Winfried Gehrke
Marco Winzker
Klaus Urbanski
Roland Woitowitz

LEHRBUCH

Digitaltechnik

Grundlagen, VHDL, FPGAs, Mikrocontroller

7. Auflage

Springer Vieweg

Springer-Lehrbuch

Winfried Gehrke · Marco Winzker
Klaus Urbanski · Roland Woitowitz

Digitaltechnik

Grundlagen, VHDL, FPGAs, Mikrocontroller

7., überarbeitete und aktualisierte Auflage

Springer Vieweg

Winfried Gehrke
Osnabrück, Deutschland

Klaus Urbanski
Osnabrück, Deutschland

Marco Winzker
St. Augustin, Deutschland

Roland Woitowitz
Osnabrück, Deutschland

OnlinePlus Material zu diesem Buch finden Sie auf
http://www.springer.com/978-3-662-49731-9

ISSN 0937-7433
Springer-Lehrbuch
ISBN 978-3-662-49730-2 ISBN 978-3-662-49731-9 (eBook)
DOI 10.1007/978-3-662-49731-9

Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.d-nb.de abrufbar.

Springer Vieweg

Gedruckt auf säurefreiem und chlorfrei gebleichtem Papier

Springer Vieweg ist Teil von Springer Nature
Die eingetragene Gesellschaft ist Springer-Verlag GmbH Germany
Die Anschrift der Gesellschaft ist: Heidelberger Platz 3, 14197 Berlin, Germany

Vorwort

Die Digitaltechnik ist ein integraler Bestandteil unseres täglichen Lebens geworden. Vielfach begegnet sie uns in Form von Desktop-PCs, Laptops, Tablets, Fernsehgeräten oder Smartphones. Wenn wir ein solches Gerät nutzen, ist klar: Wir verwenden ein digitales System. Darüber hinaus ist die Digitaltechnik aber auch in Bereiche eingezogen, bei der sie nicht sofort offensichtlich ist. In einem modernen Auto arbeiten beispielsweise zahlreiche digitale Komponenten. Sie steuern den Motor, helfen beim Einparken und unterstützen beim Fahren durch Fahrspurassistenten, ABS und ESP. Diese Form der digitalen Systeme werden häufig, weil sie in einem größeren System integriert sind, als „eingebettete Systeme“ bezeichnet. Man findet sie in vielen Bereichen des Alltags, wie zum Beispiel in Hausgeräten, Uhren, Heizungssteuerungen oder in Fotoapparaten. Auch in industriellen Anwendungen geht nichts ohne die Digitaltechnik. So wäre beispielsweise die Vernetzung industriell genutzter Maschinen, die vierte industrielle Revolution, ohne entsprechende digitale Komponenten undenkbar.

Was ist Digitaltechnik? Welche Prinzipien liegen ihr zugrunde? Wie werden digitale Systeme realisiert?−Diese und andere Fragen werden in diesem Lehrbuch beantwortet.

Das Buch beschreibt die wichtigen Themenfelder der Digitaltechnik und wendet sich vorrangig an Studierende der Studiengänge Elektrotechnik, Informatik, Mechatronik sowie verwandter Studiengänge. Es wird der Bogen von den Grundlagen der Digitaltechnik über Schaltungsstrukturen und Schaltungstechnik bis hin zu den Komponenten digitaler Systeme, wie programmierbare Logikbausteine, Speicher, AD/DA-Umsetzer und Mikrocontroller gespannt. Zahlreiche Beispiele erleichtern das Verständnis für den Aufbau und die Funktion moderner digitaler Systeme.

Mit dieser 7. Auflage und dem auf vier Autoren gewachsenen Team wurde das Lehrbuch grundlegend überarbeitet und modernisiert. Hierbei waren uns die folgenden Aspekte wichtig:

- Ein besonderes Merkmal dieses Lehrbuches ist die Breite der behandelten Themen von Grundlagen über Komponenten bis zu digitalen Systemen.
- Der Entwurf mit einer Hardwarebeschreibungssprache ist Standard in der Industrie. Auf die verständliche, schrittweise Erläuterung von VHDL wird daher besonderer

Wert gelegt. Nach einer Einführung in VHDL wird bei vielen Themen Bezug auf VHDL-Konstrukte genommen. In einem Vertiefungskapitel werden weiterführende Sprachkonstrukte erläutert.
- Ein neues Einleitungskapitel gibt eine Übersicht über die Digitaltechnik, um die Einordnung von Grundlagenwissen zu ermöglichen.
- Inhalte, die keine Praxisrelevanz mehr haben, wurden weggelassen, zum Beispiel asynchrone Zähler oder obsolete RAM-Bausteine.
- Im Gegenzug werden praxisrelevante Inhalte ausführlicher behandelt, darunter:
  - Zeitverhalten, Pipelining
  - Schaltungssimulation und -verifikation
  - Verlustleistung
  - Moderne Speichertechnologien

Die ersten sechs Kapitel legen die wesentlichen Grundlagen zum Verständnis digitaler Komponenten. Kap. 1 bietet eine Einführung in die Thematik und stellt wichtige Grundprinzipien im Überblick dar.

Kap. 2 widmet sich der digitalen Darstellung von Informationen, wobei der Schwerpunkt auf der Darstellung von Zahlen liegt. Kap. 3 führt in die Hardwarebeschreibungssprache VHDL ein, die weltweit für den Entwurf digitaler Schaltungen verwendet wird. Digitale Systeme lassen sich als Kombination von kombinatorischen und sequenziellen Schaltungen auffassen. Beide Konzepte werden in den Kapiteln 4 und 5 vorgestellt, während sich Kap. 6 den aus diesen Konzepten abgeleiteten Schaltungsstrukturen widmet. In diesen Kapiteln wird kontinuierlich die Implementierung in der Sprache VHDL thematisiert und vertieft.

In den Kapiteln 7 bis 14 werden vertiefende Themen aufgegriffen: Kap. 7 stellt unterschiedliche Konzepte zur Realisierung digitaler Systeme im Überblick vor. In Kap. 8 werden erweiterte Aspekte der Schaltungsbeschreibung in VHDL, wie zum Beispiel Testbenches für die Verifikation aufgegriffen. Die praktische Umsetzung von VHDL-Beschreibungen erfolgt heute häufig mithilfe von programmierbaren Logikbausteinen (FPGAs), welche in Kap. 9 vertieft vorgestellt werden. Das Verständnis der technologischen Grundlagen moderner Digitalschaltungen wird durch eine Einführung in die Halbleitertechnologie in Kap. 10 ermöglicht. Eine zentrale Systemkomponente ist der Speicher. Dieser kann mithilfe unterschiedlicher Technologien realisiert werden, die in Kap. 11 vorgestellt werden. Für Ein-/Ausgabe analoger Größen werden Analog-Digital- und Digital-Analog-Umsetzer benötigt, deren Aufbau und Funktionsweise in Kap. 12 näher erläutert werden. Kap. 13 und 14 widmen sich digitalen Rechnersystemen. In Kap. 13 wird der Aufbau und die Funktionsweise von Rechnern vorgestellt. Kap. 14 greift diese Aspekte auf und vertieft sie anhand eines konkreten Beispiels, einem Mikrocontroller der AVR-Familie. In Kap. 11 bis 14 werden ebenfalls Bussysteme zur Kommunikation innerhalb eines digitalen Systems vorgestellt.

Am Ende aller Kapitel befinden sich Übungsaufgaben, die wichtige Aspekte aufgreifen und zur selbstständigen Lernkontrolle herangezogen werden können. Die Lösungen der Aufgaben sind am Ende des Buches zu finden.

Ergänzendes Material steht im Internet unter www.springer.com/de/book/9783662497302 oder www.hs-osnabrueck.de/buch-digitaltechnik zur Verfügung.

Für die Rückmeldungen zu den Lehrinhalten bedanken wir uns bei den Studierenden der Hochschule Osnabrück und der Hochschule Bonn-Rhein-Sieg. Besonderer Dank geht an alle Kolleginnen und Kollegen, die uns seit der ersten Auflage durch ihre Hilfe und Rückmeldungen begleitet haben. In der aktuellen Auflage war dies insbesondere Dipl.-Ing. Andrea Schwandt. Nicht zuletzt möchten wir uns bei allen an dieser Ausgabe beteiligten Mitarbeiterinnen und Mitarbeitern des Springer-Verlages bedanken. Ohne ihre professionelle Arbeit wäre das vorliegende Buch nicht realisierbar gewesen.

Das Lehrbuch soll natürlich Leserinnen und Leser gleichermaßen ansprechen und wir haben uns bemüht, dass alle Formulierungen auch so verstanden werden.

im Oktober 2016

Winfried Gehrke
Marco Winzker
Klaus Urbanski
Roland Woitowitz

Vorwort zur ersten Auflage

Die Digitaltechnik hat seit der Einführung der ersten digitalen integrierten Halbleiterschaltungen im Jahre 1958 einen vehementen Aufschwung genommen. Maßgeblich daran beteiligt war der technologische Fortschritt in der Mikroelektronik. Mittlerweile lassen sich integrierte Schaltungen mit mehr als 100 Mio. aktiven Elementen realisieren.

Anfänglich konzentrierte sich diese Technik einerseits auf niedrigintegrierte logische Grundschaltungen und andererseits auf hochintegrierte kundenspezifische Schaltungen (Full Custom ICs), aber bereits 1971 kamen die Mikroprozessoren als neuartige programmierbare Universalschaltungen hinzu.

Seit einigen Jahren erweitert sich das Anwendungsspektrum zunehmend in Richtung der sog. Semi Custom ICs. Hierbei handelt es sich um hochintegrierte Standardschaltungen, bei denen wesentliche Designschritte mittels Computerunterstützung vom Anwender selbst übernommen werden.

Das Buch widmet sich all diesen Grundlagen der Digitaltechnik unter besonderer Berücksichtigung der zurzeit gültigen Normen für Schaltsymbole und Formelzeichen.

Der Darstellung grundlegender Logikbausteine, wie NAND, NOR, Flipflops und Zähler sowie programmierbarer Bausteine, wie PAL, PLA, LCA schließt sich eine Einführung in die Mikroprozessor- und Mikrocontroller-Technik an.

Einen besonderen Schwerpunkt bildet der systematische Entwurf von Schaltnetzen und Schaltwerken unter Einsatz programmierbarer Bausteine. Zahlreiche Beispiele hierzu erleichtern das Verständnis für Aufbau und Funktion dieser modernen digitalen Systeme.

Zu allen Kapiteln werden Übungsaufgaben mit ausführlichen Musterlösungen angeboten. Daher eignet sich dieses Buch besonders zum Selbststudium. Es wendet sich damit sowohl an Hochschulstudenten der Elektrotechnik oder Informationstechnik im Hauptstudium, als auch an den in der Berufspraxis stehenden Ingenieur, der seinen Wissensstand auf diesem Gebiet aktualisieren will.

Besonderer Dank gebührt Herrn Dr.-Ing. H. Kopp, der dieses Buch durch wertvolle Anregungen und vielfältige Unterstützung bereichert hat. Auch den Studenten der Fachhochschule Osnabrück gilt unser Dank für ihre Mitarbeit und mannigfache Hilfestellung.

Bedanken möchten wir uns ebenfalls beim Verlag für die gute Zusammenarbeit.

Osnabrück, Deutschland, Dezember 1992

Klaus Urbanski
Roland Woitowitz

Inhaltsverzeichnis

Abkürzungsverzeichnis

ADC	Analog Digital Converter
ADU	Analog-Digital-Umsetzer
AHG	Abtast-Halte-Glied
ALM	Adaptive Logic Module
ALU	Arithmetic Logical Unit
ASCII	American Standard Code for Information Interchange
ASIC	Application Specific Integrated Circuit
ASSP	Application Specific Standard Product
BCD	Binary Coded Decimal
BGA	Ball Grid Array
CISC	Complex Instruction Set Computer
CLB	Complex Logic Block
CMOS	Complementary Metal Oxide Semiconductor
CNT	Carbon Nano Tube
COB	Capacitor over Bitline
CPLD	Complex Programmable Logic Device
CPU	Central Processing Unit
DAC	Digital Analog Converter
DAU	Digital-Analog-Umsetzer
DCE	Data Communication Equipment
DDR	Double Data Rate
DIL	Dual In-Line Package
DNF	Disjunktive Normalform
DRAM	Dynamic Random Access Memory
DSP	Digital Signal Processing, Digital Signal Processor
DTE	Data Terminal Equipment
ECC	Error Correcting Code
EDA	Electronic Design Automation
EEPROM	Electrically Erasable Programmable Read Only Memory
FET	Field Effect Transistor
FFT	Fast Fourier Transform
FIFO	First In First Out

FPGA	Field Programmable Gate Array
FRAM	Ferroelectric Random Access Memory
GPU	Graphics Processing Unit
I2C	Inter-Integrated Circuit
IC	Integrated Circuit
IEC	International Electrotechnical Commission
IEEE	Institute of Electrical and Electronics Engineers
INL	Integrale Nichtlinearität
IOB	Input Output Block
ISP	In System Programming
ISR	Interrupt Service Routine
JTAG	Joint Test Action Group
KNF	Konjunktive Normalform
LE	Logic Element
LSB	Least Significant Bit (Niederwertigstes Bit)
LUT	Look-Up Table
LVDS	Low Voltage Differential Signaling
LVTTL	Low Voltage Transistor-Transistor-Logic
MLC	Multi Level Cell
MOS	Metal Oxide Semiconductor
MRAM	Magnetoresistive Random Access Memory
MSB	Most Significant Bit (Höchstwertigstes Bit)
NVRAM	Non-Volatile Random Access Memory
OSR	Oversampling Ratio
OTP	One Time Programmable
PC	Program Counter, Personal Computer
PCRAM	Phase-Change Random Access Memory
PLA	Programmable Logic Array
PLCC	Plastic Leaded Chip Carrier
PLD	Programmable Logic Device
PLL	Phase Locked Loop
PWM	Pulse Width Modulation, Pulsweitenmodulation
QDR	Quad Data Rate
QFP	Quad Flat Pack
QLC	Quad Level Cell
RAM	Random Access Memory
RISC	Reduced Instruction Set Computer
ROM	Read Only Memory
RRAM	Resistive Random Access Memory
RTL	Register Transfer Level
SAR	Successive Approximation Register
SDRAM	Synchronous Dynamic Random Access Memory
SINAD	Signal-to-Interference Ratio including Noise and Distortion
SNR	Signal-to-Noise Ratio
SPI	Serial Peripheral Interface

SPLD	Simple Programmable Logic Device
SRAM	Static Random Access Memory
THD	Total Harmonic Distortion
THS	Total Hold Slack
TLC	Triple Level Cell
TNS	Total Negative Slack
TTL	Transistor-Transistor-Logik
TWI	Two Wire Interface
UART	Universal Asynchronous Receiver Transmitter
USART	Universal Synchronous Asynchronous Receiver Transmitter
VCO	Voltage Controlled Oscillator
VHDL	Very High Speed Integrated Circuit Hardware Description Language
VLIW	Very Long Instruction Word
WHS	Worst Hold Slack
WNS	Worst Negative Slack

Einführung

1

Digitaltechnik steckt heutzutage in vielen technischen Geräten. Wenn Sie dieses Buch lesen, haben Sie vermutlich den Tag über schon etliche digitale Schaltungen benutzt. Der Rauchmelder im Schlafzimmer, der nachts auf Sie aufpasst, hat einen kleinen digitalen Mikrocontroller, genau wie der Radiowecker, der Sie geweckt hat. Mit dem Smartphone voller Digitaltechnik haben Sie vermutlich Ihre Emails und sozialen Netzwerke nach Neuigkeiten abgefragt. Und egal ob Sie mit dem Auto oder der Straßenbahn in die Hochschule gefahren sind, wieder waren digitale Schaltungen für Sie tätig. Nur falls Sie mit dem Fahrrad unterwegs waren, verlief dieser Teil des Tages ohne Digitaltechnik – es sei denn, Sie haben einen Fahrradtacho.

Digitale Schaltungen übernehmen in vielen technischen Geräten Aufgaben zur Steuerung und Regelung. Das heißt, sie fragen Informationen ab und treffen anhand von Regeln Entscheidungen. Dieses Grundprinzip wird beispielsweise beim Antiblockiersystem (ABS) im Auto deutlich. Die Digitalschaltung bekommt die Informationen, ob die Bremse betätigt ist und die Räder blockieren. Wenn dies der Fall ist, wird die Bremskraft leicht reduziert, damit die Räder wieder Haftung zur Straße bekommen und man bessere Bremswirkung sowie Manövrierbarkeit erhält.

Der besondere Vorteil von digitalen Schaltungen liegt darin, dass Berechnungen und Entscheidungen sowie das Speichern und Übertragen von Informationen sehr einfach möglich sind. Prinzipiell könnte ein Antiblockiersystem auch mit einer Analogschaltung und eventuell sogar mechanisch oder hydraulisch aufgebaut werden. Aber ein digitales System kann die Informationen wesentlich präziser verarbeiten, also beispielsweise die Geschwindigkeit vor dem Bremsen, die Stellung des Lenkrads und die Drehgeschwindigkeit aller Räder auswerten und alle Bremsen individuell ansteuern.

W. Gehrke et al., *Digitaltechnik,* Springer-Lehrbuch,
DOI 10.1007/978-3-662-49731-9_1

1.1 Arbeitsweise digitaler Schaltungen

Ein wichtiges Kennzeichen der Digitaltechnik ist die Darstellung von Informationen mit den Werten 0 und 1. Dieses Prinzip wird als *Zweiwertigkeit* bezeichnet. Daten mit zwei möglichen Werten werden *Binärdaten* genannt. Wenn eine Information mehr als zwei Werte haben kann, wird sie mit mehreren Stellen dargestellt. Am bekanntesten ist sicher das Byte, ein Datenwort mit acht Bit, also acht Stellen mit dem Wert 0 oder 1.

1.1.1 Darstellung von Informationen

Binärdaten werden meistens mit Spannungspegeln dargestellt, beispielsweise die 0 mit 0 V und die 1 mit 3,3 V. Dabei sind auch geringe Abweichungen der Spannung erlaubt, das heißt auch eine Spannung von beispielsweise 0,2 V wird noch als 0 akzeptiert. Dies ist eine wichtige Eigenschaft der Digitaltechnik, denn dadurch ist sie gegenüber kleinen Störungen und Rauschen unempfindlich. Erst bei großen Störungen kann der Wert einer Information nicht mehr korrekt erkannt werden.

Für die Darstellung von Binärdaten mit Spannungspegeln gibt es mehrere Standards. Beispielsweise wird im Standard LVTTL der Spannungsbereich von 0 bis 0,8 V als logische 0 und von 2,0 bis 3,3 V als logische 1 interpretiert. Der Bereich zwischen 0,8 und 2,0 V ist der Übergangsbereich und diese Spannungen dürfen nur kurz beim Wechsel zwischen 0 und 1 auftreten. Die Bezeichnung LVTTL bedeutet übrigens Low-Voltage-Transistor-Transistor-Logik und hat gewissermaßen „historischen“ Ursprung. Sie ist eine spannungsreduzierte Version (Low-Voltage) eines anderen Standards (TTL).

Es gibt, neben LVTTL, eine Vielzahl weiterer Standards für Spannungspegel. Früher wurden oft höhere Spannungen, z. B. 5 V, verwendet, sodass auch höhere Pegel gebräuchlich waren. Innerhalb von integrierten Schaltungen, z. B. der CPU in Ihrem Computer, werden heutzutage geringere Spannungen im Bereich von 1 V benutzt.

Die Werte 0 und 1 können je nach Anwendung auch durch andere physikalische Größen dargestellt werden, beispielsweise Lichtimpulse in einer Glasfaserleitung oder durch elektrische Ladung auf einem Kondensator.

1.1.2 Logik-Pegel und Logik-Zustand

Die Begriffe *Logik-Pegel* und *Logik-Zustand* unterscheiden Spannungswerte und Information einer binären Variablen. Der Logik-Pegel wird durch L (Low) und H (High) und der Logik-Zustand durch die Ziffern 0 und 1 bezeichnet. Für die Beschreibung des physikalischen Verhaltens einer digitalen Schaltung dienen somit die Logik-Pegel, während das logische Verhalten durch Logik-Zustände gekennzeichnet wird.

Die Zuordnung von L und H zu 0 und 1 erfolgt fast immer in *positiver Logik,* das heißt der Pegel L entspricht einer logischen 0 und Pegel H entspricht einer logischen 1.

Prinzipiell ist auch eine umgekehrte Zuordnung möglich, die als *negative Logik* bezeichnet wird. Diese Zuordnung wird in der Praxis jedoch kaum verwendet.

