bestehen. Theoretisch können zu einem Zeitpunkt bis zu 10 Befehle gleichzeitig bearbeitet werden – praktisch ist es jedoch ziemlich unwahrscheinlich, dass eine Sequenz von 10 Befehlen vorkommt.

Im Vergleich zu anderen Prozessoren sind die Leistungsdaten der PowerPC Prozessoren seit dem Jahr 2000 eher bescheiden. Apple hat zunächst zusammen mit IBM einen neuen Prozessor entwickelt, der seit Ende 2003 unter der Bezeichnung G5 erhältlich ist. Apple hat diesen Rechner überwiegend mit zwei Prozessoren vermarktet, um mit den Leistungsdaten der Konkurrenz vergleichbar zu sein. Da sich bei diesem Prozessor auch in der Folgezeit nicht die erhofften Leistungsdaten einstellten, ist Apple von dieser Prozessorfamilie wieder abgerückt und setzt derzeit auf die Intel Core i Prozessorfamilie.

Im Jahr 2006 bewarb sich IBM erfolgreich um einen Forschungskontrakt zur Entwicklung eines neuen Hochleistungsrechners. Erste Produkte wurden im Februar 2010 angekündigt. Die Serie hat den Namen *Power 7*, bereits angekündigte Server Modelle haben die Bezeichnung Power 770 und Power 780. In diesen Servern sind Prozessorkarten verbaut, die jeweils aus 2 Power 7 Prozessorschips und mehreren DDR3-Speichermodulen bestehen. Jeder Power 7 Prozessor verfügt über 8 CPU Kerne, die mit einer Frequenz von 3,86 oder 4,14 GHz betrieben werden. Dieser Betrieb wird *TurboCore Modus* genannt. Bei der höheren Geschwindigkeit sind dann aber nur 4 CPU Kerne verfügbar. Jeder Kern verfügt über einen L1 Cache von 32 kB und einen L2 Cache von 256 kB. Der L3 Cache hat 32 MB und kann aufgeteilt werden

in 4MB pro CPU Kern oder von allen genutzt werden. Der Prozessorchip wird mit einem 45 nm Prozess gefertigt, belegt 567 mm<sup>2</sup> und beschäftigt ca. 1,2 Milliarden Transistoren. Offenbar kann man mit den Prozessorkarten verschiedene Konfigurationen mit einer unterschiedlicher Zahl von Prozessorkarten und CPU-Kernen zusammenstellen. Veröffentlicht wurden Leistungsdaten für spezifische Konfigurationen:

CINT2006 = 44,0 mit 16 CPU-Kernen bei 4,14 GHz,  
CFP2006 = 71,5 mit 16 CPU-Kernen bei 3,86 GHz,  
CINT2006rate = 6 52 mit 16 CPU-Kernen bei 4,14 GHz,  
CINT2006rate = 1462 mit 32 CPU-Kernen bei 4,14 GHz,  
CINT2006rate = 2615 mit 64 CPU-Kernen bei 3,86 GHz,  
CFP2006rate = 586 mit 16 CPU-Kernen bei 3,86 GHz,  
CFP2006rate = 1314 mit 32 CPU-Kernen bei 4,14 GHz,  
CFP2006rate = 2296 mit 64 CPU-Kernen bei 3,86 GHz.

Diese Werte sind beeindruckend, insbesondere auch im Vergleich zu dem in einem der folgenden Abschnitte diskutierten Itanium Prozessor.

**Die MIPS-R4000-Architektur:** Der RISC-Prozessor R4000 von MIPS war der erste Prozessor mit einer 64-Bit-Architektur, baut auf dem Stanford RISC-Prototyp auf, sieht ein festes Befehlsformat, 64 verschiedene Befehle, 32 Register und eine integrierte Gleitpunkteinheit vor. Die Verarbeitung erfolgt in einer siebenstufigen, superskalaren Mehrfach-Pipeline, die bis zu 2 Befehle pro Takt gleichzeitig fertig stellen kann. Nachfolgeserien mit jeweils leicht verbesselter Architektur waren R5000 und R10000. Weitere Nachfolger sind wohl nicht über das Planungsstadium hinausgekommen.

**Die DEC- $\alpha$ -Architektur:** 1992 hat DEC ebenfalls einen 64-Bit-RISC-Mikroprozessor vorgestellt, dessen Architektur (zumindest behauptete dies die Herstellerfirma) die neueste und modernste der bekannten Mikroprozessoren war. Durch hohe Taktraten schaffte es DEC auch in den Folgejahren immer wieder, sich mit dem  $\alpha$ -Prozessor an die Spitze von Hitlisten mit Leistungsvergleichen zu setzen. Leider waren die Aussichten dieses Prozessortyps alles andere als günstig, nachdem DEC von Compaq übernommen wurde; die Prozessorsparte wurde sogar an den Konkurrenten Intel weiterverkauft, der die Produktion des  $\alpha$ -Prozessors hat auslaufen lassen.