1.1.3 Verarbeitung von Informationen

Digitalschaltung können die logischen Werte 0 und 1 für Berechnungen und Entscheidungen verwenden. Das Ergebnis einer Berechnung ist dabei wieder der Wert 0 oder 1. Die Grundelemente zur Berechnung werden als *Logikgatter* bezeichnet. Die wichtigsten Logikgatter sind:

- **Inverter:** Der Inverter ergibt am Ausgang das „Gegenteil“ des Eingangs. Das heißt eine 0 wird zur 1, eine 1 zur 0.
- **UND-Gatter:** Das UND-Gatter hat zwei oder mehr Eingänge. Es ergibt am Ausgang eine 1, wenn alle Eingänge 1 sind. Mit anderen Worten: Der eine **und** der andere Eingang müssen 1 sein.
- **ODER-Gatter:** Das ODER-Gatter hat ebenfalls zwei oder mehr Eingänge. Es ergibt 1, wenn mindestens ein Eingang 1 ist. Auch der Fall, dass mehrere Eingänge 1 sind ist erlaubt. Mit anderen Worten: Der eine **oder** der andere **oder** beide Eingänge müssen 1 sein.
- **XOR-Gatter:** Das XOR-Gatter ist in der Grundform für zwei Eingänge definiert. Die Bezeichnung bedeutet ausschließendes Oder (engl. *exclusiv-or*). Es ist eine Abwandlung des ODER-Gatters, die jedoch keine 1 ausgibt, wenn beide Eingänge 1 sind. Mit anderen Worten: Für eine 1 am Ausgang müssen der eine **oder** der andere Eingang aber **nicht beide** Eingänge 1 sein.

Für die Logikgatter gibt es *Schaltsymbole,* die in Abb. 1.1 dargestellt sind. Die Eingänge sind immer auf der linken Seite, der Ausgang ist rechts. Das Dreieck im Symbol des Inverters steht für eine Weiterleitung oder Verstärkung, der Kreis gibt die Invertierung, also Umkehrung des Wertes an. Das Zeichen & steht für ‚und‘. Im ODER-Gatter meint

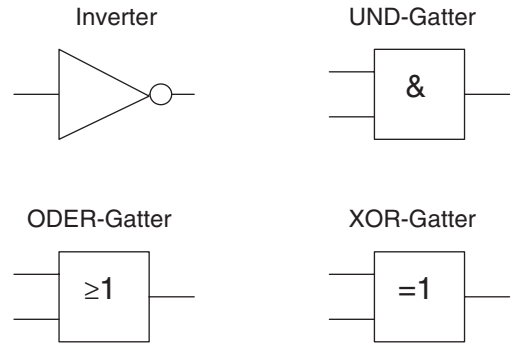


Abb. 1.1 Symbole für Logikgatter

die Bezeichnung ‚≥1', dass mindestens eine 1 am Eingang anliegen muss, damit der Ausgang 1 wird. Entsprechend bedeutet ‚=1' bei XOR, dass von zwei Eingängen exakt eine 1 vorhanden sein muss.

Mit diesen Grundelementen können Informationen miteinander verknüpft werden. Außerdem müssen in einer Digitalschaltung auch Informationen gespeichert werden und das Grundelement hierfür ist das *D-Flip-Flop (D-FF).* Dabei steht D für Daten und Flip-Flop symbolisiert das Hin- und Herschalten zwischen 0 und 1.

Das D-Flip-Flop arbeitet mit einem Takt, (engl. *Clock*), also einem periodischen Signal, welches die Arbeitsgeschwindigkeit einer Digitalschaltung vorgibt. Der Takt ist Ihnen möglicherweise von Ihrem PC bekannt. Eine moderne CPU arbeitet mit einem Takt von 2 bis 3 GHz, das heißt 2 bis 3 Milliarden mal pro Sekunde wechselt das Taktsignal von 0 auf 1. Schaltungen, die eine nicht ganz so hohe Rechengeschwindigkeit wie eine CPU haben, verwenden einen Takt mit geringerer Frequenz, beispielsweise 100 MHz.

Das Schaltsymbol des D-Flip-Flop (D-FF) ist in Abb. 1.2 dargestellt. Das Taktsignal ist am Eingang C1 (wie *Clock*) angeschlossen. Bei jeder Taktflanke, also einem Wechsel des Takts von 0 auf 1 wird der Wert am Dateneingang 1D gespeichert und unmittelbar darauf am Datenausgang ausgegeben. Diese Information wird für den Rest der Taktperiode gespeichert.

Logikgatter und D-FF werden aus Transistoren aufgebaut. Für ein Logikgatter sind rund 10, für ein D-FF rund 20 Transistoren erforderlich. In einer Digitalschaltung finden sich natürlich viele dieser Grundelemente.

1.1.4 Beispiel: Einfacher Grafikcontroller

Damit Sie sich die Arbeitsweise einer Digitalschaltung vorstellen können, soll eine Schaltung als Beispiel vorgestellt werden. Es handelt sich um einen Controller für ein einfaches Grafikmodul. Moderne PC-Grafikkarten sind sehr leistungsfähig und können realistische Bilder in hoher Geschwindigkeit erzeugen. Allerdings würde die Beschreibung eines solchen Grafikcontrollers wahrscheinlich das ganze Buch füllen. Die hier vorgestellte Schaltung ist deutlich einfacher zu verstehen und findet sich in Geräten mit geringen Grafikanforderungen. Sie entspricht auch in etwa den PC-Grafikkarten der 1980er Jahre.

Der Grafikcontroller setzt den Bildschirm aus einzelnen Zeichen zusammen. Für dieses Beispiel gehen wir davon aus, dass der Bildschirm 800 Bildpunkte breit und 600 Bildpunkte hoch ist. Jedes Zeichen soll 10 Bildpunkte breit und 15 Bildpunkte hoch sein.

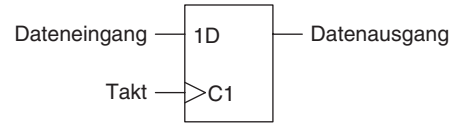


Abb. 1.2 Schaltsymbol des D-Flip-Flop (D-FF)

Damit passen 40 Zeilen mit je 80 Zeichen auf den Bildschirm. Ein Bild wird 60-mal je Sekunde also mit einer Frequenz von 60 Hz dargestellt.

Für die Zeichen gibt es einen festen Zeichensatz mit 128 Zeichen, darunter Buchstaben in Klein- und Großschreibung, Ziffern, Sonderzeichen und Symbole. Abb. 1.3 zeigt beispielhaft den Buchstaben A und die Ziffer 1 als 10 mal 15 Grafik.

Ein Prozessor teilt dem Grafikcontroller für jede Position mit, welches Zeichen dargestellt werden soll. Außerdem kann das Zeichen normal und invers dargestellt werden, das heißt bei invers ist der Hintergrund schwarz und das Zeichen weiß. Mit sieben Stellen wird eines der 128 Zeichen ausgewählt. Die achte Stelle gibt normale oder inverse Darstellung an. Damit ist für jedes Zeichen auf dem Bildschirm ein Byte, also ein Datenwort mit acht Stellen erforderlich.

Die Digitalschaltung des Grafikcontrollers benötigt einen Speicher für den aktuellen Bildschirminhalt, einen Speicher für die Grafiken der 128 Zeichen sowie zwei Zähler für die Zeile und Spalte, welche gerade dargestellt wird. Diese Schaltungsstruktur zeigt Abb. 1.4.

Der aktuelle Bildschirminhalt wird in einem Speicher abgelegt. Eine CPU schreibt für jede der 40 mal 80 Positionen ein Byte und bestimmt damit das darzustellende Zeichen.

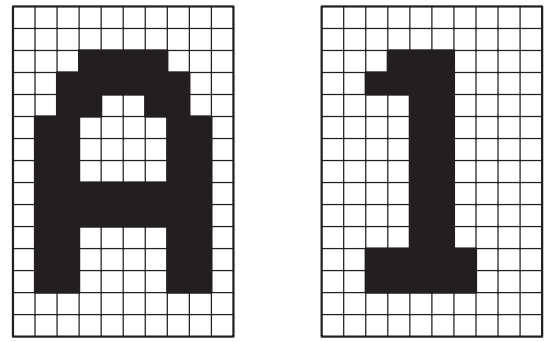


Abb. 1.3 Buchstabe A und Ziffer 1 als 10 mal 15 Grafik

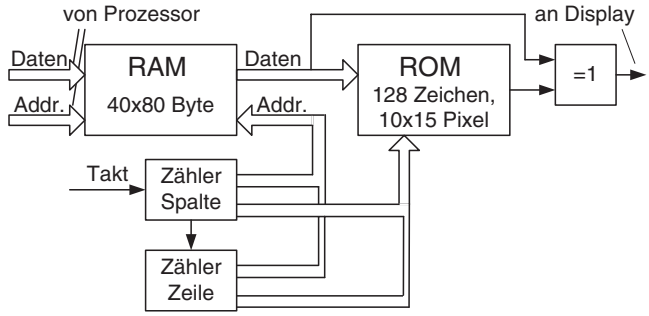


Abb. 1.4 Schaltungsstruktur eines einfachen Grafikcontrollers

Dieser Speicher mit der Kurzbezeichnung RAM *(Random Access Memory)* braucht also 3200 Speicherstellen zu jeweils einem Byte. Ein Festwertspeicher, Kurzbezeichnung ROM *(Read-Only-Memory),* enthält die 128 Zeichen zu je 10 mal 15 Bildpunkten, also 19.200 Speicherstellen zu jeweils einem Bit.

Der Grafikcontroller gibt das Bild zeilenweise aus. Die aktuell dargestellte Position wird durch zwei Zähler bestimmt, wobei ein erster Zähler die Spalte zählt. Wenn der Zähler an der letzten Spalte angekommen ist, wird der zweite Zähler aktiviert und so die nächste Zeile aufgerufen. Aus den Zählerwerten von Spalte und Zeile wird bestimmt, welches Zeichen gerade dargestellt wird.

Die Zählerwerte rufen zunächst das aktuelle Zeichen aus dem RAM auf. Dort steht zum Beispiel, dass der Buchstabe A angezeigt werden soll. Jetzt muss noch beachtet werden, welcher Bildpunkt des aktuellen Zeichens angezeigt wird, denn jedes Zeichen besteht ja aus 10 mal 15 Bildpunkten. Diese Information wird im ROM verarbeitet. Das ROM bekommt vom RAM das aktuelle Zeichen und von den Zählern die Information über die Position innerhalb des Zeichens. Für die linke obere Ecke des Buchstabens A wird dann zum Beispiel die Information „weißer Bildpunkt“ ausgegeben (siehe Abb. 1.3).

Für die Auswahl des Zeichens sind sieben Stellen eines Byte vorgesehen. Die achte Stelle kann durch ein XOR-Gatter den Helligkeitswert umdrehen, sodass eine inverse Darstellung entsteht.

Die Geschwindigkeit des Takts muss zu der Anzahl der Bildpunkte und der Bilder pro Sekunde passen. Aus 800 mal 600 Bildpunkten und 60 Bilder pro Sekunde berechnet sich theoretisch eine Frequenz von 28,8 MHz. In der Realität sind allerdings in horizontaler und vertikaler Richtung noch Abstände zwischen den aktiven Bildbereichen erforderlich, sogenannte Austastlücken. Daher wird bei der genannten Auflösung ein Takt von 40 MHz verwendet.

1.1.5 Beispiel: Zähler im Grafikcontroller

In einen Teil des Grafikcontrollers soll noch etwas detaillierter geschaut werden. Damit ein Zeichen auf dem Bildschirm dargestellt wird, muss die aktuelle Spalte an den ROM-Speicher gegeben werden. Hierzu wird ein Zähler eingesetzt, der nacheinander die Zahlen von 0 bis 9 ausgibt und danach wieder ab der 0 weiterzählt. Diese Schaltung ist ein Teil des Blocks „Zähler Spalte“ in Abb. 1.4.

Die Schaltung für so einen Zähler ist in Abb. 1.5 dargestellt. Der Zählerstand wird als *Dualzahl* dargestellt, das heißt, eine Ziffer Z besteht aus vier Stellen $z(3:0)$, wobei jede Stelle 0 oder 1 sein kann. Der Wert 0000 entspricht dem Zählerstand Null, 0001 entspricht Eins und so weiter. Die ausführliche Darstellung von Dualzahlen folgt später in Kapitel 2.

In der Schaltung von Abb. 1.5 wird der aktuelle Stand des Zählers in vier Flip-Flops für die vier Stellen der Zahl Z gespeichert. Aus dem aktuellen Wert wird mit einigen Gattern der neue Zählerstand berechnet. Der Takt sorgt für die Datenübernahme, das heißt

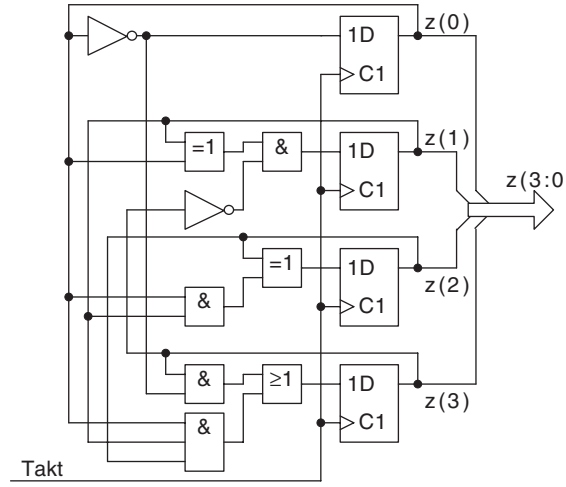


Abb. 1.5 Zähler im Grafikcontroller

bei Aktivierung übernehmen die vier Flip-Flops den neuen Zählerstand und schalten so eine Zahl weiter.

Die Flip-Flops dienen also zur Speicherung von Informationen, hier dem aktuellen Zählerstand. Die Gatter führen Rechnungen durch, ermitteln hier also den nächsten Wert des Zählers. Wie eine solche Schaltung entworfen wird, erfahren Sie in den folgenden Kapiteln.

1.2 Technische Realisierung digitaler Schaltungen

Eine Digitalschaltung kann auf verschiedene Art und Weise implementiert, also aufgebaut werden. Der Oberbegriff für eine Schaltungsimplementierung ist *Integrierte Schaltung,* englisch *Integrated Circuit* (IC). Der Begriff bezieht sich darauf, dass mehrere Transistoren auf dem gleichen Bauelement zusammengefasst, also integriert sind. Auf den ersten integrierten Schaltungen begann dies mit bis zu 50 Transistoren, heute können es über eine Milliarde Transistoren sein.

Weitere Bezeichnungen sind *Chip* und *Microchip*. Diese Begriffe beziehen sich auf das kleine Siliziumplättchen innerhalb eines ICs. In der Praxis werden diese Begriffe gleichbedeutend für IC verwendet.

Die wichtigsten Arten von ICs werden im Folgenden kurz vorgestellt.

1.2.1 Logikbausteine

Logikgatter und Flip-Flops sind als einzelne Bauelemente verfügbar. Eine Digitalschaltung kann aus diesen einzelnen *Logikbausteinen* aufgebaut werden. Es wird eine Vielzahl

verschiedener Bausteine angeboten, die in Tabellenbüchern und Datenblättern von den Herstellern beschrieben werden.

Ein Beispiel für einen Logikbaustein ist der IC 7408 mit vier Und-Gattern. Er ist in Abb. 1.6, dargestellt. Die jeweils zwei Eingänge und ein Ausgang der Und-Gatter sind auf Anschlussbeinchen, sogenannten Pins, aus dem Gehäuse herausgeführt und können mit anderen Bauelementen verbunden werden. Am Logikbaustein sind außerdem Versorgungsspannung (VDD) und Masse (GND) vorhanden, so dass der Baustein 14 Pins hat.

Sehr kleine Schaltungen, wie etwa der Zähler aus Abb. 1.5 können prinzipiell mit einzelnen Logikbausteinen realisiert werden. Für größere Digitalschaltungen wären jedoch viel zu viele Bausteine nötig. In der Praxis werden Logikbausteine eingesetzt, wenn kleine Schaltungen mit wenigen Gattern benötigt werden.

1.2.2 Kundenspezifische Integrierte Schaltung

Eine große Digitalschaltung kann aufgebaut werden, indem Logikgatter und Flip-Flop nach Bedarf verschaltet werden und dann eine Integrierte Schaltung nach diesem Bauplan hergestellt wird. So eine Schaltung wird als *Kundenspezifische Integrierte Schaltung* oder *ASIC (Application Specific Integrated Circuit)* bezeichnet.

Der Entwurf eines ASIC erfordert jedoch hohe Entwicklungskosten und eine relativ lange Entwicklungszeit. Die Entwicklung einer solchen Schaltung lohnt sich darum meist erst ab einer Stückzahl von 10 000, besser 100 000 ICs. Ein ASIC kann entweder nur in eigenen Produkten eingesetzt werden oder auch anderen Firmen zum Kauf angeboten werden.

1.2.3 Standardbauelemente

Für viele Aufgabenstellungen existieren fertige Digitalschaltungen, welche direkt eingesetzt werden können. Diese ICs werden als *ASSP* bezeichnet *(Application Specific*

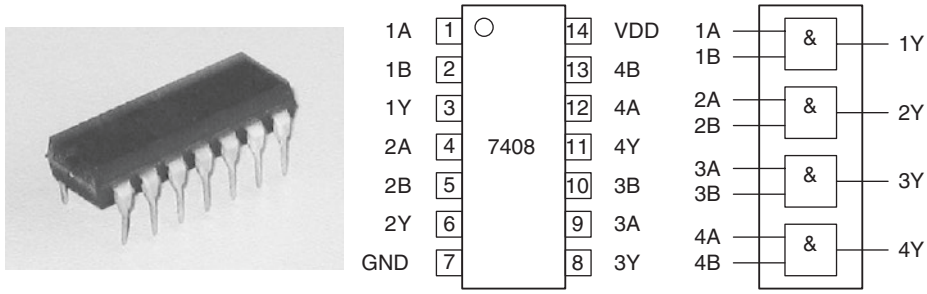


Abb. 1.6 IC 7408 mit vier Und-Gattern

Standard Product). Bekannte Beispiele hierfür sind Prozessoren und Speicherbausteine für Computer. Abb. 1.7 zeigt den Minicomputer Raspberry Pi 3, der auf der linken Seite einen IC mit Prozessor und Grafikcontroller enthält. Auf der rechten Seite ist ein etwas kleinerer IC, der für die Netzwerk- und USB-Verbindung sorgt.

Aber auch für viele andere Anwendungen sind ASSPs verfügbar. Wenn für eine Problemstellung ein ASSP verfügbar ist, kann damit meistens schnell und mit vertretbaren Kosten eine Schaltung aufgebaut werden.

1.2.4 Programmierbare Schaltung

Einen Mittelweg zwischen Standardbauelementen und ASIC bieten programmierbare Schaltungen, sogenannte *FPGAs (Field Programmable Gate Arrays).* Ein FPGA ist, genau wie ein ASSP, als IC direkt verfügbar. Anders als ein ASSP hat ein FPGA aber keine festgelegte Funktion, sondern wird vom Entwicklerteam programmiert.

Abb. 1.8 zeigt den prinzipiellen Aufbau eines FPGAs. Der Baustein enthält verschiedene Logikblöcke, die als Logikgatter und Flip-Flop programmiert werden können. Durch programmierbare Verbindungsleitungen und Ein-/Ausgänge können Schaltungen erstellt werden. Im Bild wird durch die fett gedruckten Elemente eine einfache Digitalschaltung implementiert.

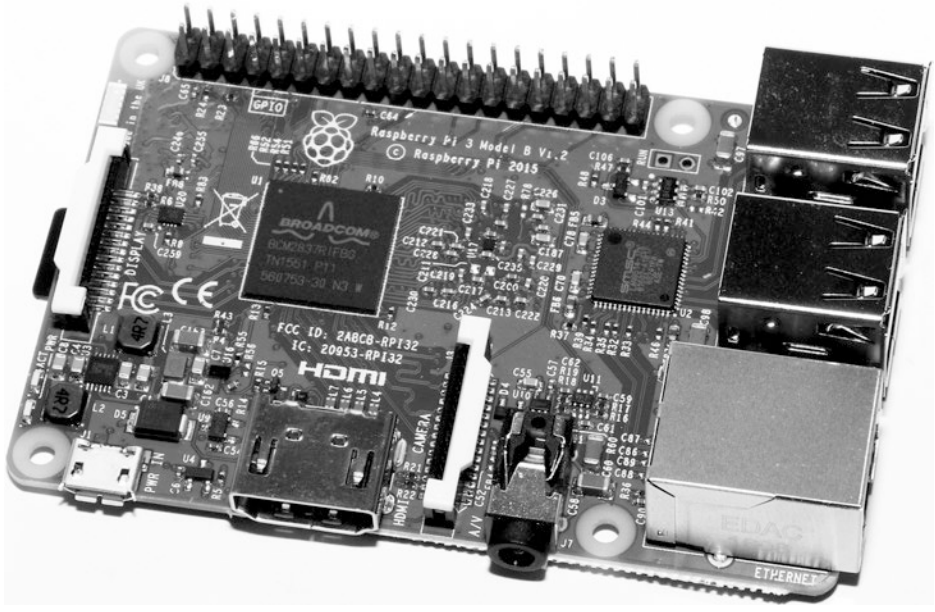


Abb. 1.7 Minicomputer Raspberry Pi

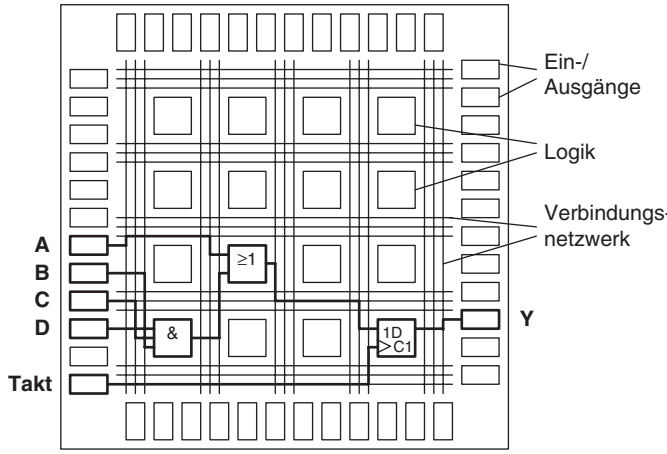


Abb. 1.8 Programmierbare Schaltung (FPGA)

Ein FPGA kann zehntausende Logikgatter und Flip-Flops enthalten. Im Vergleich zu ASICs sind Entwicklungskosten und Entwicklungszeit für eine FPGA-Schaltung geringer, sodass ein Produkt eher am Markt sein kann. Allerdings sind die Stückkosten und die Verlustleistung etwas höher.

Als Beispiel nehmen wir an, eine Firma möchte einen Monitor für medizinische Anwendungen entwickeln. Für die Darstellung von Röntgenbildern ist eine sehr hohe Abstufung von Grauwerten erforderlich.

- ASSPs zur Ansteuerung von Monitoren sind verfügbar. Sie sind jedoch nur für Computer-Anwendungen mit normaler Farbabstufung ausgelegt.
- Die Firma könnte ein eigenes ASIC als Grafikcontroller entwerfen. Der Markt für die geplanten Monitore ist jedoch nicht besonders groß und die Firma erwartet Verkaufszahlen von einigen hundert Monitoren pro Jahr. Für diese geringe Stückzahl lohnt sich der Entwurf eines ASIC nicht.
- Ein FPGA ist die bevorzugte Lösung zur Implementierung der Monitoransteuerung. Die Digitalschaltung kann mit der benötigten Farbabstufung aufgebaut werden. Da FPGAs als Komponente verfügbar sind, ist keine aufwendige Fertigung erforderlich.

1.2.5 Mikrocontroller

Zur Implementierung einer Digitalschaltung kann auch ein *Mikrocontroller* eingesetzt werden. Dabei handelt es sich um einen kleinen Computer, der komplett auf einem einzigen IC aufgebaut ist. Platzbedarf und Kosten sind viel geringer als bei einem PC; dafür ist allerdings auch die Rechenleistung beschränkt.

Ein Mikrocontroller kann genau wie ein FPGA für eine Anwendung programmiert werden. Anders als bei einem FPGA werden durch die Programmierung aber keine Logikgatter und Flip-Flops verschaltet. Die Funktion wird beim Mikrocontroller schrittweise als Computerprogramm ausgeführt. Leistungsfähigkeit und Flexibilität sind dadurch geringer als beim FPGA, aber für viele Anwendungen ausreichend.

1.3 Digitale und analoge Informationen

1.3.1 Darstellung von Informationen

Die Begriffe *digital* und *analog* beschreiben die Darstellung von Signalen. Die Aufgabe von analogen und digitalen Schaltungen ist oft die Verarbeitung von physikalischen Größen, wie Audiosignale, Bildsignale oder Sensorinformationen. Eine analoge Darstellung übersetzt eine physikalische Größe in eine andere, zweite physikalische Größe. Diese zweite physikalische Größe ist in der Elektronik normalerweise eine elektrische Spannung. Wenn beispielsweise ein Temperatursensor die Wassertemperatur misst, kann die Temperatur von 0° bis 100° C durch eine analoge Spannung von 0 bis 1 V dargestellt werden. Theoretisch kann ein analoges Signal beliebig viele Werte einnehmen.

Bei einem digitalen Signal ist die Anzahl der möglichen Werte festgelegt. Dies ist der wesentliche Unterschied zu einem analogen Signal. Wenn eine Wassertemperatur verarbeitet werden soll, kann beispielsweise festgelegt werden, dass eine Abstufung in 1°-Schritten sinnvoll ist. Das digitale Signal kann dann nur 101 verschiedene Werte einnehmen, also die Werte 0°, 1°, 2°, bis 100°. Diese Abzählbarkeit der möglichen Werte steckt auch hinter der Bezeichnung digital, denn das Wort *digit* hat eigentlich die Bedeutung „Finger“ und meint damit das Abzählen (per Finger).

Beispielsweise kann Musik auf analoger Schallplatte oder digitaler CD gespeichert werden. Bei der Schallplatte werden die Schallwellen in kleine Auslenkungen einer Rille übersetzt. Die Auslenkung repräsentiert somit das Musiksignal. Bei der CD wird das Musiksignal digital gespeichert. Pro Sekunde werden 44.100 Signalwerte als Zahl gespeichert. Mit 16 Bit pro Zahl sind 65.536 verschiedene Signalwerte möglich.

1.3.2 Vor- und Nachteile der Darstellungen

Analoge Signalverarbeitung hat den Vorteil, dass ein Signal theoretisch beliebig genau dargestellt werden kann. Digitale Signale haben eine begrenzte Genauigkeit, diese kann aber so gewählt werden, dass die Abstufungen ausreichend fein sind.

Die 65.536 möglichen Signalwerte der CD können störungsfrei ausgelesen und wiedergegeben werden. Die Schallplatte hat theoretisch eine unbegrenzte Auflösung. Diese wird durch die kleinen Abmessungen der Schallplattenrille sowie durch Staub und

Abnutzung allerdings in der Realität eher schlechter als bei der CD sein. Natürlich dürfen Fans der Schallplatte trotzdem ihrem Medium treu bleiben.

Die Verarbeitung analoger Signale war in der Vergangenheit oft einfacher als bei digitalen Schaltungen. Durch die Leistungsfähigkeit moderner Digitalschaltungen haben sich die Verhältnisse umgedreht. Heutzutage ist die Verarbeitung digitaler Signale fast immer einfacher. Hinzu kommt die problemlose Speicherung und Übertragung digitaler Informationen, die Vorteile gegenüber der analogen Darstellung bietet.

Als Beispiel nehmen wir an, dass ein aktuelles Bild von einer Sportveranstaltung für einen Zeitungsartikel benötigt wird. Ein analoges Foto auf Filmmaterial wurde früher zunächst chemisch entwickelt, das passende Bild wurde ausgewählt und persönlich oder per Kurier in die Redaktion gebracht. Heute kann auf einer Digitalkamera sofort das passende Bild ausgewählt und per Mobiltelefon als Email in die Redaktion geschickt werden. Innerhalb von Minuten ist eine Veröffentlichung auf der Homepage möglich.

Digitale Systeme haben in vielen Anwendungen die analogen Techniken abgelöst:

- Audiosignale werden nicht mehr analog auf Schallplatte und Musikkassette, sondern digital auf CD und als MP3 gespeichert.
- Videosignale werden nicht mehr analog auf VHS-Band, sondern digital als MPEG auf Festplatten, DVD und Blu-Ray gespeichert.
- Das analoge Telefon wurde zunächst durch digitales ISDN und mittlerweile durch Voice-over-IP ersetzt.
- Fotos werden kaum noch auf chemischem Filmmaterial, sondern meist als digitale JPEG-Datei gemacht.

Allerdings sind noch nicht alle Anwendungen digital. Für Radio gibt es zwar digitale Übertragung, das analoge UKW-Radio wird aber weiter verwendet. Gründe für die Beibehaltung von UKW-Radio sind die ausreichende Qualität, der einfache Aufbau analoger Radios sowie die Vielzahl von vorhandenen Geräten.

1.3.3 Wert- und zeitdiskret

Die digitale Darstellung von Signalen wird durch die Fachbegriffe *wertdiskret* und *zeitdiskret* beschrieben. Das Wort *diskret* bedeutet dabei voneinander abgetrennt, einzelstehend.

Mit wertdiskret ist gemeint, dass für die Signalwerte nur bestimmte, einzelne Werte möglich sind. Das Gegenteil ist *wertkontinuierlich,* das heißt es gibt keine Lücken zwischen den möglichen Werten.

Mit zeitdiskret ist gemeint, dass die Signalwerte nur zu bestimmten Zeiten vorhanden sind. Das Gegenteil ist *zeitkontinuierlich,* das heißt zu jeder Zeit ist das Signal definiert.

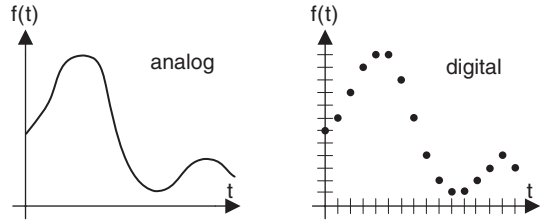


Abb. 1.9 Verlauf eines analogen und digitalen Signals

Betrachten wir wieder CD und Schallplatte:

- Ein Musiksignal auf einer CD ist wertdiskret, denn es sind fest definierte 65.536 verschiedene Werte möglich. Und es ist zeitdiskret, denn pro Sekunde sind genau 44.100 Signalwerte definiert. Die Werte zwischen diesen Zeitpunkten sind nicht abgespeichert. Für die Wiedergabe kann man diese Zwischenwerte problemlos interpolieren, aber sie sind nicht auf der CD enthalten.
- Ein Musiksignal auf Schallplatte ist wertkontinuierlich, denn die Schallplattenrille ist stufenlos verschoben. Und es ist zeitkontinuierlich, denn die Rille hat keine Lücke. Für jede Position, also für jeden Zeitpunkt ist eine Verschiebung der Rille vorhanden.

Abb. 1.9 zeigt ein analoges und ein digitales Signal im Zeitverlauf. Das analoge Signal ist durchgängig über der horizontalen Zeitachse und der vertikalen Werteachse. Das digitale Signal ist nur zu bestimmen Zeiten definiert und kann nur bestimmte Werte einnehmen. Die Schrittweite im digitalen Signal ist zur Verdeutlichung sehr groß gewählt. In der Realität sind die Abstände so klein, dass ein digitales Signal keine erkennbaren Stufen zeigt.

Digitale Signale sind also wertdiskret und zeitdiskret, analoge Signale sind wertkontinuierlich und zeitkontinuierlich. Es gibt Spezialfälle von wertdiskret und zeitkontinuierlich oder zeitdiskret und wertkontinuierlich. Diese werden jedoch nicht gesondert betrachtet, sondern sind meist analog. Ein solcher Spezialfall sind Kinofilme auf Filmrolle. Pro Sekunde sind typischerweise 24 Einzelbilder vorhanden (zeitdiskret), die Farbinformationen der einzelnen Bilder sind stufenlos (wertkontinuierlich).

1.4 Übungsaufgaben

Haben Sie den Inhalt des Kapitels verstanden? Prüfen Sie sich mit den Fragen am Kapitelende. Die Antworten finden Sie am Ende des Buches.

Bei allen Auswahlfragen ist immer genau eine Antwort korrekt.

Aufgabe 1.1
Was gilt IMMER für Binärdaten?

a) Binärdaten stellen einen Zahlenwert dar
b) Binärdaten arbeiten mit 0 und 3,3 V
c) Es gibt zwei Zustände

Aufgabe 1.2
Was gilt IMMER für einen Inverter?

a) Ein Inverter hat eine Verzögerungszeit von 1 ns
b) Eine 0 am Eingang wird zu einer 1 am Ausgang
c) Wenn am Eingang 3,3 V anliegt, ergibt der Ausgang 0 V

Aufgabe 1.3
Was gilt für ein UND-Gatter?

a) Nur wenn genau ein Eingang 1 ist, ist der Ausgang 1
b) Wenn mindestens ein Eingang 1 ist, ist der Ausgang 1
c) Nur wenn alle Eingänge 1 sind, ist der Ausgang 1

Aufgabe 1.4
Was gilt für ein ODER-Gatter?

a) Nur wenn alle Eingänge 1 sind, ist der Ausgang 1
b) Wenn mindestens ein Eingang 1 ist, ist der Ausgang 1
c) Nur wenn genau ein Eingang 1 ist, ist der Ausgang 1

Aufgabe 1.5
Was gilt für ein XOR-Gatter (mit zwei Eingängen)?

a) Wenn mindestens ein Eingang 1 ist, ist der Ausgang 1
b) Nur wenn alle Eingänge 1 sind, ist der Ausgang 1
c) Nur wenn genau ein Eingang 1 ist, ist der Ausgang 1

Aufgabe 1.6
Was gilt für ein UND-Gatter?

a) Nur wenn alle Eingänge 0 sind, ist der Ausgang 0
b) Wenn mindestens ein Eingang 0 ist, ist der Ausgang 0
c) Nur wenn genau ein Eingang 0 ist, ist der Ausgang 0

Aufgabe 1.7
Was gilt für ein ODER-Gatter?

a) Wenn mindestens ein Eingang 0 ist, ist der Ausgang 0
b) Nur wenn alle Eingänge 0 sind, ist der Ausgang 0
c) Nur wenn genau ein Eingang 0 ist, ist der Ausgang 0

Aufgabe 1.8
Was gilt für ein XOR-Gatter (mit zwei Eingängen)?

a) Wenn mindestens ein Eingang 0 ist, ist der Ausgang immer 1
b) Nur wenn alle Eingänge 0 sind, ist der Ausgang 1
c) Nur wenn genau ein Eingang 0 ist, ist der Ausgang 1

Aufgabe 1.9
Was gilt für ein D-Flip-Flop (D-FF)?

a) Wenn Daten und Takt den gleichen Wert haben, wechselt der Ausgang
b) Wenn Daten und Takt einen ungleichen Wert haben, wechselt der Ausgang
c) Daten werden bei einer Taktflanke gespeichert
d) Daten werden bei Takt gleich 1 gespeichert
e) Daten werden bei Takt gleich 0 gespeichert

Aufgabe 1.10
Welche Eigenschaften hat ein digitales Signal?

a) wertdiskret und zeitkontinuierlich
b) wertdiskret und zeitdiskret
c) wertkontinuierlich und zeitkontinuierlich
d) zeitdiskret und wertkontinuierlich

Digitale Codierung von Informationen

2

Genau wie wir Menschen verarbeiten auch digitale Systeme Informationen, die sie aus ihrer Umgebung erhalten.

Lesen Sie zum Beispiel den Wetterbericht in der Tageszeitung und erhalten die Information, dass mit Regen zu rechnen ist, nehmen Sie einen Schirm mit, wenn Sie das Haus verlassen. Wird dagegen wolkenloses Sommerwetter angekündigt, ist die Mitnahme einer Sonnenbrille vermutlich die bessere Entscheidung.

Um als Mensch eine Information aufnehmen und verarbeiten zu können, muss sie in einer für uns zugänglichen Form vorliegen. Der Wetterbericht in der Zeitung besteht aus einzelnen Zeichen, die wir zu Wörtern und Sätzen zusammenfügen. Die in den Sätzen enthaltene, man kann auch sagen „codierte", Information extrahieren wir und reagieren entsprechend. Allerdings hätten wir große Schwierigkeiten den Wetterbericht zu verstehen, wenn er in einer uns unbekannten Sprache verfasst wäre. Da wir die Regeln nicht kennen, die beschreiben wie die Information durch die Aneinanderreihung der Buchstaben codiert ist, könnten wir mit dem scheinbaren Buchstabensalat nichts anfangen.

Wie lassen sich diese Überlegungen auf ein digitales System übertragen? Zunächst ist es selbstverständlich wichtig, dass die zu verarbeitenden Informationen in digitaler Form, also als Bits, vorliegen. Darüber hinaus müssen aber auch Regeln vereinbart sein, die die Bedeutung der Bits beschreiben. Andernfalls kann ein digitales System die in den Bits enthaltene Information nicht extrahieren – es kann mit dem „Bitsalat" nichts anfangen.

In diesem Kapitel werden einige Regeln zur digitalen Codierung von Informationen vorgestellt, die die Grundlage für die Realisierung vieler digitaler Schaltungen darstellen. Da in vielen praktischen Anwendungsfällen Zahlenwerte verarbeitet werden, liegt der Schwerpunkt dieses Kapitels auf der binären Codierung von Zahlen. In diesem Kapitel werden darüber hinaus einige gebräuchliche Codes vorgestellt, die sich zur Codierung sowohl numerischer als auch nicht-numerischer Informationen eignen.

W. Gehrke et al., *Digitaltechnik,* Springer-Lehrbuch,
DOI 10.1007/978-3-662-49731-9_2

2.1 Grundlagen

Für die binäre Codierung einer Information werden Codewörter definiert, die aus Bits zusammengesetzt sind. Je mehr Bits zur Anwendung kommen, desto mehr Codewörter können definiert werden: Wird ein Bit verwendet, ergeben sich die zwei möglichen Codierungen „0“ und „1“. Mit 2 Bits ergeben sich bereits 4 Möglichkeiten, „00“, „01“, „10“ und „11“. Allgemein gilt, dass die maximale Anzahl der Codewörter eine Zweierpotenz ist. Mit n Bits lassen sich 2^n unterschiedliche binäre Codierungen darstellen. Ausgewählte Zweierpotenzen sind in Tab. 2.1 dargestellt.

Für Zehnerpotenzen sind Vorsätze klar definiert. Zum Beispiel steht k (Kilo) für 10^3, M (Mega) für 10^6 oder G (Giga) für 10^9. Als die Vorsätze für Zweierpotenzen eingeführt wurden, orientierte man sich an den bekannten Vorsätzen für Zehnerpotenzen. Da $2^{10} \approx 10^3$ ist, setzte man den Zehnerpotenzvorsatz Kilo auch für die Zweierpotenz ein. Zur Unterscheidung wurde teilweise der Zweierpotenzvorsatz K anstelle von k verwendet. Weiterhin sind dann die Abkürzungen M für $2^{20} \approx 10^6$, G für $2^{30} \approx 10^9$ und T für $2^{40} \approx 10^{12}$ eingeführt worden. Hier war jedoch eine Unterscheidung mittels Groß- und Kleinschreibung nicht mehr möglich und es gibt das Problem einer möglichen Zweideutigkeit. Gibt zum Beispiel ein Hersteller die Kapazität einer Festplatte mit 5,0 TByte an, so meint er in der Regel $5 \cdot 10^{12}$ Byte und nicht $5 \cdot 2^{40}$ Byte. Die Differenz beträgt immerhin fast 10 %.

Weitere Probleme entstehen bei der Kennzeichnung von Übertragungsgeschwindigkeiten. In Datenübertragungsnetzen sind die Bezeichnungen kbit/s, Mbit/s und Gbit/s üblich. Hier sind die üblichen Abkürzungen für Zehnerpotenzen gemeint. Um die Zweideutigkeit der Vorsätze zu vermeiden hat das internationale Normierungsgremium IEC

Tab. 2.1 Ausgewählte Zweierpotenzen

n	1	2	3	4	5	6	7	8	9	16	20	30
2^n	2	4	8	16	32	64	128	256	512	65.536	1.048.576	1.073.741.824

Tab. 2.2 Binäre Vorsätze für Zweierpotenzen

Zweierpotenz	Abkürzung (gesprochen)	Abgeleitet von	z. B. Speicherkapazität in bit	z. B. Speicherkapazität in Byte
2^{10}	Ki (Kibi)	Kilobinär	Kibit	KiB (= 8 Kibit)
2^{20}	Mi (Mebi)	Megabinär	Mibit	MiB (= 8 Mibit)
2^{30}	Gi (Gibi)	Gigabinär	Gibit	GiB (= 8 Gibit)
2^{40}	Ti (Tebi)	Terabinär	Tibit	TiB (= 8 Tibit)
2^{50}	Pi (Pebi)	Petabinär	Pibit	PiB (= 8 Pibit)
2^{60}	Ei (Exbi)	Exabinär	Eibit	EiB (= 8 Eibit)
2^{70}	Zi (Zebi)	Zettabinär	Zibit	ZiB (= 8 Zibit)
2^{80}	Yi (Yobi)	Yottabinär	Yibit	YiB (= 8 Yibit)

(International Electrotechnical Commission) in der Norm IEC 60027 neue Vorsätze für binäre Vielfache festgelegt. In Tab. 2.2 sind diese Vorsätze zusammengefasst.