**Die IA64-Architektur von Intel und HP:** Intel hat zusammen mit HP frühzeitig ein völlig neues Design für zukünftige 64-Bit Prozessoren entwickelt. Die Architektur baut auf einem völlig anderen Konzept auf als RISC. Viele Instruktionen werden zu Einheiten zusammengefasst, die parallel abgearbeitet werden. Dieses Konzept ist auch unter dem Namen VLIW (Very Large Instruction Word) bekannt geworden. Intel und HP haben allerdings einen anderen Begriff für die von ihnen entwickelte Rechnerarchitektur geprägt: EPIC (Explicitly Parallel Instruction Computing). Diese Architektur baut auf einer sehr großen Zahl von Registern auf – jeweils 128 Integer und Gleitpunktregister. Instruktionen werden zu Gruppen und Bündeln zusammengefasst, die weitgehend parallel abgearbeitet werden können. Die Gruppierung soll von Compilern betrieben werden – daher die Bezeichnung explizite Parallelität. Die Ausführung von Befehlen kann auch auf Verdacht (spekulativ) oder in Abhängigkeit von Prädikaten durchgeführt werden. Eine erste Implementierung erfolgte mit dem Prozessor Ita-

nium, es folgte der Itanium 2. Im Februar 2010 wurde die neueste Modellreihe Itanium 9300 vorgestellt. Das Spitzmodell 9350 ist ein Quad-Core Prozessor mit 2,046 Milliarden Transistoren auf 699 mm<sup>2</sup> gefertigt in einer 65nm Technologie. Jeder der 4 CPU Kerne wird normalerweise mit 1,73 GHz getaktet. Mit Hilfe der Turbo Boost Technologie können einzelne Kerne zu Lasten der anderen auch höher getaktet werden und zwar bis zu 1,86 GHz. Jeder CPU-Kern verfügt über einen eigenen Cache mit maximal 6MB aufgeteilt in L1, L2 und L3-Cache. Die maximale Leistungsaufnahme liegt bei 185 W. Der Prozessor kommt auf wenig beeindruckende Werte bei den CPU2006 Werten. Werte für CINT2006 und CFP2006 wurden bisher noch nicht veröffentlicht. Eine Konfiguration mit 2 Prozessor-Chips (also mit 8 CPU-Kernen, erreicht folgende Werte CINT2006rate = 134 und CFP2006rate = 136. Eine andere Konfiguration mit 8 Prozessor-Chips (also mit 32 CPU-Kernen, erreicht folgende Werte CINT2006rate = 531 und CFP2006rate = 520. Offenbar hatte Intel Schwierigkeiten bei der Entwicklung dieses Itanium Prozessors, es wird vermutet das er mit erheblicher Verspätung auf den Markt kam und daher mit anderen Hochleistungsprozessoren nicht mithalten kann- ausser in der Zahl der verbauten Transistoren.

Bisher wird der Itanium nur für den Servermarkt angeboten. Es gibt derzeit (Herbst 2010) noch keine Aussagen von Intel, ob es in absehbarer Zeit Desktop-Rechner auf Itanium Basis geben wird. Im Servermarkt fährt Intel offensichtlich zweigleisig. Neben den Itanium Prozessoren werden hier weiterhin Xeon Prozessoren angeboten, die der konventionellen Prozessorserie x86 entstammen. Dies wundert auch nicht, da die neueren Core i Prozessoren die oben genannten Leistungsdaten übertreffen.

## 5.9 Architektur der Intel-PC-Mikroprozessorfamilie

Die Architektur der Prozessorserie x86 von Intel geht auf den 8080 Chip zurück, der 1974 auf den Markt kam und als der erste kommerziell angebotene Mikroprozessor gilt. Während der 8080 noch ein 8-Bit-Mikroprozessor war, bot Intel 1978 erstmalig mit dem 8086 einen 16-Bit-Mikroprozessor an, der aber zum 8080 weitgehend kompatibel war. Mit dem 80286 erweiterte Intel die Adressbreite des 8086 von 20 auf 24 Bit und erweiterte den Befehlssatz. Mit dem 80386 führte Intel eine 32-Bit-Architektur für die x86-Chipserie ein. Allerdings ist der Prozessor umschaltbar, im 8086-/80286-Mode als 16-Bit-Prozessor und im neuen 80386-Mode als 32-Bit-Prozessor nutzbar. Unter den Betriebssystemen MS-DOS und Windows 3.0 bzw. 3.1 werden die 80386-Prozessoren bzw. ihre Nachfolger, lediglich im 16-Bit-Modus betrieben. UNIX, OS2, Windows 95, Windows 98, Windows ME bzw. Windows NT und Windows XP betreiben diese Prozessoren jedoch im 32-Bit-Modus.