Die IEC-Norm hat sich bisher nur zum Teil in der Praxis verbreitet. In vielen Fällen werden die Vorsätze für Zehnerpotenzen verwendet, obwohl eigentlich Vorsätze für Zweierpotenzen gemeint sind.

2.2 Vorzeichenlose Zahlen

In diesem Abschnitt werden Zahlendarstellungen und grundlegende arithmetische Operationen für vorzeichenlose duale Ganzzahlen erläutert. Der betrachtete Zahlenraum umfasst also die natürlichen Zahlen inklusive der Null.

2.2.1 Stellenwertsysteme

Wenn Sie die Ziffernfolge „123“ sehen, werden Sie diese vermutlich sofort mit dem Zahlenwert Einhundertdreiundzwanzig verbinden. Wir haben in unseren ersten Schuljahren gelernt, dass Zahlen durch einzelne Zeichen dargestellt werden, die hintereinander geschrieben einen Zahlenwert repräsentieren. Die am weitesten rechts stehende Ziffer ist die Einerstelle. Diese wird gefolgt von der Zehnerstelle und der Hunderterstelle. Sollen größere Zahlenwerte dargestellt werden, werden einfach weitere Stellen hinzugefügt. Diese Vereinbarung legen wir im Alltag bei der „Decodierung“ einer Ziffernfolge zugrunde.

Man kann die im Alltag verwendete Vereinbarung auch mathematisch als Formel darstellen. Der Zahlenwert Z_{10} einer Folge von N Ziffern, die aus den Ziffern z_{N-1} bis z_0 besteht, ergibt sich aus der Formel:

$$Z_{10} = \sum_{i=0}^{\mathrm{N}-1} z_i \cdot 10^i$$

Als Ziffernzeichen werden die zehn Symbole 0,1, … 8,9 verwendet, denen jeweils ein Zahlenwert im Bereich von Null bis Neun zugeordnet ist.

Diese Form der Zahlendarstellung nennt man Stellenwertsystem. Jeder Stelle einer Ziffernfolge ist ein *Stellengewicht* zugeordnet. Im Dezimalsystem ist dies eine Zehnerpotenz. Die Summe der einzelnen Produkte aus *Stellenwert* und Stellengewicht ergibt den dargestellten Zahlenwert.

Dass wir im Alltag zehn unterschiedliche Symbole zur Darstellung der Ziffern verwenden, ist eine willkürliche Festlegung. Man kann zum Beispiel auch die Vereinbarung treffen, ein Siebener-System zu verwenden. Dann würden die Symbole 7, 8 und 9 nicht benötigt und es gälte die Rechenregel:

$$Z_7 = \sum_{i=0}^{N-1} z_i \cdot 7^i$$

Da sich somit eine Einerstelle, eine Siebenerstelle und eine Neunundvierzigerstelle ergibt, würde die Ziffernfolge „123“ dem Zahlenwert Sechsundsechzig entsprechen.

Diese Überlegungen lassen sich auf beliebige Anzahlen von Ziffernsymbolen erweitern. Werden B Ziffernsymbole verwendet, ergibt sich der codierte Zahlenwert aus der Formel:

$$Z_B = \sum_{i=0}^{N-1} z_i \cdot B^i$$

Der Wert B wird als Basis des jeweiligen Zahlensystems bezeichnet und man spricht von einer Zahlendarstellung „zur Basis B“ oder von einem B-adischen Zahlensystem. Um die verwendete Basis explizit deutlich zu machen, kann sie als Index an die Ziffernfolge angefügt werden. Zum Beispiel gilt:

$$66_{10} = 102_8 = 123_7 = 1002_4 = 2110_3$$

In vielen Fällen wird jedoch auf den Index verzichtet, da aus dem Zusammenhang bereits deutlich wird, welche Basis verwendet wird.

Einer der Vorteile der hier vorgestellten Stellenwertsysteme gegenüber anderen Zahlensystemen ist die einfache Möglichkeit alle vier Grundrechenarten mit übersichtlichen Regeln umzusetzen.

Eine Zahlendarstellung, die nicht auf Stellenwertigkeiten basiert, ist beispielsweise das Römische Zahlensystem. Eine Addition lässt sich in diesem System durch „Zusammenziehen“ der beiden Operanden relativ einfach realisieren. Eine Multiplikation ist dagegen deutlich aufwendiger als im dezimalen Stellenwertsystem.

2.2.2 Darstellung vorzeichenloser Zahlen in der Digitaltechnik

Zur Implementierung digitaler Systeme werden nur zwei Zustände verwendet. Daher ist es konsequent, genau zwei Ziffernsymbole zu verwenden. Es wird also die Basis 2 für die Darstellung von Zahlen gewählt. Eine Zahl wird in diesem *Dualsystem* durch eine Folge von Nullen und Einsen dargestellt und ergibt sich entsprechend der Überlegungen des vorangegangenen Abschnitts zu:

$$Z_2 = \sum_{i=0}^{N-1} z_i \cdot 2^i$$

Selbst bei relativ kleinen Zahlen ergibt sich hierbei schnell eine große Stellenzahl. So kann der dezimale Wert 98_{10} im Dezimalsystem mit zwei Ziffern dargestellt werden. Im Dualsystem werden dagegen mindestens 7 Stellen benötigt: $98_{10} = 1100010_2$.

Um die Darstellung dualer Zahlen übersichtlicher zu gestalten, können mehrere Bits einer Dualzahl zusammengefasst werden. So können zum Beispiel 3 Bits zu einer neuen Ziffer kombiniert werden. Der Wert dieser neuen Ziffer kann 8 verschiedene Werte annehmen. Man erhält ein Zahlensystem zur Basis 8, das Oktalsystem.

In der Praxis wird sehr häufig eine Gruppierung von jeweils vier Bits vorgenommen. Dieses ist insbesondere dann sinnvoll, wenn die Zahlenwerte mit Vielfachen von vier Bits codiert werden, was bei allen heute üblichen Rechnersystemen der Fall ist. Da sich bei einer Kombination von vier Bits zu einer neuen Ziffer 16 mögliche Werte ergeben, reichen die Ziffernsymbole des Dezimalsystems nicht mehr aus. Es werden neben den Symbolen 0 bis 9 noch sechs weitere Symbole für die Werte 10 bis 15 benötigt. Hierfür werden die ersten Buchstaben des Alphabets verwendet. Auf diese Weise erhält man das sogenannte Hexadezimalsystem, ein Stellenwertsystem zur Basis 16.

Die verschiedenen Darstellungen von Zahlenwerten in unterschiedlichen Zahlensystemen fasst Tab. 2.3 für die Zahlen von 0 bis 18_{10} zusammen. Bei der Verwendung des Oktal- oder des Hexadezimalsystems arbeitet die zugrundeliegende digitale Hardware weiterhin mit einzelnen Bits, also im Dualzahlensystem. Die Kombination von Bits zu einer Oktal- oder Hexadezimalziffer dient lediglich der kompakteren Darstellung der Zahlenwerte.

Tab. 2.3 Darstellung der Zahlen 0 bis 18 im Dezimal-, Dual-, Oktal- und Hexadezimalsystem

Dezimal $B = 10$	Dual $B = 2$	Oktal $B = 8$	Hexadezimal $B = 16$
0	0	0	0
1	1	1	1
2	10	2	2
3	11	3	3
4	100	4	4
5	101	5	5
6	110	6	6
7	111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F
16	10000	20	10
17	10001	21	11
18	10010	22	12

2.2.3 Umwandlung zwischen Zahlensystemen

Für die Umrechnung eines Zahlenwertes aus einem System zur Basis B_1 in ein System zur Basis B_2 kann direkt unter Verwendung der bereits vorgestellten Summenformel erfolgen:

$$Z = \sum_{i=0}^{N-1} z_i \cdot B_1^i$$

Hierbei muss die Berechnung zur Basis B_2 erfolgen. Das Rechnen in einem anderem als dem dezimalen Zahlensystem ist jedoch gewöhnungsbedürftig, sodass es sich empfiehlt zunächst eine Umwandlung der Zahl in das Dezimalsystem vorzunehmen. In einem zweiten Schritt erfolgt dann die Umwandlung des Dezimalwertes in das gewünschte Zahlensystem zur Basis B_2.

Die Umrechnung aus dem Dezimalsystem in ein anderes Zahlensystem kann mithilfe der Divisionsmethode erfolgen, die im Folgenden vorgestellt wird.

Die Divisionsmethode basiert auf einem iterativen Vorgehen, bei dem zunächst die Ausgangszahl ganzzahlig durch die Basis B_2 des Zielsystems dividiert wird. Der Rest der Division ergibt eine Stelle der zu berechnenden Zahl. Anschließend wird der Quotient der Division wiederum durch B_2 dividiert. Dieses Vorgehen wird so lange wiederholt, bis der berechnete Quotient Null ist. Die gesuchte Zahlendarstellung ergibt sich aus den berechneten Resten, wobei der zuerst berechnete Rest die Einerstelle repräsentiert.

Die Umwandlung einer Zahl zur Basis B_1 in eine Zahl zur Basis B_2 kann wie folgt als iteratives Vorgehen formuliert werden:

1. Umwandlung der Ausgangzahl in das Dezimalsystem (Summenformel anwenden).
2. Ganzzahl-Division durch B_2.
3. Rest der Division ergibt eine Stelle der gesuchten Zahl.
4. Falls Quotient ungleich Null: Zurück zu Schritt 2. Der Dividend der erneuten Division ist der zuvor berechnete Quotient.

2.2.4 Beispiele zur Umwandlung zwischen Zahlensystemen

Beispiel 1
Die Zahl 110010_2 soll in eine Dezimalzahl umgewandelt werden. Hier kann die Summenformel direkt angewendet werden:

$$Z = \sum_{i=0}^{N-1} z_i \cdot 2^i = 1 \cdot 2^1 + 1 \cdot 2^4 + 1 \cdot 2^5 = 2 + 16 + 32 = 50$$

Die gesuchte Dezimalzahl ist 50.

Beispiel 2
Die Zahl 89_{10} soll in eine binäre Zahl umgewandelt werden. Mit der Divisionsmethode ergibt sich die in Tab. 2.4 dargestellte Rechnung und damit die gesuchte binäre Repräsentation 1011001_2.

Beispiel 3
Die Zahl $83ED_{16}$ soll in eine Dualzahl überführt werden. Die Umrechnung zwischen dem Dualzahlensystem und dem Hexadezimalsystem kann sehr einfach erfolgen, da 4 Bit einer Dualzahl exakt einer Stelle der Hexadezimalzahl entsprechen. Man benötigt lediglich die Zuordnung einer hexadezimalen Ziffer zu ihrem dualen Äquivalent (vgl. Tab. 2.3) und kann die Umwandlung direkt durch Ablesen aus der Tabelle durchführen. Die einzelnen Hexadezimalstellen werden sukzessive durch ihre binären Entsprechungen ersetzt und es ergibt sich:

$$83ED_{16} = 1000\,0011\,1110\,1101_2$$

Beispiel 4
Die Dualzahl 1011111011101111_2 soll in eine Hexadezimal gewandelt werden. Nach der Gruppierung der Dualzahl in Gruppen zu jeweils 4 Bit ergibt sich das Ergebnis wiederum durch Ablesen aus Tab. 2.3:

$$1011\,1110\,1110\,1111_2 = BEEF_{16}$$

Beispiel 5
Die Zahl 14505_6 soll in eine Oktalzahl umgewandelt werden. In diesem Fall bietet sich ein Vorgehen in zwei Schritten an.

Zunächst wird die gegebene Zahl mithilfe der Summenformel in eine Dezimalzahl umgewandelt und es ergibt sich

$$14505_6 = 2345_{10}$$

Anschließend erfolgt die Umwandlung in das Zielsystem mithilfe der Divisionsmethode (vgl. Tab. 2.5) Die gesuchte Oktalzahl lautet 4451_8.

Tab. 2.4 Umwandlung der Dezimalzahl 89 in eine Dualzahl

Iteration	Dividend	Divisor	Quotient	Rest
1	89	2	44	**1**
2	44	2	22	**0**
3	22	2	11	**0**
4	11	2	5	**1**
5	5	2	2	**1**
6	2	2	1	**0**
7	1	2	0	**1**

Tab. 2.5 Umwandlung der Dezimalzahl 2345 in eine Oktalzahl

Iteration	Dividend	Divisor	Quotient	Rest
1	2345	8	293	**1**
2	293	8	36	**5**
3	36	8	4	**4**
4	4	8	0	**4**

2.2.5 Wertebereiche und Wortbreite

Für alle Zahlendarstellungen gilt, dass mit einer konkreten Anzahl an Stellen nur eine begrenzte Anzahl von Zahlenwerten dargestellt werden kann. Besitzt eine Dezimalzahl beispielsweise drei Stellen, kann diese nur die Werte von 0 bis 999 annehmen. Mit einer 7-stelligen Dualzahl kann nur der Zahlenbereich von 0 bis $1111111_2 = 127_{10}$ dargestellt werden.

Werden zwei Dualzahlen addiert, kann es (je nach Zahlenwerten) passieren, dass für die Summe mehr Bits als für die beiden Operanden benötigt werden. So kann beispielsweise die Summe der Zahlen 1101_2 (13_{10}) und 0101_2 (5_{10}) nicht mit 4 Bit dargestellt werden. Für das Ergebnis 18_{10} werden 5 Bit benötigt ($18_{10} = 10010_2$).

Generell gilt, dass bei der Addition von n binären Zahlen $log_2(n)$ zusätzliche Bits für das Ergebnis benötigt werden. Addiert man beispielsweise 8 Zahlen mit der Wortbreite 6 Bit, muss für das Ergebnis eine Wortbreite von $6 + log_2(8) = 9$ *Bit* vorgesehen werden.

Vermutlich finden Sie diese Erkenntnis nicht sonderlich bemerkenswert, da wir aus dem täglichen Leben daran gewöhnt sind, dass das Ergebnis einer Rechnung mehr Stellen als die Operanden benötigt. Zur Veranschaulichung dieses Sachverhalts wird bereits in den ersten Jahren der Schulausbildung der Zahlenstrahl eingeführt. Mit ihm lassen sich unter anderem auch die Addition und Subtraktion übersichtlich grafisch darstellen. Durchläuft man den Zahlenstrahl von Null in Richtung positiver Zahlen, wird mit jedem Schritt eine 1 addiert (Additionsrichtung). Durchlaufen des Zahlenstrahls in entgegengesetzter Richtung entspricht der Subtraktion (Subtraktionsrichtung). Je weiter man sich auf dem Zahlenstrahl vom Wert Null entfernt, desto größer werden die Zahlen. An der Grenze zu einer Zehnerpotenz (zum Beispiel 99) wird die Anzahl der Stellen zur Darstellung der Zahlen erhöht (statt zwei Stellen für 99 werden drei Stellen für die Darstellung des Wertes 100 verwendet).

Für ein digitales System ist dieses Prinzip jedoch schwer umsetzbar. Ist ein System einmal realisiert, steht nur eine feste Anzahl von Stellen in der Hardware zur Verfügung. Das Prinzip „ich nehme mir so viele Stellen wie ich brauche“ funktioniert in digitalen Systemen daher nicht. Hieraus ergeben sich einige Konsequenzen für die arithmetischen Komponenten eines digitalen Systems, die im folgenden Abschnitt näher erläutert werden.

2.2.6 Zahlendarstellung mit begrenzter Wortbreite

Stellen Sie sich vor, Sie sollen ein digitales System realisieren, dass intensiv von der Addition Gebrauch macht. Für die Implementierung der Addierer des Systems könnten Sie sich entscheiden, dass immer die benötigte Anzahl von Ergebnisbits zur Verfügung stehen soll, das Ergebnis also ein Bit mehr als die Operanden umfasst. Allerdings ist zu beachten, dass die Wortbreite der Ergebnisse mit zunehmender Anzahl durchgeführter Additionen kontinuierlich wächst. Besitzen die Eingangswerte des Systems zum Beispiel eine Wortbreite von 8 bit, würde das Ergebnis einer ersten Addition eine Wortbreite 9 bit benötigen. Werden die so berechneten Zwischenergebnisse mit einer weiteren Addition weiterverarbeitet, sind bereits 10 bit für diese Ergebnisse erforderlich.

Selbstverständlich kann man ein digitales System realisieren, das beispielsweise vier 8-Bit-Zahlen addieren kann und ein Ergebnis mit der Wortbreite 10 bit liefert. Aber stellen Sie sich vor, Sie sollen eine arithmetische Komponente für ein Rechnersystem entwerfen. Sie wissen nicht welches Programm später auf dem Rechner laufen wird und welche Wortbreiten für Operanden und Ergebnisse sinnvoll sind. Darüber hinaus besitzen die Speicherstellen eines Rechners, in denen auch Zwischenergebnisse abgelegt werden, feste Wortbreiten (meist Vielfache eines Bytes). Daher verwenden die arithmetischen Einheiten eines Rechners meist identische Operanden- und Ergebniswortbreiten. Ergibt sich bei einer Berechnung ein Ergebnis, das eine größere Wortbreite als die implementierte Ergebniswortbreite benötigt, werden die führenden Bits des Ergebnisses einfach weggelassen. Die Ausgabe der arithmetischen Einheit wäre in diesem Fall also nicht korrekt. Nehmen wir an, dass mithilfe eines Addierers die Zahlen $1011_2 = 11_{10}$ und $1001_2 = 9_{10}$ addiert werden. Es steht ein Addierer mit einer Wortbreite von 4 bit zur Verfügung. Der Addierer kann also Operanden und Ergebnisse im Bereich von 0 bis 15 verarbeiten bzw. ausgeben. Das korrekte Ergebnis der Summe aus 11 und 9 ist jedoch 20 und überschreitet damit den möglichen Zahlenbereich der Ergebnisse des 4-Bit-Addierers. Statt des korrekten Ergebnisses 10100_2 wird der Addierer führende 1 verwerfen und $0100_2 = 4_{10}$ ausgegeben.

Was bedeutet die begrenzte Wortbreite für die grafische Darstellung von Zahlen? Am Beispiel eines 4-Bit-Addierers lässt sich dies anschaulich erläutern: Startet man bei 0 und addiert sukzessive eine 1, durchläuft das Ergebnis die Zahlen von 0 bis $15_{10} = 1111_2$. Bei der Addition von 15_{10} und 1 erreicht man wieder den Ausgangspunkt: Das vom Addierer ausgegebene Ergebnis ist 0000_2, da die Zahl $16_{10} = 10000_2$ nicht mit 4 Bit dargestellt werden kann.

Die grafische Darstellung dieses Verhaltens kann also kein Zahlenstrahl sein. Vielmehr ergibt sich ein *Zahlenkreis,* der bei Addition im Uhrzeigersinn durchlaufen wird. Entsprechend wird der Kreis bei der Subtraktion entgegen dem Uhrzeigersinn durchlaufen (Abb. 2.1).

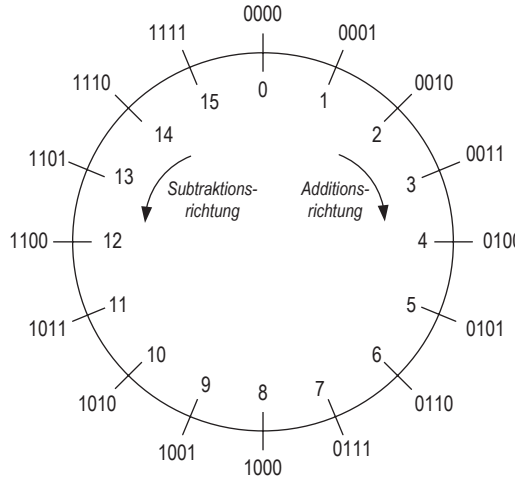


Abb. 2.1 Zahlenkreis für positive Zahlen mit einer Wortbreite von 4 bit

2.2.7 Binäre vorzeichenlose Addition

Die Regeln zur Addition und Subtraktion im Dualsystem sind mit denen des Dezimalsystems vergleichbar. Beide Operationen werden stellenweise, beginnend mit der niederwertigsten Stelle (die Stelle mit dem niedrigsten Stellengewicht), durchgeführt. Bei dieser Operation kann wie im Dezimalsystem ein *Überlauf* auftreten, welcher entsprechend zu berücksichtigen ist. Der wesentliche Unterschied zwischen dem Dezimal- und dem Dualsystem ist, dass der 10er-Übergang des Dezimalsystems einem 2er-Übergang im Dualsystem entspricht. Für die Addition zweier Dualzahlen bedeutet dies, dass ein Übertrag in der nächsthöheren Stelle zu berücksichtigen ist, wenn die Summe der Ziffern den Wert 1 überschreitet. Es ergeben sich 8 mögliche Fälle für die einstellige binäre Addition, welche in Tab. 2.6 zusammengefasst sind.

Zur Verdeutlichung ein Beispiel: Die beiden binären Zahlen 0011 und 1001 sollen addiert werden. Die Addition der beiden niederwertigsten Stellen ergibt den Wert 2. Dieses Ergebnis wird durch eine 1 in der nächsthöheren Stelle (Übertrag) und eine 0 in der aktuellen Stelle dargestellt (vgl. Abb. 2.2). Unter Berücksichtigung des Übertrags und der zwei Operandenbits der nächsthöheren Stelle ergibt sich wiederum ein Übertrag 1 und ein Ergebnisbit mit dem Wert 0. Dieses Verfahren wird für alle Operandenbits durchgeführt und man erhält ein Ergebnis mit der Wortbreite 4 bit.

Überlaufsdetektion bei der vorzeichenlosen Addition
Variante 1: Betrachtung des höchstwertigen Übertragsbits

Ist das höchstwertige Übertragsbit bei der Addition zweier vorzeichenloser Zahlen 0, ist das Ergebnis korrekt. Andernfalls ist bei der Addition ein Überlauf aufgetreten und das ausgegebene Ergebnis nicht korrekt.

Tab. 2.6 Übersicht über die einstellige binäre Addition

Eingabewerte			Ausgabewerte	
1. Summand	2. Summand	Übertragsbit	Summenbit	Übertragsbit
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Abb. 2.2 Beispiel für die binäre Addition

```
          0011
      +   1001
Übertrag: 0011
          1100
```

Variante 2: Betrachtung der höchstwertigen Bits der Operanden und des Ergebnisses

Sind beide höchstwertigen Bits der Operanden identisch, tritt bei der Addition ein Überlauf auf, wenn diese Bits gleich 1 sind. Sind die beiden höchstwertigen Bits der Operanden unterschiedlich, ist ein Überlauf aufgetreten, wenn das höchstwertige Ergebnisbit gleich 0 ist. In allen anderen Fällen ist kein Überlauf aufgetreten.

2.2.8 Binäre vorzeichenlose Subtraktion

Bei der binären Subtraktion können ähnliche Rechenregeln angewandt werden wie sie aus dem Dezimalsystem bekannt sind. Sukzessive werden die einzelnen Bits des Minuenden und Subtrahenden beginnend mit dem niederwertigsten Bit betrachtet. Es wird die Differenz aus dem Minuendenbit und dem Subtrahendenbit bestimmt. Sofern ein Übertrag zu berücksichtigen ist, wird dieser mit negativem Vorzeichen einbezogen. Es ergeben sich wie bei der Addition 8 mögliche Fälle (Tab. 2.7)

Soll beispielsweise die binäre Zahl 0111 von der Zahl 1100 subtrahiert werden, ergibt sich die in Abb. 2.3 dargestellte Rechnung. Die Subtraktion der beiden niederwertigsten Stellen ergibt den Wert -1. Dieses Ergebnis wird durch einen (negativ bewerteten) Übertrag mit dem Wert -1 in der nächsthöheren Stelle und einem Ergebnisbit mit dem Wert 1 in der aktuellen Stelle dargestellt. Unter Berücksichtigung des Übertrags und der zwei Operandenbits der nächsthöheren Stelle ergibt sich ein Übertrag -1 und ein Ergebnisbit mit dem Wert 0. Dieses Verfahren wird für alle Bits der Operanden durchgeführt und so die Differenz mit der Wortbreite 4 bit bestimmt.

Tab. 2.7 Übersicht über die einstellige binäre Subtraktion

Eingabewerte			Ausgabewerte	
Minuend	Subtrahend	Übertrag	Differenz	Übertrag
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

Abb. 2.3 Beispiel für die binäre Subtraktion

```
          1100
    -     0111
Übertrag: 0111
          0101
```

Wie bei der Addition kann im Anschluss an die Berechnung überprüft werden, ob das ausgegebene Ergebnis korrekt ist. Bei der Subtraktion vorzeichenloser Zahlen entsteht ein *Unterlauf,* wenn der Minuend kleiner als der Subtrahend ist. In diesem Fall ist das wahre Ergebnis negativ und lässt sich nicht als vorzeichenlose Zahl darstellen. Für die Detektion eines Unterlaufs können wieder zwei alternative Möglichkeiten eingesetzt werden:

Unterlaufsdetektion bei der vorzeichenlosen Subtraktion
Variante 1: Betrachtung des höchstwertigen Übertragsbits

Ist das höchstwertige Übertragsbit bei der Subtraktion zweier natürlicher Zahlen 0, ist das Ergebnis korrekt. Andernfalls ist ein Unterlauf aufgetreten und das ausgegebene Ergebnis nicht korrekt.

Möglichkeit 2: Betrachtung der höchstwertigen Bits der Operanden und des Ergebnisses

Sind beide höchstwertigen Bits der Operanden identisch, tritt bei der Addition ein Unterlauf auf, wenn das höchstwertige Ergebnisbit gleich 1 ist. Ebenfalls tritt ein Unterlauf auf, wenn das höchstwertige Bit des Minuenden 0 und das des Subtrahenden 1 ist. In allen anderen Fällen tritt kein Unterlauf auf und das Ergebnis ist korrekt.

2.2.9 Binäre vorzeichenlose Multiplikation und Division

Für die Addition und Subtraktion im Binärsystem gelten vergleichbare Regeln wie im Dezimalsystem. Es ist lediglich zu beachten, dass der 10er-Übergang des Dezimalsystems einem 2er-Übergang im Binärsystem entspricht. Unter Beachtung dieser Besonderheit lassen sich auch Vorgehensweisen zur Durchführung der binären Multiplikation oder Division formulieren, die weitgehend den bekannten Regeln des Dezimalsystems entsprechen.

Für die Durchführung der Multiplikation wird der Multiplikator sukzessive mit den einzelnen Stellen des Multiplikanden multipliziert. Da die Ziffern des Multiplikanden nur die Werte 0 oder 1 annehmen können, ist das Ergebnis dieser stellenweisen Multiplikation also entweder Null oder identisch mit dem Multiplikator.

Schreibt man die einzelnen Produkte entsprechend dem Stellengewicht des verwendeten Multiplikandenbits untereinander und summiert anschließend die gebildeten Produkte erhält man als Ergebnis das Produkt der beiden Operanden.

In vielen Fällen möchte man mögliche Überläufe bei der Multiplikation vermeiden und wählt für die Produktwortbreite einen Wert, der sich aus der Summe der Wortbreiten des Multiplikanden und des Multiplikators ergibt.

Die binäre Multiplikation ist für die Zahlen 0101 und 1011 in Abb. 2.4 dargestellt.

Ebenso kann die Division der Grundschulmathematik auf die binäre Division übertragen werden. Hierbei wird der Divisor testweise von einem Teil des Dividenden subtrahiert. Tritt bei der Subtraktion kein Überlauf auf, ergibt sich ein Quotientenbit mit dem Wert 1 und das Ergebnis der Subtraktion wird für weitere Berechnungen weiterverwendet. Ist dagegen ein Überlauf aufgetreten, ist das berechnete Quotientenbit 0 und das Ergebnis der Subtraktion wird verworfen. Es wird mit dem Minuenden weiter gerechnet. Vor der nachfolgenden Subtraktion zur Bestimmung eines weiteren Quotientenbits wird ein weiteres Bit des Dividenden an die berechnete Differenz (kein Überlauf) bzw. den Minuenden (bei aufgetretenem Überlauf) angefügt. Auf diese Weise wird sukzessive der gesamte Dividend durchlaufen. Das Ergebnis der letzten Subtraktion ergibt den Rest der Division. Es ist zu beachten, dass die führenden Nullen des Divisors nicht berücksichtigt werden.

Die Vorgehensweise für eine binäre Addition wird in für einen Dividenden mit dem Wert 01010101 und einem Divisor mit dem 1011 verdeutlicht (Abb. 2.5).

Abb. 2.4 Beispiel für die binäre Multiplikation

```
    0101 * 1011
-------------------
+    0101
+   0101
+  0000
+ 0101

  0001000
-------------------
  00110111
```

```
   01010101 : 1011 = 00111
-  1011:::::   Unterlauf, Quotientenbit = 0
    1010:::::  Minuendenbits verwenden
-   1011:::::  Unterlauf, Quotientenbit = 0
    10101:::::  Minuendenbits verwenden
-    1011:::::  kein Unterlauf, Quotientenbit = 1
     10100:::::  Differenz verwenden (10101-1011=1010)
-     1011:::::  kein Unterlauf, Quotientenbit = 1
      10011::::  Differenz verwenden (10100-1011=1001)
-      1011::::  kein Unterlauf, Quotientenbit = 1
       1000::::  Differenz ergibt den Rest der Division (10011-1011=1000)
```

Abb. 2.5 Beispiel für die binäre Division

Die vorgestellten Rechenvorschriften können als Basis für die Implementierung digitaler Arithmetikschaltungen verwendet werden. In der Praxis kommen teilweise auch modifizierte Verfahren zum Einsatz, die Vorteile im Hinblick auf die Rechenzeit oder den Schaltungsaufwand bieten. Die Schaltungsstruktur eines Addierers wird in Kapitel 6 vorgestellt.

2.3 Vorzeichenbehaftete Zahlen

In vielen Fällen ist die ausschließliche Verwendung vorzeichenloser Zahlen nicht ausreichend und es müssen sowohl positive als auch negative Zahlen verwendet werden. Hieraus ergibt sich zwangsläufig die Frage nach einer geeigneten Codierung vorzeichenbehafteter Zahlen.

Eine naheliegende Idee wäre es, die Zahlendarstellung des täglichen Lebens auch auf Dualzahlen anzuwenden. Üblicherweise kennzeichnen wir Zahlenwerte mit einem vorangestellten Vorzeichen, einem Plus- oder Minuszeichen. Der Zahlenwert nach dem Vorzeichen entspricht dem Betrag der Zahl. Diese Form der Zahlendarstellung wird als *Vorzeichen-Betrag-Darstellung* bezeichnet. Die am weitesten verbreitete Darstellungsform vorzeichenbehafteter Zahlen ist die sogenannte *Zweierkomplement-Darstellung,* die in Abschn. 2.3.2 vorgestellt wird.

2.3.1 Vorzeichen-Betrag-Darstellung

In der üblichen Dezimaldarstellung werden vorzeichenbehaftete Zahlenwerte als eine Kombination von Vorzeichen und Betrag dargestellt. Es handelt sich um die Vorzeichen-Betrag-Darstellung. Dieses Prinzip lässt sich auch auf Dualzahlen übertragen. Es bietet sich an, das Vorzeichen durch ein einzelnes Bit zu codieren. Üblicherweise verwendet man eine führende 0 um einen positiven Zahlenwert darzustellen und eine führende 1 für

negative Zahlen. Die restlichen Bits entsprechen dem Betrag der dargestellten Zahl, welcher als vorzeichenlose Dualzahl codiert ist.

Genauso wie für vorzeichenlose Zahlen kann als grafische Darstellung ein Zahlenkreis verwendet werden. Abb. 2.6 zeigt den Zahlenkreis für eine Wortbreite von 4 bit.

Betrachtet man den Zahlenkreis in Abb. 2.6 genauer, fallen mehrere Besonderheiten auf:

1. Es existieren zwei Repräsentationen der Null, „+0“ und „−0“.
2. Es gibt zwei Stellen, an denen Überläufe bzw. Unterläufe auftreten können, nämlich zwischen -7 und $+0$ sowie zwischen $+7$ und -0
3. Die Additionsrichtung im Bereich positiver Zahlen entspricht der Subtraktionsrichtung im Bereich negativer Zahlen.

Alle drei Beobachtungen sind Nachteile, die das Rechnen in dieser Zahlendarstellung erschweren bzw. die Implementierung arithmetischer Schaltungen aufwendiger machen.

Um beispielsweise eine Addition durchzuführen, können verschiedene Vorgehensweisen definiert werden. Am einfachsten ist es, wenn das Vorzeichen der Operanden für die eigentliche arithmetische Operation unberücksichtigt bleibt und eine Operation wie für vorzeichenlose Zahlen durchgeführt wird. Um dabei das korrekte Ergebnis zu erhalten, ist eine Fallunterscheidung auf Basis der Vorzeichen der Operanden erforderlich. Je nach vorliegendem Fall, wird gegebenenfalls eine Vertauschung der Operanden vorgenommen, statt einer Addition eine Subtraktion durchgeführt oder das Vorzeichen des Ergebnisses invertiert (Tab. 2.8).

Äquivalent zur Addition können auch für andere Grundoperationen Rechenregeln formuliert werden, wobei eine geeignete Fallunterscheidung vorzusehen ist. Dies stellt

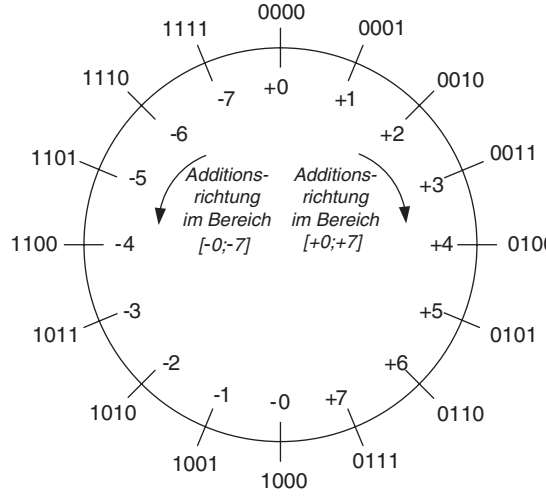


Abb. 2.6 Zahlenkreis für vorzeichenbehaftete Zahlen in Vorzeichen-Betrag-Darstellung

Tab. 2.8 Fallunterscheidung für die Addition in Vorzeichen-Betrag-Darstellung

Vorzeichenbit der Operanden		Erforderliche Schritte		
1. Summand	2. Summand	Operanden vertauschen	Ausgeführte Operation	Vorzeichen des Ergebnisses invertieren
0	0	nein	Addition	nein
0	1	nein	Subtraktion	nein
1	0	ja	Subtraktion	nein
1	1	nein	Addition	ja

einen wesentlichen Nachteil für den Einsatz der Vorzeichen-Betrag-Darstellung in digitalen Systemen dar, da die Fallunterscheidungen in Hardware implementiert werden müssten, wodurch sich der schaltungstechnische Aufwand erhöht.

2.3.2 Zweierkomplement-Darstellung

Aus den Überlegungen des vorangegangenen Abschnitts lassen sich Forderungen formulieren, die für eine Darstellung vorzeichenbehafteter Zahlen gelten sollten. So ist es wünschenswert, dass

1. nur eine Codierung dem Zahlenwert Null entspricht,
2. die Additionsrichtung für den gesamten Zahlenbereich identisch ist,
3. nur an einer Position im Zahlenkreis ein Überlauf bzw. Unterlauf auftritt.

Eine Zahlendarstellung die diese Forderungen erfüllt, ist die sogenannte *Zweierkomplement-Darstellung*. Die Codierung der Zahlen im Zweierkomplement ergibt sich aus den ersten beiden Forderungen: Zwischen den Zahlenwerten -1 und $+1$ darf nur eine Codierung existieren, die den Wert 0 repräsentiert. Setzt man voraus, dass die positiven Zahlen wie bei der Vorzeichen-Betrag-Darstellung durch eine führende 0 zu identifizieren sind und legt zugrunde, dass die selbstverständliche Gleichung $1-2=-1$ gelten soll, lässt sich die Codierung der Zahl -1 wie folgt anhand des Zahlenkreises bestimmen: Als Startpunkt wählt man auf dem Zahlenkreis die Codierung „0001“, was der Zahl $+1$ entspricht. Läuft man auf dem Zahlenkreis einen Schritt in Subtraktionsrichtung, muss sich die Codierung der Zahl 0 ergeben. Diese entspricht bei einer Wortbreite von 4 bit der Codierung 0000 und entspricht somit der Darstellung der Null für vorzeichenlose Zahlen. Ein weiterer Schritt in Subtraktionsrichtung muss zwangsläufig zur Codierung der Zahl -1 führen. Für eine Wortbreite von 4 bit ergibt sich für -1 also die Codierung 1111. Die Codierungen aller weiteren negativen Zahlen können durch weitere Schritte in Subtraktionsrichtung gefunden werden.

Für die Zweierkomplement-Darstellung gilt, dass alle Codierungen mit einer führenden 1 als negative Zahlen zu interpretieren sind. Dies hat den Vorteil, dass sich der Wert einer Zweierkomplement-Zahl durch eine einfache Summenformel angeben lässt. Als einziger Unterschied zu der Formel für vorzeichenlose Zahlen ist bei Zweierkomplement-Zahlen beim höchstwertigen Bit ein negatives Stellengewicht zu berücksichtigen:

$$Z = -z_{N-1} \cdot 2^{N-1} + \sum_{i=0}^{N-2} z_i \cdot 2^i$$

So ergibt sich für eine Wortbreite von 4 bit die Zahl -8 als kleinste darstellbare negative Zahl, welche durch die Bitfolge 1000 codiert wird. Der Zahlenkreis für Zweierkomplement-Zahlen mit einer Wortbreite von 4 bit ist in Abb. 2.7 dargestellt.

2.3.2.1 Negieren einer Zweierkomplement-Zahl

Möchte man eine vorzeichenbehaftete Zahl in Zweierkomplement-Darstellung negieren, kann man die vorgestellte Summenformel verwenden um den Wert der Ausgangszahl zu bestimmen. Anschließend wird das Vorzeichen der Zahl invertiert und wiederum mithilfe der Summenformeln die Codierung der gesuchten Zahl bestimmt. Dieses Vorgehen ist jedoch relativ umständlich und fehlerträchtig.

Aufgrund der Eigenschaften der Zweierkomplement-Zahlen lässt sich glücklicherweise ein einfacheres zweischrittiges Verfahren definieren: Zunächst werden alle Stellen der Ausgangszahl invertiert. Anschließend wird dieses Zwischenergebnis inkrementiert (= eine 1 addiert). Das Ergebnis stellt die entsprechende negierte Zahl dar.

Hierzu ein Beispiel: Die 6 bit breite Zweierkomplement-Zahl „011101“ soll negiert werden.

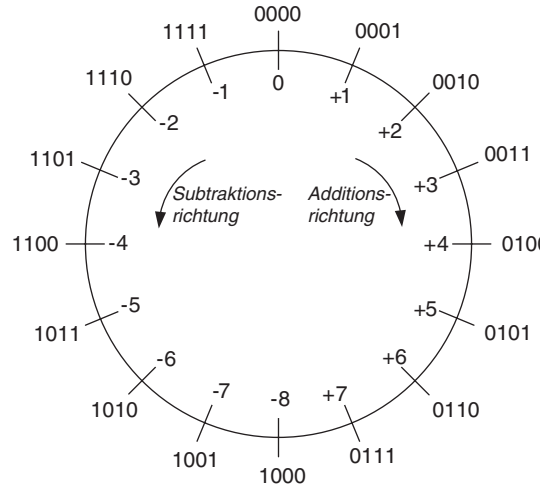


Abb. 2.7 Zahlenkreis für vorzeichenbehaftete Zahlen in Zweierkomplement-Darstellung

$$NEG(011101) = \overline{011101} + 1 = 100010 + 1 = 100011$$

Mithilfe der Summenformel für Zweierkomplement-Zahlen kann das Ergebnis überprüft werden:

$$011101_2 = 16 + 8 + 4 + 1 = 29$$
$$100011_2 = -32 + 2 + 1 \quad = -29$$

2.3.2.2 Vorzeichenerweiterung

In einigen praktischen Anwendungsfällen ist es erforderlich die Wortbreite einer Zahl zu vergrößern und zum Beispiel aus einer 8 bit breiten Zahl eine Zahl mit der Wortbreite 16 bit zu generieren. Für vorzeichenlose Zahlen ist es lediglich erforderlich die Zahl mit führenden Nullen aufzufüllen. Im Fall der Zweierkomplement-Darstellung werden die zusätzlichen Stellen dagegen mit dem höchstwertigen Bit (Vorzeichenbit) der Ausgangszahl aufgefüllt.