Die folgende Tabelle vergleicht verschiedene ältere Prozessoren. Die Leistung wird in der (fragwürdigen) Einheit MIPS (Millionen Befehle pro Sekunde im Durchschnitt) angegeben. Neuere Leistungsdaten (SPEC95 oder SPEC2000) sind für diese Modelle meist nicht verfügbar.

Chip	Markteinführung	anfänglicher Preis	späterer Preis	MIPS zu Anfang	MIPS später	Transistoranzahl
8086	1978	360\$	/	0.33	0.75	29.000
80286	1982	360\$	8\$	1.2	2.66	134.000
80386	1985	299\$	91\$	5	11.4	275.000
80486	1989	950\$	317\$	20	54	1.200.000
Pentium 66	1993	900\$	300\$	112	112	3.100.000
Pentium 100	1994	700\$	100\$	166	166	3.300.000

Abb. 5.110: Leistung und Preise älterer Intel-Prozessoren. Quelle: Byte, Mai 1993 und spätere Ausgaben

Der Name *Pentium* wurde statt 80586 gewählt, um ein Copyright für den Namen erwerben zu können; für Namen, die nur aus Ziffern bestehen, ist das in den USA nicht möglich. Die Prozessoren 80486 und Pentium erweitern die 80386-Architektur jeweils nur geringfügig und unterscheiden sich hauptsächlich in der Implementierung.

### 5.9.1 Datenstrukturen und Befehle des Pentium

Der Pentium Prozessor hat Befehle zur Bearbeitung folgender Datentypen:

- Bit, Bit-Feld (4 Byte), Bit-Kette (maximal 4 Gigabit lang).
- Zahlen ohne Vorzeichen in den Längen 8, 16 oder 32 Bit.
- Zahlen mit Vorzeichen in Zweier-Komplement-Darstellung (8, 16, 32 oder 64 Bit).
- Byte-Ketten als ASCII-Zeichenketten, BCD-Zahlen oder gepackte BCD-Zahlen.
- 32-Bit und 64-Bit Pointer (*lineare Adressierung*).
- 48-Bit Logische Adressen (*segmentierte Adressierung*).
- Kurze Gleitpunktzahlen mit 8-Bit-Exponenten und 23-Bit-Mantisse.
- Lange Gleitpunktzahlen mit 11-Bit-Exponenten und 52-Bit-Mantisse.
- Temporäre Gleitpunktzahlen mit 15-Bit-Exponenten und 63-Bit-Mantisse.
- Ganze Zahlen mit oder ohne Vorzeichen bestehend aus 64 Bit.
- In 10 Bytes gepackte BCD-Zahlen mit 18 Ziffern und einem Vorzeichenbyte.

Es gibt zehn Kategorien von Maschinenbefehlen:

- Befehle zum Laden, Speichern und Bewegen von Daten,
- Arithmetische Befehle,
- Schiebebefehle und logische Befehle,
- Befehle zur Bearbeitung von Byteketten,
- Befehle zur Bearbeitung von Bitketten,
- Bedingte und unbedingte Sprünge,
- Unterprogrammaufrufe und Unterbrechungen,
- Befehle zur Unterstützung höherer Programmiersprachen,
- Befehle zur Kontrolle des Protected Mode,
- Befehle zur Kontrolle des Prozessors.

### 5.9.2 MMX- und SSE-Befehle

Speziell für die Bearbeitung von Grafik, Audio- und Videodaten hat Intel dem Pentium neue Register und neue Befehle spendiert. Diese Erweiterung wird als *MMX (MultiMedia-eXtension)* bezeichnet (siehe auch S. 528). Bei den genannten Anwendungen hat man oft viele kleine gleichartige Datenpakete, die mit einem Befehl gleichzeitig bearbeitet werden können. So kann man z.B. 8 Pixel zu je 8 Bit oder 4 Audio-Samples zu 16 Bit in ein 64 Bit großes Register packen und dieses mit einem Befehl manipulieren. Typisch sind Befehle wie PADDB (parallel add Byte) oder PADDW (parallel add word) etc. Die Idee ist nicht neu und als Konzept der Parallelverarbeitung unter dem Namen *SIMD (single instruction multiple data)* bekannt.