2.3.3 Addition und Subtraktion in Zweierkomplement-Darstellung

Für die Bestimmung der Ergebnisbits einer Addition oder Subtraktion von Zahlen in Zweierkomplement-Darstellung gilt das gleiche Vorgehen wie für vorzeichenlose Zahlen. Dies bedeutet unter anderem, dass eine Additions- bzw. Subtraktionsschaltung für vorzeichenlose Zahlen unverändert auch für Zweierkomplement-Zahlen eingesetzt werden kann. Dieses ist insbesondere dann vorteilhaft, wenn in einem digitalen System sowohl vorzeichenlose als auch vorzeichenbehaftete Zahlen verarbeitet werden, wie dies zum Beispiel in digitalen Rechnern der Fall ist.

Für die Bestimmung von Überläufen und Unterläufen bei der Zweierkomplement-Addition bzw. -Subtraktion gelten andere Regeln als bei vorzeichenlosen Zahlen. Eine Überschreitung des darstellbaren Zahlenbereichs kann ebenfalls durch die Betrachtung der höchstwertigen Bits der Operanden und des Ergebnisses detektiert werden. Für die Addition gilt beispielsweise, dass nur dann ein Überlauf oder Unterlauf auftreten kann, wenn beide Summanden das gleiche Vorzeichen besitzen. Besitzen beispielsweise beide Operanden ein positives Vorzeichen (repräsentiert durch eine führende Null), so muss auch die Summe ein positives Vorzeichen besitzen. Besitzt das Ergebnis dagegen eine führende Eins und repräsentiert somit einen negativen Zahlenwert, ist dieses offenbar falsche Ergebnis auf einen Überlauf zurückzuführen. Entsprechendes gilt für den Fall der Addition zweier negativer Zahlen. Die Überlegungen für die Addition lassen sich entsprechend für die Subtraktion anstellen. Hierbei gilt, dass eine Bereichsüberschreitung nur dann auftritt, wenn die beiden Operanden unterschiedliche Vorzeichen besitzen.

Über-/Unterlaufsdetektion bei der vorzeichenbehafteten Addition

Sind beide höchstwertigen Bits der Operanden identisch und ist das höchstwertige Ergebnisbit ungleich der höchstwertigen Operandenbits, ist ein Überlauf bzw.

Unterlauf aufgetreten. In allen anderen Fällen ist keine Überschreitung des darstellbaren Zahlenbereichs aufgetreten und das Ergebnis ist korrekt codiert.

Über-/Unterlaufsdetektion bei der vorzeichenbehafteten Subtraktion
Sind beide höchstwertigen Bits der Operanden unterschiedlich und ist das höchstwertige Ergebnisbit ungleich dem höchstwertigen Operandenbit des Minuenden, ist ein Überlauf bzw. Unterlauf aufgetreten. In allen anderen Fällen ist keine Überschreitung des darstellbaren Zahlenbereichs aufgetreten und das Ergebnis ist korrekt codiert.

2.3.4 Multiplikation und Division in Zweierkomplement-Darstellung

Für die Multiplikation und die Division von Zweierkomplement-Zahlen bietet sich als einfachste Vorgehensweise ein dreischrittiges Verfahren an. Hierbei werden zunächst die Beträge der Operanden berechnet und anschließend die eigentliche Operation mit vorzeichenlosen Zahlen durchgeführt. Im letzten Schritt wird gegebenenfalls das Ergebnis durch Negierung korrigiert, falls die Operanden unterschiedliche Vorzeichen besitzen. Diese Korrektur muss für das Produkt bei der Multiplikation oder dem Quotienten bei der Division ausgeführt werden. Für die Korrektur des Restes einer binären Zweierkomplement-Division wird lediglich das Vorzeichen des Dividenden berücksichtigt: Ist der Dividend negativ, ist eine Korrektur des Restes durch Negierung vorzunehmen.

Alternativ zu der oben beschriebenen Vorgehensweise kann beispielsweise für Multiplikation eine Vorgehensweise gewählt werden, die berücksichtigt, dass das höchstwertige Bit der Operanden negativ zu gewichten ist. Unter Berücksichtigung dieser Eigenschaft der Zweierkomplement-Zahlen kann die Multiplikation äquivalent zur Multiplikation vorzeichenloser Zahlen ausgeführt werden. Hierbei ergeben sich in den Teilprodukten einzelne negativ zu bewertende Einsen, die bei der Summation der Teilprodukte negativ zu berücksichtigen sind. Das nachfolgende Beispiel verdeutlicht die Vorgehensweise, wobei negativ zu berücksichtigende Bits kursiv dargestellt sind.

Sollen zum Beispiel die beiden vorzeichenbehafteten Zahlen 1101 und 1001 multipliziert werden, ergäbe sich das in Abb. 2.8 dargestellte Vorgehen.

Abb. 2.8 Beispiel für die Zweierkomplement-Multiplikation

```
   1101 * 1001
-------------
+    1101
+   0000
+  0000
+ 1101

  01110000
-------------
  00010101
```

2.3.5 Bias-Darstellung

Eine weitere Möglichkeit vorzeichenbehaftete Zahlen darzustellen, ist die sogenannte *Bias-Darstellung* (bzw. Excess-Darstellung). Der Begriff „Bias“ stammt aus dem Englischen und bedeutet in etwa „Vorbeaufschlagung“ oder „Vorspannung“. Bei dieser Darstellung kann der Zahlenwert mithilfe der Summenformel für vorzeichenlose Zahlen bestimmt werden, wobei nach der Summenbildung eine Konstante B subtrahiert wird. Durch die Subtraktion der Konstanten können auch negative Zahlenwerte dargestellt werden. Der Wert der Konstanten kann im Prinzip beliebig gewählt werden. Da man in der Regel einen symmetrischen Zahlenbereich anstrebt (Absolutwert der kleinsten negativen Zahl entspricht etwa dem Wert der größten positiven Zahl), wird B im Allgemeinen entsprechend der Wortbreite N der Zahlendarstellung gewählt:

$$B = \frac{2^N}{2} - 1 = 2^{N-1} - 1$$

Betrachten wir die Bitfolge 100101, welche eine Zahl in Bias-Darstellung repräsentiert. Welcher Zahlenwert wird durch die Bitfolge dargestellt?

Mit $N = 6$ ergibt sich

$$Z = \sum_{i=0}^{\mathrm{N}-1} z_i \cdot 2^i - B = \left(2^5 + 2^2 + 2^0\right) - \left(2^5 - 1\right) = 6$$

2.3.6 Darstellbare Zahlenbereiche

Häufig ergibt sich beim Entwurf eines digitalen Systems die Frage, welche Wortbreite für die Darstellung von Zahlenwerten verwendet werden muss. Um Aufwand zu sparen möchte man natürlich so wenige Bits wie möglich verwenden. Andererseits muss die Wortbreite aber ausreichend sein, um den gewünschten Zahlenbereich abzudecken. Tab. 2.9 fast den darstellbaren Zahlenbereich für Zahlen mit einer Wortbreite von N bit zusammen:

Tab. 2.9 Darstellbarer Zahlenbereich in Abhängigkeit der verwendeten Wortbreite N bit

Zahlendarstellung	Kleinster Wert	Größter Wert
Vorzeichenlos	0	$2^N - 1$
Vorzeichen-Betrag	$-2^{N-1} + 1$	$2^{N-1} - 1$
Zweierkomplement	-2^{N-1}	$2^{N-1} - 1$
Bias ($B = 2^{N-1} - 1$)	$-2^{N-1} + 1$	2^{N-1}

2.4 Reelle Zahlen

In den vorangegangenen Abschnitten wurde die binäre Darstellung ganzer Zahlen betrachtet. Viele Problemstellungen der Digitaltechnik lassen sich mit ausreichender Genauigkeit mithilfe ganzer Zahlen lösen. Es gibt aber auch Anwendungen, die den Einsatz reeller Zahlen erfordern. Im Folgenden wird daher eine Übersicht über die Möglichkeiten zur binären Darstellung reeller Zahlen gegeben, wobei die Varianten *Festkomma-Darstellung* und *Gleitkomma-Darstellung* unterschieden werden.

2.4.1 Festkomma-Darstellung

Für die Darstellung von ganzen Zahlen wurde die Vereinbarung getroffen, dass das niederwertigste Bit die Einerstelle darstellt, also mit 2^0 gewichtet wird. Diese Vereinbarung ist zwar für ganze Zahlen sinnvoll, aber letztlich willkürlich. Genauso gut kann als Stellengewicht des niederwertigsten Bits einer binären Zahl auch eine Zweierpotenz mit negativem Exponenten gewählt werden. Um den Wert einer solchen Zahl zu bestimmen, muss die Summenformel für ganze Zahlen geringfügig modifiziert werden und lautet nun

$$Z = \sum_{i=-\mathrm{L}}^{\mathrm{M}-1} z_i \cdot 2^i$$

für vorzeichenlose Zahlen bzw.

$$Z = -\mathrm{z}_{\mathrm{M}-1} \cdot 2^{M-1} + \sum_{i=-\mathrm{L}}^{\mathrm{M}-2} z_i \cdot 2^i$$

für vorzeichenbehaftete Zahlen.

Die benötigte Wortbreite N einer derartigen Zahl ergibt sich aus der Summe der Anzahl der Vorkommastellen M und der Nachkommastellen L:

$$N = M + L$$

Vereinbart man beispielsweise, dass zwei Nachkommastellen ($L = 2$) verwendet werden. Welchem Zahlenwert würde dann die binäre Ziffernfolge „10111“ als vorzeichenlose Zahl entsprechen? Welcher Zahlenwert ergibt sich als vorzeichenbehaftete Zahl?

Mithilfe der obigen Summenformeln ist die Lösung leicht zu bestimmen. Werden die Bits als vorzeichenlose Zahl interpretiert ergibt sich

$$Z_{\text{vorzeichenlos}} = 2^2 + 2^0 + 2^{-1} + 2^{-2} = 5,75$$

Wenn die Bits eine vorzeichenbehaftete Zahl in Festkommadarstellung repräsentieren ergibt sich der dargestellte Zahlenwert zu

$$Z_{\text{zweierkomplement}} = -2^2 + 2^0 + 2^{-1} + 2^{-2} = -2,25$$

Für die arithmetischen Grundoperationen ergeben sich keine bzw. lediglich geringe Änderungen. Besitzen beide Operanden die gleiche Anzahl an Nachkommastellen *L,* kann die Addition und Subtraktion genauso wie für ganze Zahlen durchgeführt werden. Das Ergebnis besitzt ebenfalls *L* Nachkommastellen. Bei der Multiplikation besitzt das Ergebnis dagegen 2 · L Nachkommastellen. Um bei der Division die gewünschte Genauigkeit des Quotienten zu erhalten, können die Nachkommastellen des Dividenden vor Ausführung der Division mit Nullen erweitert werden.

Müssen dagegen Zahlen mit unterschiedlichen Wortbreiten verarbeitet werden, sind beispielsweise bei der Addition und Subtraktion Korrekturschritte erforderlich um die Stellengewichte der einzelnen Bits anzupassen.

Nehmen wir an, die Zahl 01001 mit zwei Nachkommastellen und die Zahl 10110 mit drei Nachkommastellen sollen addiert werden. Das niederwertigste Bit der ersten Zahl besitzt das Gewicht 2^{-2} und das der zweiten Zahl 2^{-3}. Diese beiden Bits dürfen also nicht einfach addiert werden, da die bekannten Regeln zur binären Addition darauf basieren, dass immer Bits mit gleichem Stellengewicht betrachtet werden. Also müssen die Zahlen zunächst so erweitert werden, dass die Stellengewichte der einzelnen Bits übereinstimmen: Die erste Zahl wird rechts um eine Stelle mit dem Wert 0 erweitert, während bei der zweiten Zahl auf der linken Seite eine 0 angefügt wird (in Abb. 2.9 kursiv dargestellt). Anschließend kann die Addition wie gewohnt ausgeführt werden. Sofern erforderlich, kann die Wortbreite des Ergebnisses durch Weglassen der niederwertigsten Nachkommastelle anschließend wieder auf 5 reduziert werden.

2.4.2 Gleitkomma-Darstellung

Insbesondere in digitalen Rechnersystemen, hat sich die Gleitkomma-Darstellung, wie sie in der internationalen Norm IEEE 754 definiert ist, durchgesetzt. Solche Rechnersysteme sollen sowohl kleine als auch große Datenwerte verarbeiten können und genau dies ermöglicht die Gleitkomma-Darstellung. Eine detaillierte Beschreibung dieser Zahlendarstellung würde den Rahmen dieses Buches sprengen. Daher wird im Folgenden lediglich das Grundprinzip der Gleitkomma-Darstellung betrachtet.

Bei Verwendung dieser Gleitkomma-Darstellung wird der Zahlenwert durch eine Mantisse *M* und einen Exponenten *E* dargestellt. Sowohl *M* als auch *E* werden hierbei als ganze Zahlen codiert, wobei für *M* die Vorzeichen-Betrag-Darstellung und für *E* die

Abb. 2.9 Beispiel für die Festkomma-Addition

```
  01001 0
+ 0 10110
-------
  101000
```

Bias-Darstellung gewählt wird. Zusätzlich wird ein Vorzeichenbit S angeben. Der Zahlenwert Z_{GK} einer Gleitkommazahl kann wie folgt bestimmt werden:

$$Z_{GK} = (-1)^S \cdot M \cdot 2^E$$

Die verwendeten Wortbreiten für die Mantisse und den Exponenten sind der Norm IEEE 754 festgelegt, die unterschiedliche Genauigkeiten spezifiziert. Für die einfache Genauigkeit (C-Datentyp *float*) werden insgesamt 32 Bit verwendet, die sich in 24 Bit für die Mantisse inklusive Vorzeichenbit und 8 Bit für den Exponenten aufteilen. Für die sogenannte doppelte Genauigkeit (C-Datentyp *double*) werden die Mantisse mit 53 Bit und der Exponent mit 11 Bit codiert.

2.4.3 Reelle Zahlen in digitalen Systemen

In der Praxis steht man häufig vor der Problemstellung einen Algorithmus entwerfen zu müssen, welcher im Anschluss in einem digitalen System in Software oder Hardware implementiert werden soll. Für die Entwicklung eines Algorithmus mag es bequem erscheinen, wenn man sich über die Wortbreiten der verwendeten Zahlen möglichst wenig Gedanken machen muss. Also ist es naheliegend alle Berechnungen mit einer möglichst flexiblen Zahlendarstellung, wie zum Beispiel einer Gleitkomma-Darstellung mit doppelter Genauigkeit, durchzuführen. Soll der Algorithmus später in Form einer digitalen Hardware realisiert werden, wird man allerdings auf Probleme stoßen, da die Hardware-Umsetzung von Berechnungen in Gleitkomma-Darstellung relativ aufwendig ist. Kann dieser Aufwand, zum Beispiel aus Kostengründen, nicht betrieben werden, müssen die algorithmischen Vorgaben in Gleitkomma-Darstellung in eine weniger komplexe ganzzahlige Darstellung umgewandelt werden. Hierbei werden möglicherweise wichtige Eigenschaften des entwickelten Algorithmus verändert, sodass nicht ohne Weiteres gewährleistet werden kann, dass das finale Produkt den ursprünglich ins Auge gefassten Qualitätsvorgaben entspricht.

In der Praxis werden daher frühzeitig die erforderlichen Wortbreiten ermittelt. Auf den Einsatz einer Gleitkomma-Darstellung wird verzichtet. Dies gilt insbesondere dann, wenn ein Algorithmus in digitale Hardware überführt oder in Software auf einem preisgünstigen Rechnersystem, wie zum Beispiel einem einfachen Mikrocontroller, ausgeführt werden soll.

2.5 Codes

In diesem Abschnitt werden gebräuchliche Möglichkeiten vorgestellt, um Informationen in digitaler Form darzustellen. Diese Informationen müssen nicht zwangsläufig Zahlenwerte repräsentieren. Einer Bitfolge können auch beliebige andere Bedeutungen zugeordnet werden. So kann man mit Codes zum Beispiel Farben oder auch die Fehlerzustände einer Maschine darstellen.

2.5.1 BCD-Code

Der BCD-Code *(Binary Coded Digit)* dient der Codierung der zehn Dezimalziffern. Für die Codierung jeder Ziffer werden 4 Bit verwendet, die auch als *Tetraden* bezeichnet werden. Die verwendeten Bitfolgen entsprechen der dualen Darstellung der vorzeichenlosen Zahlen 0 bis 9. Da bei der Verwendung von 4 Bits 16 verschiedene Bitkombinationen möglich sind, jedoch nur 10 hiervon zur Codierung der Ziffern benötigt werden, werden 6 Bitkombinationen nicht verwendet. Diese nicht verwendeten Kombinationen werden als *Pseudotetraden* bezeichnet. In Tab. 2.10 ist die Codierung einer Dezimalziffer in Form einer BCD-Tetrade dargestellt.

Der BCD-Code wird zum Teil in Digitaluhren und für digitale Displays (zum Beispiel in Multimetern) eingesetzt. Der BCD-Code kann auch für die Implementierung von Rechnersystemen eingesetzt werden. Hierbei kann es vorkommen, dass das Ergebnis einer Addition zu einer Pseudotetrade führt. Um ein Ergebnis, das eine Pseudotetrade enthält, wieder in eine gültige BCD-Darstellung umzuwandeln, sind Korrekturschritte erforderlich, die die Implementierung der BCD-Arithmetik komplizieren. Darüber hinaus ist die BCD-Darstellung nicht speichereffizient, da mit einer Tetrade nur 10 statt der sonst 16 möglichen Codierungen verwendet werden. So können beispielsweise mit 8 Bit nur die Zahlen von 0 bis 99 dargestellt werden, während mit der Darstellung als vorzeichenlose Dualzahl der Bereich von 0 bis 255 abgedeckt ist.

Tab. 2.10 Codierung einer Dezimalziffer auf Basis des BCD-Codes

a_3	a_2	a_1	a_0	Codierte Dezimalziffer
0	0	0	0	0
0	0	0	1	1
0	0	1	0	2
0	0	1	1	3
0	1	0	0	4
0	1	0	1	5
0	1	1	0	6
0	1	1	1	7
1	0	0	0	8
1	0	0	1	9
1	0	1	0	Pseudotetraden
1	0	1	1	
1	1	0	0	
1	1	0	1	
1	1	1	0	
1	1	1	1	

Abb. 2.10 Beispiel für die BCD-Addition

```
    0011 0111   (37)
+   0101 0101   (55)
-------------------
    1000 1100   (Pseudo-Tetrade)

+   0000 0110   (Korrekturschritt: +6)
-------------------
    1001 0010   (92)
```

Nehmen wir an, die beiden BCD-Zahlen 37 und 55 sollen addiert werden. Auch das Ergebnis soll in BCD-Darstellung vorliegen. Die Addition kann ohne weitere Beachtung der BCD-Codierung durchgeführt werden. Man erhält dann das Ergebnis in der üblichen binären Darstellung. In diesem Beispiel ergibt sich für die untere Hälfte des Ergebnisses die Pseudotetrade 1100.

Zur Korrektur des Ergebnisses kann zunächst die nächsthöhere BCD-Stelle um 1 erhöht werden, was der binären Addition des Wertes 16 entspricht. Interpretiert man die so erhaltenen Ergebnisbits als BCD-Zahl, wäre das Ergebnis um 10 zu groß. Dies kann korrigiert werden, indem die untere BCD-Stelle um 10 verringert wird.

Das zweischrittige Vorgehen (16 addieren und anschließend 10 subtrahieren) kann natürlich auch in einem Schritt durch die Addition des Wertes 6 realisiert werden.

Die Korrektur muss sukzessive, beginnend mit den niederwertigsten Bits, immer dann durchgeführt werden, wenn der BCD-Stellen eine Pseudotetrade enthält (Abb. 2.10).