Beim Pentium III wurde der Befehlssatz nochmals um *Vektorbefehle* erweitert. Diese von Intel entwickelte Befehlssatzerweiterung erhielt den Namen *ISSE* (Internet Streaming SIMD Extensions). In vielen Dingen flexibler als die MMX-Befehlssatzerweiterung, dient sie gleichfalls dazu, Programme durch höhere Parallelisierung zu beschleunigen. Obwohl im Namen ausdrücklich erwähnt, hat diese Technologie nichts direkt mit dem Internet zu tun. Der Verweis war lediglich ein Marketingargument. Nach kurzer Zeit wurde das „I“ weggelassen, so dass man heutzutage nur noch von SSE spricht. SSE2, SSE3 und SSE4 sind jüngere Erweiterungen von SSE.

Pentium Prozessoren und ihre Nachfolgemodelle verfügen bereits seit einiger Zeit über eine Hierarchie von Cache-Speichern, die auf dem Prozessorchip untergebracht sind. Direkt mit dem Prozessor verbunden ist der so genannte L1-Cache. Dieser besteht aus zwei getrennten Teilen für Daten und Befehle. Für Daten und Befehle stehen bei den neueren Modellen je 32 kB zur Verfügung. Mit insgesamt 32 kB ist der L1-Cache im Vergleich zu den neuesten AMD-Prozessoren eher unterdimensioniert. Diese verwenden 64 kB L1-Cache. Der L1-Cache bezieht seine Daten aus dem L2-Cache. Dieser ist bei heutigen Modellen mit 256 kB bei Intel bzw. 512 kB bei AMD pro CPU-Kern dimensioniert. Hinzu kommt ein L3-Cache der für alle CPU-Kerne zuständig ist mit 12 MB bei Intel- bzw. 6 MB bei AMD-Spitzenmodellen.

Trotz aller Risc-Prinzipien ist und bleibt die x86-Architektur eine CISC-Architektur. Das macht sich vor allem bemerkbar durch:

- relativ wenige Register,
- viele verschiedene Befehlsformate mit einer komplexen Befehlscodierung,
- Operationen mit Speicheroperanden,
- viele komplexe Befehle, die Mikroprogramme erfordern.

Alle *einfachen* Befehle werden vom Pentium ohne Mikroprogramm in einem Takt erledigt. Durch die Mehrfach-Pipelines können in einem Takt sogar mehrere Befehle ausgeführt werden. Die komplexe Befehlscodierung kostet viel Zeit und die geringe Registerzahl führt zu häufigeren Speicherzugriffen. Beides benachteiligt den Pentium gegenüber vergleichbaren RISC-Prozessoren.

### 5.9.3 Adressierung

In einem Maschinenbefehl werden Datenadressen folgenden Typs verwendet:

Basis + (Index x Skalierungsfaktor) + Distanz.

Basis und Index werden einem der Mehrzweckregister entnommen, Skalierungsfaktor und Distanz sind absolute Zahlen, die dem jeweiligen Befehl entnommen werden. Von den drei Komponenten einer oben definierten effektiven Datenadresse (EA) können ein oder zwei entfallen (siehe dazu auch die vorangegangen Beispiele). Wenn Daten adressiert werden sollen, wird die oben beschriebene effektive DatenAdresse gebildet und der Segmentierungseinheit als 32-Bit- bzw. 64-Bit-Adresse übergeben. Wenn Befehle adressiert werden sollen, wird die Befehlsadresse aus dem IP-Register (*Instruction Pointer*) bzw. RIP-Register entnommen bzw. von der Prefetch-Einheit vorausberechnet und dann der Segmentierungseinheit übergeben.

### 5.9.4 Die Segmentierungseinheit

Ein auf einem Pentium-Rechner ablaufendes Programm kann Daten und Befehle benutzen, die in Segmenten organisiert sind. Diese Segmente werden mithilfe einer Betriebssystem-Tabelle verwaltet. Diese enthält jeweils Informationen über ein in Ausführung befindliches Programm, das wir *Prozess* nennen wollen. Diese Tabelle kann Einträge für maximal 16 000 Segmente enthalten. Es wäre nicht effizient, wenn ständig auf diese im Speicher befindliche Tabelle zugegriffen werden müsste. Daher sind in der Segmentierungseinheit Register enthalten, die ständig die nötigen Informationen für 6 Segmente enthalten:

CS	Code Segment
SS	Stack Segment
DS,ES, FS,GS	Daten Segmente

Jedes Segment wird durch eine Datenstruktur, den sogenannten Segmentdeskriptor, festgelegt. Der Pentium beherrscht aus Kompatibilitätsgründen drei Arten der Auswertung von Segmentadressen – von denen zwei in den folgenden Abschnitten beschrieben sind.