2.5.2 Gray-Code

Stellen Sie sich vor, Sie sollen einen Temperaturwarner realisieren, der aus einem digitalen Thermometer und einer Einheit zur Temperaturüberprüfung besteht. Sinkt die Temperatur unter einen bestimmten Wert, soll ein Alarm ausgegeben werden. Die Temperaturüberprüfung fragt die aktuelle Temperatur, die vom Thermometer als Dualzahl übertragen wird, in regelmäßigen Abständen ab und gibt gegebenenfalls einen Alarm aus. Würden Sie das System so realisieren, könnten sporadische Fehlalarme auftreten.

Wie kann das sein? Nehmen wir vereinfachend an, dass das Thermometer die aktuelle Temperatur mit einer Wortbreite von 4 bit ausgibt und Temperaturen zwischen 0 und 15 °C messen kann. Steigt die Temperatur zum Beispiel von 7 °C auf 8 °C, würde das Thermometer zunächst 0111 und anschließend 1000 ausgeben. Alle vier vom Thermometer ausgegebenen Bits müssen sich in diesem Fall ändern. In einem realen System werden die Bitwechsel auf Grund von zeitlichen Toleranzen bei der Messwertausgabe aber nicht exakt gleichzeitig stattfinden. In Abb. 2.11 ist ein möglicher zeitlicher Verlauf der Thermometerausgabe für den Wechsel von 7 °C auf 8 °C dargestellt, wobei ts_0, ts_1, ts_2 und ts_3 die einzelnen Bits des Temperatursignals und *TSdual* die duale Interpretation der Bits repräsentiert.

Es ist zu erkennen, dass zwischen den tatsächlich gültigen Zahlenwerten 7 und 8 auch ungültige Werte, die nicht der wahren Temperatur entsprechen, an die Einheit zur Temperaturüberprüfung gesendet werden. Wird die Temperatur in einem Moment abgefragt, in dem ein ungültiger Wert ausgegeben wird, kann dies zu einem Fehlalarm führen.

Möglicherweise werden Sie einwenden, dass diese ungültigen Werte nur für sehr kurze Zeiten auftreten und in den meisten Fällen ein korrekter Wert ausgegeben wird. Obwohl dies sicher richtig ist, verschlimmert diese Tatsache die Lage eher noch: Da das System nur selten Fehlalarme ausgeben würde, gestaltet sich eine systematische Fehlersuche extrem schwierig.

Das Kernproblem der oben beschriebenen Temperaturüberwachung liegt darin, dass bei einer Änderung der Temperatur mehrere Bits invertiert werden müssen. Wäre es da nicht eine einfache Lösung des Problems, wenn bei einer Temperaturänderung nur ein einzelnes Bit zu modifizieren wäre? Genau dieser Ansatz wird vom *Gray-Code,* der nach seinem Erfinder Frank Gray benannt ist, aufgegriffen. Der Gray-Code zeichnet sich dadurch aus, dass sich zwei benachbarte Codierungen nur in einer Stelle unterscheiden. Der Gray-Code für eine Wortbreite von 4 bit ist in Tab. 2.11 dargestellt.

Wird der Gray-Code für das Beispiel der Temperaturüberwachung eingesetzt, käme es zu keiner unbeabsichtigten Ausgabe ungültiger Werte und Fehlalarme würden vermieden. Der zeitliche Verlauf des Temperatursignals ist für den Wechsel von 7 °C nach 8 °C in Abb. 2.12 dargestellt.

Der Gray-Code kann immer dann sinnvoll eingesetzt werden, wenn zwischen zwei digitalen Komponenten Werte übertragen werden sollen, deren Änderung stetig ist. So wird der Gray-Code unter anderem auch für die Positions- oder Winkelbestimmung eingesetzt. Ein weiteres Einsatzgebiet ist die Übertragung von Speicherfüllständen innerhalb digitaler Systeme. Für die Implementierung arithmetischer Operationen ist der Gray-Code dagegen nicht gut geeignet.

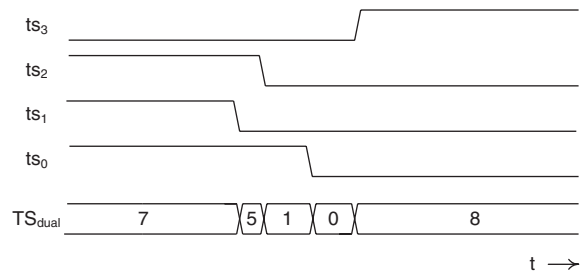


Abb. 2.11 Beispiel des zeitlichen Verlaufs der Ausgabe eines digitalen Thermometers mit dualer Codierung

Tab. 2.11 Gray-Code für eine Wortbreite von 4 bit

Codierter Wert	a_3	a_2	a_1	a_0
0	0	0	0	0
1	0	0	0	1
2	0	0	1	1
3	0	0	1	0
4	0	1	1	0
5	0	1	1	1
6	0	1	0	1
7	0	1	0	0
8	1	1	0	0
9	1	1	0	1
10	1	1	1	1
11	1	1	1	0
12	1	0	1	0
13	1	0	1	1
14	1	0	0	1
15	1	0	0	0

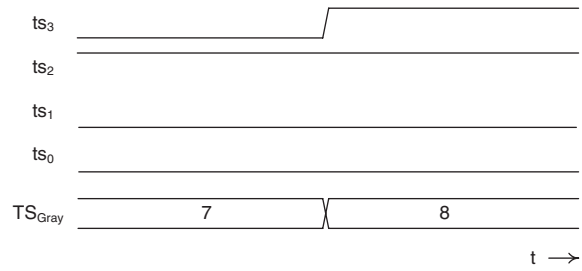


Abb. 2.12 Beispiel des zeitlichen Verlaufs der Ausgabe eines digitalen Thermometers mit Gray-Codierung

2.5.3 1-aus-N-Code

Der *1-aus-N-Code* stellt eine weitere Alternative zur binären Codierung von Informationen dar. Dieser Code zeichnet sich dadurch aus, dass in jedem Codewort mit der Wortbreite N bit nur ein einzelnes Bit auf 1 gesetzt ist; alle anderen Bits besitzen den Wert 0.

Der 1-aus-N-Code ist ein sogenannter redundanter Code, da sich mit N Bits 2^N unterschiedliche binäre Wörter darstellen lassen, von denen jedoch nur N als gültige Codewörter genutzt werden. Der Code geht also verschwenderisch mit der Wortbreite um. Dies wird durch den Vorteil aufgewogen, dass sich die Codewörter relativ leicht codieren bzw. decodieren lassen.

Eine mögliche Codierung der Zahlenwerte 0 bis 5 mit einem 1-aus-6-Code ist exemplarisch in Tab. 2.12 dargestellt.

Tab. 2.12 Codierung der Werte 0 bis 5 mithilfe eines 1-aus-6-Codes

Codierter Wert	a_5	a_4	a_3	a_2	a_1	a_0
0	0	0	0	0	0	***1***
1	0	0	0	0	***1***	0
2	0	0	0	***1***	0	0
3	0	0	***1***	0	0	0
4	0	***1***	0	0	0	0
5	***1***	0	0	0	0	0

2.5.4 ASCII-Code

Mit dem *ASCII-Code (American Standard Code for Information Interchange)* werden ausschließlich Zeichen, also Buchstaben, Ziffern und Sonderzeichen, codiert. Jedes Zeichen wird durch 7 Bit repräsentiert. Der ASCII-Code entspricht nahezu dem 7-Bit-Code nach DIN 66003, welcher im Gegensatz zum ASCII-Code unter anderem auch deutsche Umlaute abdeckt.

Die Zeichencodierung gemäß dem ASCII-Code ist in Tab. 2.13 dargestellt. Die Bits a_4, a_5 und a_6 dienen in dieser Tabelle der Auswahl der Spalten und die Bits a_0, a_1, a_2 und a_3 der Zeilenauswahl. Bei der Übertragung wird für ein ASCII-Zeichen im Allgemeinen ein Byte (8 bit) verwendet. In der Datentechnik wird häufig auch das achte Bit zu einer Erweiterung des Zeichenvorrats herangezogen. Dadurch kann die Anzahl der codierten Zeichen verdoppelt werden.

Da der ASCII-Code nur einen sehr eingeschränkten Zeichensatz von 128 bzw. 256 unterschiedlichen Zeichen bietet, wird in vielen Rechnersystemen auch der sogenannte Unicode zur Codierung von Zeichen eingesetzt. Ziel des Unicodes ist es, alle existierenden Zeichen codieren zu können. Hierzu werden in Unicode Ebenen *(planes)* definiert, die bis zu 65535 Zeichen enthalten können. Der Vorteil, alle gebräuchlichen Zeichen codieren zu können, wird allerdings durch den Nachteil erkauft, dass pro Zeichen eine deutlich höhere Anzahl an Bits vorgesehen werden muss. Daher wird in einfachen Anwendungsfällen (zum Beispiel Status- und Fehlermeldungen eines digitalen Systems) in der Regel auf den Einsatz von Unicode verzichtet und auf den weniger komplexen ASCII-Code zurückgegriffen.

2.5.5 7-Segment-Code

Der *7-Segment-Code* wird ausschließlich zur Codierung von Zahlen verwendet, die mithilfe einer einfachen Anzeige dargestellt werden sollen. Sehr weit verbreitet sind 7-Segment-Anzeigen in digitalen Weckern, in denen sie zur Anzeige der Uhrzeit dienen. Auch bei einfachen Taschenrechnern kommen Segment-Anzeigen zum Einsatz. Ein Beispiel einer solchen Anzeige auf einer Platine für digitaltechnische Experimente ist in Abb. 2.13 dargestellt.

Tab. 2.13 Siebenstelliger ASCII-Code

				a_6	0	0	0	0	1	1	1	1
				a_5	0	0	1	1	0	0	1	1
				a_4	0	1	0	1	0	1	0	1
a_3	a_2	a_1	a_0									
0	0	0	0		NUL	DLE	SP	0	@	P	`	p
0	0	0	1		SOH	DC1	!	1	A	Q	a	q
0	0	1	0		STX	DC2	“	2	B	R	b	r
0	0	1	1		ETX	DC3	#	3	C	S	c	s
0	1	0	0		EOT	DC4	$	4	D	T	d	t
0	1	0	1		ENQ	NAK	%	5	E	U	e	u
0	1	1	0		ACK	SYN	&	6	F	V	f	v
0	1	1	1		BEL	ETB	‘	7	G	W	g	w
1	0	0	0		BS	CAN	(	8	H	X	h	x
1	0	0	1		HT	EM	)	9	I	Y	i	y
1	0	1	0		LF	SUB	*	:	J	Z	j	z
1	0	1	1		VT	ESC	+	;	K	[	k	{
1	1	0	0		FF	FS	,	<	L	\	l	|
1	1	0	1		CR	GS	–	=	M	]	m	}
1	1	1	0		SO	RS	.	>	N	^	n	~
1	1	1	1		SIX	US2	/	?	O	_	o	DEL

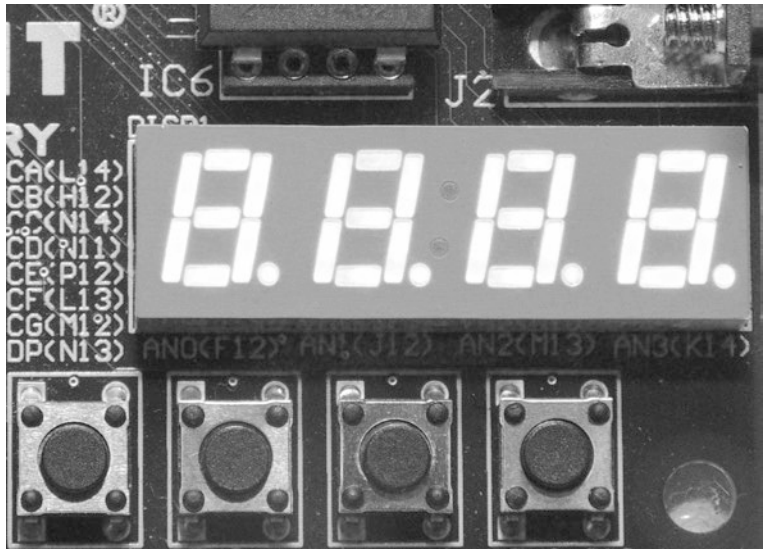


Abb. 2.13 Vierstellige 7-Segment-Anzeige

Die Darstellung der Ziffern wird häufig durch Leuchtdioden realisiert, die in Form einer eckigen 8 angeordnet sind. Durch Einschalten ausgewählter Leuchtdioden können nicht nur die Ziffern 0 bis 9, sondern auch die Hexadezimalziffern A bis F (zum Teil als Kleinbuchstaben) angezeigt werden. Auf diese Weise kann pro Ziffer einer solchen Anzeige der Wert von jeweils 4 Bits visualisiert werden.

Um Hexadezimalziffern mithilfe einer 7-Segment-Anzeige darstellen zu können, müssen die 4 Bits einer Hexadezimalziffer in geeigneter Weise in 7 Bits zur Ansteuerung der Leuchtdioden der Anzeige umgewandelt werden. In Tab. 2.14 ist eine hierfür geeignete Codierung dargestellt, wobei davon ausgegangen wird, dass eine 1 einer leuchtenden LED entspricht. Tab. 2.14 zeigt die Zuordnung zwischen den Bits des Codewortes *(a* bis *g)* und den LEDs der Anzeige (Abb. 2.14).

Tab. 2.14 Codierung einer Hexadezimalziffer für die Ausgabe auf einer 7-Segment-Anzeige

Hex-Ziffer	Code für die Ansteuerung der Segmente						
	a	b	c	d	e	f	g
0	1	1	1	1	1	1	0
1	0	1	1	0	0	0	0
2	1	1	0	1	1	0	1
3	1	1	1	1	0	0	1
4	0	1	1	0	0	1	1
5	1	0	1	1	0	1	1
6	1	0	1	1	1	1	1
7	1	1	1	0	0	0	0
8	1	1	1	1	1	1	1
9	1	1	1	1	0	1	1
A	1	1	1	0	1	1	1
b	0	0	1	1	1	1	1
C	1	0	0	1	1	1	0
d	0	1	1	1	1	0	1
E	1	0	0	1	1	1	1
F	1	0	0	0	1	1	1

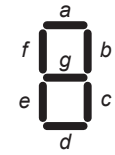


Abb. 2.14 Kennzeichnung der LEDs einer 7-Segment-Anzeige mit den Buchstaben a bis g

2.6 Übungsaufgaben

Prüfen Sie sich selbst mithilfe der folgenden Aufgaben. Am Ende dieses Buches finden Sie die Lösungen.

Aufgabe 2.1
Stellen Sie die Dezimalzahl 57_{10} in anderen Zahlensystemen dar:

a) binär
b) oktal
c) hexadezimal

Aufgabe 2.2
Welchen dezimalen Wert repräsentiert die Bitfolge „10010111“, wenn es sich

a) um eine vorzeichenlose Dualzahl handelt?
b) um eine Zweierkomplement-Zahl handelt?
c) um eine BCD-codierte Zahl handelt?

Aufgabe 2.3
Wie viele Bits sind für die Darstellung des Wertes 32_{10} erforderlich, wenn als Zahlendarstellung

a) die vorzeichenlose Dualzahlen-Darstellung gewählt wird?
b) die binäre Vorzeichen-Betrag-Darstellung gewählt wird?
c) die Zweierkomplement-Darstellung gewählt wird?

Aufgabe 2.4
Welcher Zahlenbereich kann mit 8 Bits dargestellt werden, wenn die folgenden Darstellungen gewählt werden?

a) vorzeichenlos
b) Vorzeichen-Betrag
c) Zweierkomplement

Aufgabe 2.5
Die nachfolgenden 6-Bit-Zahlen sollen addiert werden. Bestimmen Sie jeweils das (6 bit breite) Ergebnis für den Fall, dass es sich um vorzeichenlose Dualzahlen handelt und ermitteln Sie, ob bei der Addition ein Überlauf auftritt.

a) $110011 + 001010$
b) $100010 + 101001$

c) 010111 + 101101
d) Wie würden sich die Ergebnisse ändern, wenn die Operanden und das Ergebnis die Zweierkomplement-Darstellung verwenden?
e) Was würde sich im Hinblick auf Bereichsüberschreitungen (Überlauf) ändern, wenn die Operanden und das Ergebnis die Zweierkomplement-Darstellung verwenden?

Aufgabe 2.6
Nachfolgend sind 8-Bit-Zahlen in Hexadezimal-Darstellung angegeben. Diese Zahlen sollen addiert werden. Bestimmen Sie jeweils das Ergebnis in Hexadezimal-Darstellung und ermitteln Sie, ob Bereichsüberschreitungen auftreten. Die Zahlenwerte sollen sowohl als vorzeichenlose Dualzahlen als auch als Zweierkomplement-Zahlen interpretiert werden.

Hinweis: Sie können die Zahlen zunächst in eine binäre Darstellung überführen, eine binäre Addition durchführen und anschließend das binäre Ergebnis in einer hexadezimale Darstellung überführen. Einfacher ist es, wenn Sie die Subtraktion direkt in der Hexadezimal-Darstellung durchführen. Wenden Sie hierzu die Rechenregeln aus der Grundschule an und beachten Sie, dass der 10er-Übergang des Dezimalsystems einem 16er-Übergang im Hexadezimalsystem entspricht. Beide Wege führen zum Ziel.

a) 27 + 33
b) 9A + 89
c) DE + CD

Aufgabe 2.7
Nachfolgend sind 8-Bit-Zahlen in Hexadezimal-Darstellung angegeben. Diese Zahlen sollen subtrahiert werden. Bestimmen Sie jeweils das (8 bit breite) Ergebnis in Hexadezimal-Darstellung und ermitteln Sie, ob Bereichsüberschreitungen auftreten. Die Zahlenwerte sollen sowohl als vorzeichenlose Dualzahlen als auch als Zweierkomplement-Zahlen interpretiert werden.

Hinweis: Wie bei der Addition ist auch hier ist die Berechnung im Hexadezimalsystem einfacher.

a) A9 − 42
b) 83 − 37
c) 5C − BF

Aufgabe 2.8
Welche besondere Eigenschaft besitzt der Gray-Code?

Aufgabe 2.9
Welche der folgenden Bitfolgen sind Pseudotetraden des BCD-Codes? *(mehrere Antworten können richtig sein)*

a) 1000
b) 1011
c) 1100
d) 1001

Aufgabe 2.10
Es wird ein 1-aus-8 Code betrachtet.

a) Welche Wortbreite besitzt ein Codewort?
b) Wie viele unterschiedliche Codewörter lassen sich darstellen?

Aufgabe 2.11
Achtung, Transferleistung erforderlich: Man kann theoretisch auch für das Dezimalsystem eine Komplementdarstellung wählen, also eine Zahlendarstellung im „Zehnerkomplement“. Wie würden in dieser Zahlendarstellung die folgenden Werte dargestellt werden, wenn 3 Dezimalstellen zur Verfügung stehen?

a) 0
b) -1
c) -2
d) -10

Einführung in VHDL

3

In Kapitel 1 wurden bereits die wichtigsten Grundelemente digitaler Systeme vorgestellt. Eine digitale Hardware verarbeitet Informationen, indem die Eingangssignale zum Beispiel mithilfe von logischen Grundelementen, den Gattern, verknüpft werden. Wie kann man nun festlegen wie die Gatter verschaltet werden sollen, um die Ausgangssignale einer Schaltung zu berechnen?

Möglicherweise kennen Sie Schaltpläne für elektrische Geräte. Durch grafische Symbole werden die Komponenten des Gerätes beschrieben und die elektrischen Verbindungen werden durch Striche dargestellt. Eine naheliegende Möglichkeit wäre es, diese grafische Darstellung auch zur Spezifikation einer digitalen Schaltung zu verwenden. Die elektrisch zu verbindenden Komponenten könnten dann zum Beispiel logische Grundelemente sein. Man kann hierbei auch eine hierarchische Darstellung wählen, indem einzelne Elemente zu Blöcken zusammenfasst werden, die dann in anderen Teilen des Schaltplans als Module eingesetzt werden. Diese Form der Schaltungsbeschreibung wurde tatsächlich in den Anfängen der Digitaltechnik eingesetzt. Allerdings durchlief die Digitaltechnik von Beginn an eine rasante Entwicklung. Bis heute verdoppelt sich etwa alle zwei Jahre die Anzahl der Schaltfunktionen, die sich in einer einzelnen elektronischen Komponente (einem „Chip“) integrieren lässt. Dies bedeutet unter anderem, dass die Komplexität digitaler Systeme kontinuierlich zunimmt. Mit den Fortschritten der Digitaltechnik wurden die Schaltpläne zunehmend komplexer und man suchte etwa ab Mitte der 1980er-Jahre nach Alternativen zur Schaltplaneingabe.