Zur Bearbeitung von 8086-Programmen muss der Rechner mit 20-Bit-Adressen arbeiten können. In diesem Fall ist die Eingangsadresse der Segmentierungseinheit 16 Bit breit, einem Segmentregister wird der 16 Bit breite Segmentselektor entnommen und, um vier Bits nach links verschoben (d.h. mit  $16=2^4$  multipliziert), zu der Eingangsadresse addiert:

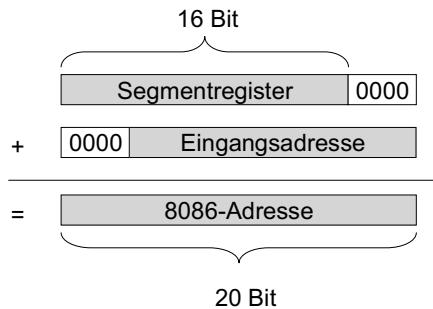


Abb. 5.111: Bearbeitung von 8086-Adressen

Zur Bearbeitung neuer Programme im so genannten *protected mode* wird der Segmentselektor lediglich als Index in die Segmenttabelle des Prozesses interpretiert. Die Eingangsadresse wird zu der im Deskriptor enthaltenen Anfangsadresse des Segmentes addiert. Nunmehr wird geprüft, ob das Ergebnis unterhalb der ebenfalls im Deskriptor enthaltenen Grenzadresse liegt. Die daraus resultierende 32-Bit- bzw. 64-Bit-Adresse wird im korrekten Fall an die Adressübersetzungseinheit weitergereicht. Welches der Segmentregister verwendet wird, hängt von der Adressierung ab.

Befehlsadressen beziehen sich auf das Code-Segmentregister, Datenadressen, die für Stack-Befehle umgewandelt werden, auf das Stack-Segmentregister und alle anderen Adressen auf das DS-Daten-Segmentregister. Die von der CPU implizit vorgenommene Auswahl eines Segmentregisters kann explizit durch einen Prefixbefehl für den nachfolgenden Befehl verändert werden – so können z.B. auch die Segmentregister ES, FS und GS angesprochen werden.

Eine logische Adresse des 80386 ist also 48 Bit lang: die ersten 16 Bit definieren einen Segmentselektor, die restlichen 32 Bit definieren die Relativadresse im Segment. Jedes Segment kann somit bis zu  $2^{32}$  Byte groß sein – also 4 Gigabyte. Eine Segmenttabelle kann bis zu  $2^{14}$  Segmenteinträge haben – damit ergibt sich ein logischer Adressraum von 64 Terabyte. Allerdings ist das eine eher theoretische Überlegung, da nur die Segmente effizient adressierbar sind, die in einem 4 GB-Adressraum untergebracht sind.

Die Segmentierung des Speichers ist wohl von Intel weniger als Ausweitung des Adressraumes gedacht denn als Möglichkeit, einen 4 GB großen Adressraum in geschützte Segmente zu unterteilen: die Segmentierung verhindert die Bildung von Adressen, die außerhalb definierter Segmentgrenzen liegen. Diese Segmentgrenzen sind für ein Anwenderprogramm nicht zugänglich und können nur vom Betriebssystem vergeben werden – daher wird der Betrieb eines Programms mit dieser Art von Segmentadressen auch als *protected mode* bezeichnet.

Im neueren 64-Bit Mode wird die Segmentadressierung nur noch in Ausnahmefällen verwendet. Stattdessen wird ein flaches 64-Bit Adressierungsmodell ohne Segmentierung empfohlen. Allerdings kann auch im 64-Bit Modus ein segmentiertes Speichermodell verwendet werden. Die Segmente werden dann nur noch verwendet um Code-, Daten- und Stackbereiche in dem von einem Programm verwendeten Speicherbereich zu unterscheiden und zu schützen.

### 5.9.5 Adressübersetzung

Das Ergebnis der Segmentierungseinheit ist bei den klassischen x86-Prozessoren eine virtuelle 32-Bit- oder 64-Bit-Adresse. Diese wird in eine reale 32-Bit- oder 64-Bit-Hauptspeicheradresse umgesetzt. Die Umsetzung erfolgt in diesem Fall mithilfe einer Adressumsetzungstabelle, die 32 Einträge (bzw. mehr in neueren Rechnern) hat und *TLB* genannt wird (siehe auch Abschnitt 6.3.11):

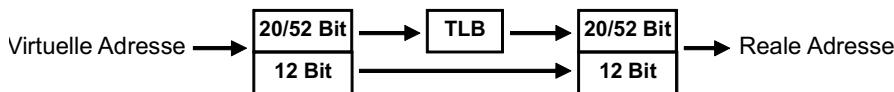


Abb. 5.112: TLB

Falls in der TLB kein passender Eintrag vorhanden ist, wird einem Kontrollregister eine Adresse entnommen, über die in einem zweistufigen Verfahren eine Seitentabelle erreicht wird, der der gesuchte Eintrag entnommen wird, wenn die fragliche Seite im Hauptspeicher resident ist. Andernfalls muss das Betriebssystem über einen Seitenwechselfehler informiert werden und die fehlende Seite beschaffen. Die Adressübersetzung erfolgt parallel zu den anderen Aktivitäten der CPU – so wird erreicht, dass fast keine Zeit zusätzlich benötigt wird.

### 5.9.6 Betriebsarten des Pentium

Der Pentium kann im so genannten Real-Mode betrieben werden – er verhält sich dann wie ein 8086. In dieser Betriebsart können 8086-Programme ohne jede Änderung bearbeitet werden – allerdings wesentlich schneller als mit dem Vorgängermodell.

Im Protected Mode können alte 80286-Programme und neuere Programme ablaufen, die die Möglichkeiten dieses Prozessors überhaupt erst richtig nutzen. Daneben können 8086-Programme im so genannten Virtual-8086-Mode ausgeführt werden. Alle diese Programme können als Prozesse konkurrierend betrieben werden. Der Prozessor unterstützt dies durch vordefinierte Datenstrukturen zur Verwaltung von Prozessen, durch Maschinenbefehle, die den Prozesswechsel unterstützen, und durch verschiedene Schnittstellen, die einen Prozess in die Lage versetzen, Anforderungen anderer Prozesse zu erfüllen. Dabei sind die von den einzelnen Prozessen verwendeten Speicherbereiche gegen unberechtigte Zugriffe anderer Prozesse geschützt. Die Prozesse haben definierte Rechte und Privilegien. Typisch für eine solche Umgebung ist es, dass nur die Betriebssystem-Prozesse das Recht haben, bestimmte privilegierte Befehle auszuführen, wie z. B. Ein- und Ausgabebefehle. Analog können derzeit Programme im 64-Bit Modus betrieben werden. In einem Kompatibilitätsmodus können ältere Programme ablaufen.

Intel hat den Pentium mit der neuesten, sonst nur bei RISC-Prozessoren vorhandenen Technologie ausgestattet. Diese Technik konnte bei der Entwicklung des PentiumPro, des Pentium II, des Pentium III, des Pentium-4 und des Pentium-4E noch weiter verbessert werden. Allerdings führte die Entwicklung der Pentium-4 Prozessoren wegen zu großer Hitzeentwicklung

in eine Sackgasse. Die neueren Core 2 Duo bzw. i3, i5 und i7 Prozessoren bauen auf der neueren Core/Nehalem Mikroarchitektur auf und erzielen eine höhere Leistung bei niedrigerer Wärmeentwicklung.

Die ursprüngliche Leistung des Pentium 66 lag bei 112 MIPS im Jahre 1993, steigerte sich 1994 beim Pentium 100 auf 168 MIPS. Seit dieser Zeit sind die SPECint95- und SPECfp95-Werte für diesen Prozessor erhältlich. Sie liegen bei 3,3 und 2,59. Diese Werte konnten mit den neueren Pentium-4 Modellen um den Faktor 15 verbessert werden und bei den neuesten Core i7 Modellen weiterhin um einen Faktor von mindestens 10.

Die folgende Tabelle vergleicht Intel-Prozessoren aus „mittleren Jahrgängen“. Die Leistungsdaten liegen als SPECint95 bzw. SPECfp95 vor:

Chip	Markt-einführung	SPECint95	SPECfp95	Zahl der Transistoren
Pentium 100	1994	3,3	2,59	3,3 Millionen
Pentium 200 MMX	1997	6,41	4,66	4,5 Millionen
PentiumPro 200	1995	8,09	6,75	5,5 Millionen
Pentium-II 266	1997	10,8	6,89	7,5 Millionen
Pentium-II 450	1998	18,5	13,3	7,5 Millionen
Pentium-III 550	1999	22,3	15,1	9,5 Millionen

Abb. 5.II3: Leistungsdaten von Intel-Prozessoren der 90er Jahre. Quelle: Intel

Die neuere Entwicklung der Nachfolger des 8086-Prozessors wird durch einen intensiven Konkurrenzkampf der Firmen AMD und Intel geprägt. Bis 1998 konnte Intel sich stets rühmen, die leistungsfähigsten Prozessoren herzustellen. Seither aber bietet AMD Prozessoren unter dem Namen *Athlon* und *Opteron* an, die den Intel-Prozessoren ebenbürtig oder sogar überlegen sind. Neuere Marktnamen der AMD Prozessoren sind *Athlon 64*, *Athlon 64 FX* und *Athlon 64 X2*. Der Name *Opteron* wird weiterhin für Prozessoren verwendet, die im Servermarkt angeboten werden.

Intel hatte Anfang 2001 den Pentium-4 Prozessor eingeführt und diese Prozessorfamilie dann in mehreren Schritten verbessert. Dabei setzte Intel auf die mit dem Pentium-4 eingeführte *Netburst*-Architektur. Diese setzt auf eine extrem lange Pipeline, die eine sehr hohe Taktrate ermöglichen soll. Ursprüngliche Pläne sahen vor, mit dieser Architektur frühzeitig 4 GHz und später bis zu 10 GHz zu erreichen. Tatsächlich erreichten die letzten Pentium-4 Modelle nur 3,8 GHz. Intel musste einsehen, dass der Energieverbrauch und die damit verbundene Wärmeentwicklung der Prozessoren bei hohen Taktraten nicht in den Griff zu kriegen war.

Im Gegenzug hatte sich AMD frühzeitig entschlossen, die 32-Bit-Architektur zu einer abwärtskompatiblen 64-Bit-Architektur weiterzuentwickeln und gleichzeitig eine Prozessorarchitektur benutzt, die auf höhere Durchsatzleistung bei niedrigeren Taktraten baut. Mit diesem Konzept hatte AMD jahrelang bei dem Wettrennen um die leistungsfähigsten Prozessoren die Nase vorn.

Die folgende Tabelle vergleicht Intel und AMD Prozessoren aus „neueren Jahrgängen“. Die Leistungsdaten liegen als SPEC-CINT2000 bzw. SPEC-CFP2000 vor:

Prozessor	Jahr	Takt GHz	CINT2000	CFP2000	Millionen Transistoren	Chipfläche	max. Watt
Pentium-4	2001	1,5	502	524	42	217 mm <sup>2</sup>	54,7
Athlon-C	2001	1,4	554	458	37	117 mm <sup>2</sup>	80
Athlon-XP	2001	1,533	597	504	37,5	129 mm <sup>2</sup>	66
Athlon 64	2003	2,0	1266	1355	105,9	193 mm <sup>2</sup>	89
Pentium-4C	2003	3,2	1205	1267	55	131 mm <sup>2</sup>	82
Athlon 64 FX55	2004	2,6	1750	1854	105,9	193 mm <sup>2</sup>	104
Pentium-4E	2004	3,4	1400	1397	125	112 mm <sup>2</sup>	103

Abb. 5.114: Leistungsdaten von Intel- und AMD-Prozessoren der Jahre 2001 bis 2004

Intel hatte jahrelang der besseren Prozessorarchitektur von AMD nichts entgegenzusetzen, musste schließlich sogar die 64-Bit-Architektur von AMD übernehmen und die Weiterentwicklung der *Netburst-Architektur* aufgeben. Statt dessen setzt Intel nunmehr auf die *Intel Core Mikroarchitektur*. Diese wurde am 7. März 2006 offiziell angekündigt und sollte noch im Jahr 2006 die NetBurst-Architektur komplett ersetzen. Die ersten Modelle mit der neuen Architektur wurden Ende Juli 2006 unter der Bezeichnung Core 2 Duo mit den Modellvarianten E6300, E6400, E6600, E6700 und X6800 eingeführt. Ab 2008 wurde die neuere *Intel Nehalem Mikroarchitektur* angekündigt. In den Folgejahren wurden auf dieser Architektur aufbauend die Prozessorfamilien i3, i5 und i7 eingeführt, mit zahlreichen Prozessoren sowohl für Desktop- als auch für mobile Computer.

Die neu entwickelten Mikroarchitektur besitzt Ähnlichkeit mit der alten *P6-Architektur*, die erstmalig für den Pentium Pro entwickelt wurde und später auch für den Pentium III eingesetzt wurde. Diese Architektur wurde parallel zu der Netburst-Architektur für die mobilen Prozessoren von Intel weiterentwickelt. Die neueren Versionen dieser Architektur, die z.B. von dem Pentium-M benutzt werden, waren bereits auf hohe Leistung bei geringem Energieverbrauch ausgelegt. Niedrigere Verlustleistung und mehrere CPU-Kerne gehören zu den besonderen Merkmalen der Core/Nehalem Mikroarchitektur. Im Gegensatz zur Netburst-Architektur, die eine mehr als 30-stufige Pipeline aufweist, ist die neuere Mikroarchitektur mit einer relativ kurzen, 14-stufigen Pipeline auf niedrigere Taktraten ausgelegt und erreicht ihre Leistung vor allem aufgrund einer hohen Anzahl von Befehlen per Taktzyklus. Die Intel Core/Nehalem Mikroarchitektur ist ein vierfach superskalares Design im Gegensatz zum dreifach superskalaren Design des Pentium M und Pentium-4. Bereits beim Pentium-4 wurde der Befehlssatz nochmals um weitere Multimediatebefehle SSE (Streaming Extensions) erweitert. Die SSE-Einheiten der Core Architektur besitzen intern eine auf 128 Bit verdoppelte Busbreite und können daher SSE-Befehle in nur einem Taktzyklus verarbeiten. Ebenfalls verbes-

sert wurden in der neueren Mikroarchitektur das Stromsparkkonzept, das nunmehr eine feinere Abstufung besitzt und deswegen effizienter arbeitet.

Die Core/Nehalem Mikroarchitektur wurde konsequent für mehrere CPU-Kerne entwickelt. Unter anderem sieht das Konzept vor, den L2-Cache dynamisch den verschiedenen CPU-Kernen zuzuweisen. Falls ein CPU-Kern inaktiv sein sollte, bekommt ein anderer CPU-Kern den gesamten L2-Cache zugewiesen. Ebenso kann ein CPU-Kern mit einer höheren Taktfrequenz betrieben werden, wenn die anderen dafür mit niedrigerem Takt arbeiten.

Mit der im Juli 2006 eingeführten Prozessorfamilie Core 2 Duo, die die Core Mikroarchitektur erstmalig vollständig implementiert, ist es Intel gelungen, die zu diesem Zeitpunkt leistungsfähigsten Prozessoren anzubieten und den jahrelangen Rückstand gegenüber AMD wieder einzuholen. Bereits nach kurzer Zeit wurden die weiter verbesserten Prozessorfamilien i3, i5 und i7 auf den Markt gebracht. In der folgenden Tabelle werden zwei aktuelle Prozessoren der Herstellerfirmen AMD und Intel verglichen.

	Phenom II X6 1090T	Intel Core i7 980X
Taktfrequenz (Mitte 2010)	3,2 GHz bzw 3,6 GHz	3,33 GHz bzw 3,6 GHz
Anzahl CPU-Kerne	6	6
Größe des L1-Cache	64 kB Befehle 64 kB Daten	32 kB Befehle 32 kB Daten
Größe des L2-Cache (auf Chip)	512 kB je CPU Kern	256 kB je CPU Kern
Größe des L3-Cache (auf Chip)	6 MB	12 MB
Prozess	45 nm	32 nm
Chipfläche	346 mm <sup>2</sup>	248 mm <sup>2</sup>
Transistoren	904 Millionen	1,17 Milliarden
Speicherbus	Hyper Transport 3.0	QuickPath Interconnect
Speicherart	bis zu DDR3-1333	bis zu DDR3-1066
CINT2006rate/CFP2006rate	?	173/115
Thermal Design Power (Watt)	125 W	130 W

Abb. 5.115: Vergleich von AMD und Intel Prozessoren aus dem Jahr 2010.

Bei der Implementierung der Pentium bzw. Core Duo Prozessoren hat Intel weitgehend die heute bei RISC-Prozessoren üblichen Prinzipien verwirklicht. Der Pentium verfügt über fünf mehrstufige Pipelines für Integer-Arithmetik und zwei mehrstufige Pipelines für Gleitkomma-Arithmetik. Die Anzahl der Bearbeitungsstufen wurde beim Pentium-4 mit 32 angegeben. Man kann sich allerdings kaum vorstellen wie die Bearbeitung eines Maschinenbefehls in 32 Einzelschritte aufgeteilt werden kann. Vermutlich werden in einer solchen langen Pipeline mehrere Befehle zusammen bearbeitet. Wie bereits erwähnt, hat sich das Konzept einer langen Pipeline

nicht bewährt. Mit der neuen *Core/Nehalem Microarchitecture* schrumpfte die Länge der Pipeline wieder auf 14.

Angespornt durch die Athlon Konkurrenten von AMD, gelingt es Intel immer wieder, die vermeintlichen Leistungsvorteile der RISC-Konkurrenten durch „schnellere“ Chip-Technologie wett zu machen. Trotzdem leidet diese Prozessorfamilie an den Schwächen des in den 70er Jahren definierten Befehlssatzes, an der geringen Registerzahl und an der segmentierten Adressierung. Wünschenswert wäre daher der Übergang zu einer moderneren Prozessorarchitektur. Der Itanium besitzt beispielsweise eine wesentlich moderne Architektur, wurde aber vom Markt nicht mit dem erwarteten Enthusiasmus angenommen und erreicht auch immer noch nicht die Leistungsdaten eines modernen i7 Prozessors.

Andererseits hat sich die Architektur der x86 Familie als de facto-Standard für Personal Computer durchgesetzt. Weltweit werden hunderte Millionen PCs pro Jahr verkauft, die zu diesem Standard kompatibel sind. Viele Milliarden Standard-PCs sind im Einsatz – mit einer ungeheuren Menge von installierter Software, die nur auf diesen Prozessoren lauffähig ist. Daher wird der Bedarf an immer schnelleren Prozessoren, die kompatibel zu den Pentium-Prozessoren sind, auf absehbare Zeit eher zunehmen – auch wenn man heute Prozessoren kaufen könnte, die moderner und schneller sind.