Als Lösung wurden die sogenannten Hardwarebeschreibungssprachen (engl. *Hardware Description Language, HDL*) erfunden. Diese Sprachen ermöglichen es, die Funktion einer digitalen Schaltung, ähnlich wie ein Programm für einen Rechner, in textueller Form zu beschreiben. Im Gegensatz zu den üblichen Software-Programmiersprachen wie C/C++ oder Java, besitzen Hardwarebeschreibungssprachen Sprachelemente, die besonders für die Beschreibung digitaler Hardware geeignet sind. In der Praxis werden zwei

W. Gehrke et al., *Digitaltechnik,* Springer-Lehrbuch,
DOI 10.1007/978-3-662-49731-9_3

Beschreibungssprachen eingesetzt: Verilog und VHDL (***V**ery High Speed Integrated Circuits **H**ardware **D**escription **L**anguage*). VHDL bietet gegenüber Verilog einen größeren Funktionsumfang und wird daher meist als bevorzugte Sprache zur Beschreibung digitaler Systeme eingesetzt.

In diesem Kapitel werden die Grundlagen der Sprache VHDL vorgestellt. Nachdem Sie dieses Kapitel gelesen haben, kennen Sie die wichtigsten Sprachelemente und sind in der Lage eigene digitale Schaltungen in VHDL zu beschreiben. Praktische Hinweise für die Durchführung eigener VHDL-Experimente finden Sie auch auf der im Vorwort angegebenen Internetseite zum Buch.

3.1 Designmethodik im Überblick

Der Ausgangspunkt einer HDL-basierten Beschreibung sind eine oder mehrere VHDL-Dateien, welche die Funktion der späteren digitalen Hardware festlegen. Wie bei der Erstellung von Software handelt es sich um Textdateien, die eine für den Menschen lesbare Beschreibung der gewünschten Module enthalten.

Nicht jeder syntaktisch richtige VHDL-Code kann auch in Hardware überführt werden. VHDL bietet zum Beispiel Sprachkonstrukte um Dateien einzulesen oder Texte auszugeben. Diese Sprachelemente können nicht in Hardwaremodule übersetzt werden. Der Compiler, welcher aus den VHDL-Beschreibungen Hardware erzeugt, würde entsprechende Warn- bzw. Fehlermeldungen ausgeben. Da der Übersetzungsprozess in der Regel als *Synthese* bezeichnet wird, spricht man auch von „synthesefähigem“ oder „synthetisierbarem“ VHDL-Code.

Die nicht-synthetisierbaren Sprachelemente werden vielfach in sogenannten *Testbenches* eingesetzt. Als eine Testbench wird VHDL-Code bezeichnet, der zur Überprüfung der Funktion des synthetisierten Codes geschrieben wurde.

Die VHDL-Dateien werden mithilfe eines sogenannten Simulators auf einem PC ausgeführt. Der Simulator ermöglicht es, den zeitlichen Verlauf aller Signale zu visualisieren oder in Dateien auf dem PC abzulegen.

Für die Simulation werden die zu testenden VHDL-Module als Komponenten in den Testbench-Code eingefügt. Der Code der Testbench legt wechselnde Eingangssignale (im Fachjargon „Stimuli“) an die Eingänge der zu prüfende Komponente an. Das Konzept einer VHDL-Testbench, in die eine zu prüfende VHDL-Komponente eingesetzt wird, ist in Abb. 3.1 dargestellt.

Der zeitliche Verlauf von Eingangs- und Ausgangssignalen als auch von internen Signalen einer VHDL-Beschreibung kann während der Simulation mithilfe sogenannter Waveform-Viewer visualisiert werden. Die grafische Darstellung der Signalverläufe gibt häufig wichtige Hinweise zur Lokalisierung eines Fehlers und ist ein nicht wegzudenkendes Handwerkszeug der VHDL-Entwicklung. Ein Beispiel für die Ausgabe eines Waveform Viewers ist in Abb. 3.2 dargestellt. In diesem Beispiel wird das Ergebnis der UND-Verknüpfung von a und b dem Signal q zugewiesen.

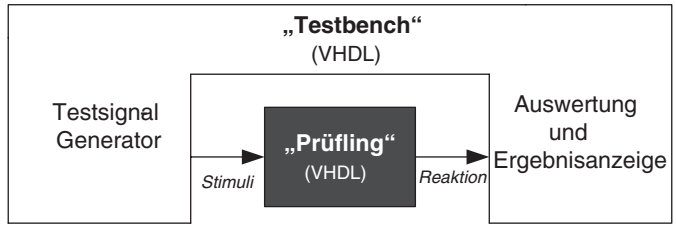


Abb. 3.1 Verifikation einer Komponente mithilfe einer VHDL-Testbench

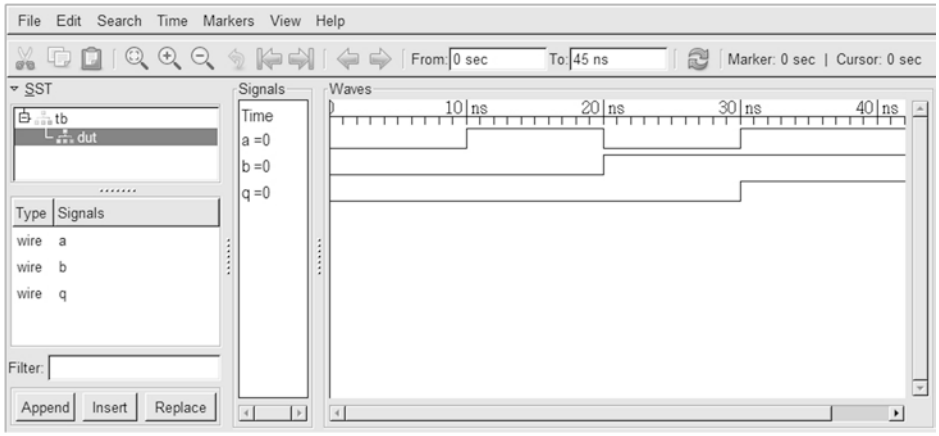


Abb. 3.2 Waveform Viewer

In Abb. 3.3 ist der Ablauf eines VHDL-basierten Entwurfsprozesses dargestellt: Der Ausgangspunkt sind VHDL-Dateien, welche die gewünschte Funktion der digitalen Hardware beschreiben. Darüber hinaus werden Testbench-Dateien erstellt. Mithilfe der Simulation der VHDL-Hardware-Module in Kombination mit den Testbench-Dateien wird die korrekte Funktion der Hardware-Beschreibung überprüft und gegebenenfalls entdecktes Fehlverhalten korrigiert. Anschließend kann die Synthese, also die Überführung der VHDL-Hardware-Beschreibungen in digitale Hardware, erfolgen. Auch nach diesem Schritt können Änderungen am VHDL-Code erforderlich werden um beispielsweise den benötigten Realisierungsaufwand zu reduzieren oder das zeitliche Verhalten des Systems zu verbessern. Der Entwurfsprozess ist also ein iterativer Prozess, bei dem (insbesondere bei komplexen Systemen) die Schritte *Simulation* und *Synthese* mehrfach durchlaufen werden.

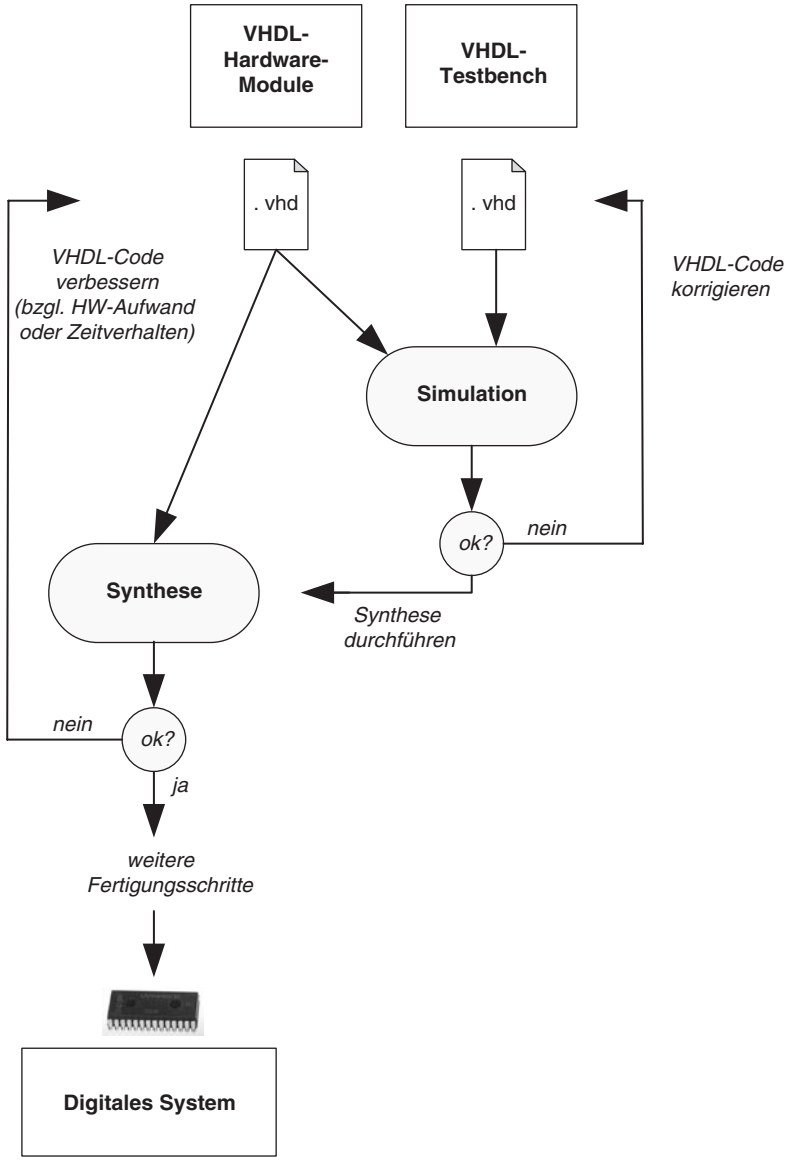


Abb. 3.3 VHDL-basierter Entwurfsprozess

3.2 Grundstruktur eines VHDL-Moduls

Ein VHDL-Modul repräsentiert meistens einen Teil eines größeren Systems und wird in Form einer Textdatei beschrieben. In diesem Abschnitt werden einige grundlegende Konzepte und Sprachelemente vorgestellt, die bei einem VHDL-basierten Hardwareentwurf verwendet werden.

3.2.1 Bibliotheken

VHDL-Beschreibungen müssen vor ihrer Verwendung (in einer Simulation oder für die Synthese) zunächst kompiliert werden. Die durch den Übersetzungsvorgang erzeugte binäre Beschreibung wird in einer sogenannten *Bibliothek* abgelegt und kann anschließend mit anderen kompilierten VHDL-Beschreibungen zu einer Simulationsdatei bzw. der zu realisierenden Hardware zusammengefügt werden.

Es ist freigestellt, ob man für jedes VHDL-Modul eine eigene Bibliothek anlegt oder ob mehrere VHDL-Dateien in einer gemeinsamen Bibliothek abgelegt werden. Insbesondere für kleinere Systeme ist es häufig völlig ausreichend, eine gemeinsame Bibliothek für alle übersetzten VHDL-Dateien zu wählen.

Ein Aufruf eines VHDL Compilers zum Übersetzen der VHDL-Datei *my_module.vhd* kann wie folgt aussehen:

```
vcom -work my_work_lib my_module.vhd
```

In diesem Beispiel wird der VHDL-Compiler *vcom* aufgerufen. Mithilfe des Schalters *-work* wird der Name der zu verwendenden Bibliothek angegeben – in diesem Beispiel *my_work_lib*.

Drei Bibliotheken sind besonders wichtig: *work*, *std* und *ieee*.

Der Bibliotheksname *work* ist ein Synonym für die jeweils aktuelle Arbeitsbibliothek, in der die Ergebnisse des Übersetzungsvorgangs abgelegt werden. Es ist zum Beispiel möglich, alle VHDL-Elemente in einer Bibliothek *my_work_lib* abzulegen und die bereits übersetzten Elemente wahlweise über den Namen *work* oder *my_work_lib* zu referenzieren. Da *work* ein vordefinierter symbolischer Name für die aktuelle Arbeitsbibliothek ist, sollte *work* nicht als Bibliotheksname verwendet werden. Andernfalls hätte die Referenzierung der Bibliothek *work* zwei mögliche Bedeutungen: Es kann sich um die aktuelle Arbeitsbibliothek (welche einen beliebigen Namen besitzen kann) oder um die Bibliothek mit dem Namen *work* handeln.

In der Bibliothek *std* sind einige grundlegende Sprachkonstrukte und Datentypen definiert. Darüber hinaus enthält die Bibliothek *std* auch Funktionen zur Ein- und Ausgabe.

Die Bibliothek *ieee* enthält wichtige und häufig verwendete Datentypen sowie viele hilfreiche Funktionen. Die wichtigsten Elemente dieser Bibliothek werden im Verlauf dieses Kapitels vorgestellt und in Kapitel 8 weiter vertieft.

Sollen Bibliotheken, die nicht bereits im VHDL-Standard vordefiniert sind (dies ist für die Bibliotheken *work* und *std* der Fall), müssen sie vor ihrer Verwendung mithilfe einer Library-Anweisung bekanntgemacht werden. Anschließend wird mithilfe einer Use-Anweisung ausgewählt, welche Teile der Bibliothek in dem nachfolgenden VHDL-Code verwendet werden sollen. Hinter dem Schlüsselwort *use* folgt zunächst die Angabe der gewünschten Bibliothek und dann, durch Punkte abgetrennt, das zu verwendenden Paket der Bibliothek sowie die Elemente aus dem jeweiligen Paket. Meist ist eine explizite Auswahl einzelner Elemente nicht erforderlich: Man wählt mit dem Schlüsselwort

all einfach alle vorhandenen Elemente aus. Im nachfolgenden VHDL-Code stehen dann alle Elemente des jeweiligen Bibliothekspakets zur Verfügung.

Die folgenden Beispiele verdeutlichen die Syntax zur Verwendung von Bibliotheken:

```
-- Die Bibliotheken std und work benötigen keine Library-Anweisung
-- mithilfe einer Use-Anweisung werden die Teile der Bibliothek bekannt
-- gemacht, die in der nachfolgenden VHDL-Beschreibung verwendet
-- werden
-- Verwendung von Ein-/Ausgabe-Funktionen aus der Bibliothek std
use std.textio.all;
-- Verwendung von Funktionen eines eigenen Paketes, welches bereits
-- in der aktuellen Arbeitsbibliothek abgelegt (übersetzt) worden ist
use work.my_package.all;
-- Verwendung von Datentypen, Funktionen etc.
-- wie sie im IEEE-Standard 1164 festgelegt worden sind
library ieee;
use ieee.std_logic_1164.all;
```

3.2.2 Entity und Architecture

VHDL-Beschreibungen entsprechen einzelnen Hardware-Komponenten. Damit eine solche Komponente vollständig beschrieben ist, müssen vor allem zwei Teile der Beschreibung erstellt werden:

1. Die äußeren Anschlüsse der Komponente: Welche Signale werden in das Modul hineingeführt und welche kommen heraus? Welche Wortbreite haben die Signale?
2. Die Funktion des Moduls: Nach welcher digitalen Rechenvorschrift werden die Ausgangssignale aus den Eingangssignalen berechnet?

Die Beschreibung der „Sicht von außen“ wird als *Entity* und das „Innenleben“ als *Architecture* bezeichnet. Diese beiden Teile eines VHDL-Moduls werden häufig in einer gemeinsamen Textdatei abgelegt. Die Beschreibung einer Entity beginnt mit dem VHDL-Schlüsselwort *entity.* Der Name des Moduls wird durch die Schlüsselwörter *entity* und *is* eingerahmt. Das Ende der Entity-Beschreibung wird durch *end* gekennzeichnet. Zwischen dem Beginn und dem Ende der Entity werden die von außen sichtbaren Eigenschaften des Moduls definiert. Anschlüsse für Eingangs- und Ausgangssignale, im englischen Sprachgebrauch als *Ports* bezeichnet, werden in Form einer Liste angegeben, welche mit dem Schlüsselwort *port* eingeleitet wird. Der eigentliche Inhalt der Portliste wird in Klammern angegeben, wobei die einzelnen Listenelemente durch ein Semikolon voneinander getrennt werden. Für jeden Port wird ein Name angegeben und festgelegt, ob es sich um einen Eingang oder einen Ausgang handelt (Schlüsselwörter *in* und *out*).

Darüber hinaus muss für die Anschlüsse ein Datentyp angegeben werden. In der Praxis hat sich für die Beschreibungen einzelner Bits der Datentyp *std_logic* (gesprochen: *„standard logic“*) durchgesetzt, welcher durch die Norm IEEE 1164 definiert ist. Um diesen Datentyp verwenden zu können, muss das Paket std_logic_1164 aus der IEEE-Bibliothek hinzugefügt werden.

Betrachten wir das Beispiel eines UND-Gatters mit zwei Eingängen. Die Entity kann in VHDL wie folgt realisiert werden:

```
library ieee;
use ieee.std_logic_1164.all;
entity and_2 is
   port (a : in  std_logic;
         b : in  std_logic;
         q : out std_logic);
end;
```

Groß- und Kleinschreibung wird in VHDL nicht unterschieden und daher kann für alle Sprachelemente sowohl Groß- als auch Kleinschrift verwendet werden. Selbst Mischformen sind erlaubt und syntaktisch korrekt. So kann das Schlüsselwort *entity* auch *Entity* oder *eNTiTy* geschrieben werden.

Die Architecture-Beschreibung startet mit dem Schlüsselwort *architecture,* gefolgt von einem Namen der Architecture. Welcher Entity die Architecture zuzuordnen ist, wird direkt danach mit *of* festgelegt. Zwischen den Schlüsselwörtern *begin* und *end* wird der VHDL-Code eingefügt, der die Funktion des Moduls beschreibt. Die Architecture eines UND-Gatters ist recht übersichtlich. Die Zuweisung der UND-Verknüpfung der beiden Eingänge an den Ausgangsport benötigt nur eine Codezeile.

```
architecture behave of and_2 is
begin
   q <= a and b;
end;
```

3.2.3 Bezeichner

Namen von VHDL-Elementen wie zum Beispiel Entity-, Architecture-, oder Signalnamen usw. beginnen immer mit einem Buchstaben. Anschließend sind sowohl Buchstaben als auch Zahlen oder der Unterstrich „_“ erlaubt. Die Verwendung von Schlüsselwörtern ist nicht erlaubt. In Tab. 3.1 sind die VHDL-Schlüsselwörter zusammengefasst.

Es ist nicht unbedingt notwendig die Bedeutung aller Schlüsselwörter zu verstehen. Einige der reservierten Wörter werden selbst von Experten nur selten verwendet.

Tab. 3.1 Übersicht über reservierte Wörter der Hardwarebeschreibungssprache VHDL

abs	downto	library	postponed	srl
access	else	linkage	procedure	subtype
after	elsif	literal	process	then
alias	end	loop	pure	to
all	entity	map	range	transport
and	exit	mod	record	type
architecture	file	nand	register	unaffected
array	for	new	reject	units
assert	function	next	rem	until
attribute	generate	nor	report	use
begin	generic	not	return	variable
block	group	null	rol	wait
body	guarded	of	ror	when
buffer	if	on	select	while
bus	impure	open	severity	with
case	in	or	signal	xnor
component	inertial	others	shared	xor
configuration	inout	out	sla	
constant	is	package	sll	
disconnect	label	port	sra	

Für die Erstellung von VHDL-Code ist ein kontextsensitiver Editor empfehlenswert, der Schlüsselwörter automatisch farblich hervorhebt. Damit kann zum Beispiel erkannt werden, ob versehentlich ein Schlüsselwort als Bezeichnung eines VHDL-Elements verwendet wird.

3.3 Grundlegende Datentypen

Genauso wie Programmiersprachen zur Entwicklung von Software, stellt VHDL verschiedene Datentypen zur Verfügung. In diesem Abschnitt werden die wichtigsten Datentypen vorgestellt.

3.3.1 Integer

Mithilfe des Datentyps *integer* können ganze Zahlen im Bereich von -2^{31} bis $+2^{31}-1$ dargestellt werden, also der Zahlenbereich, welcher mit einer 32 bit breiten Zweierkomplementzahl dargestellt werden kann.

Das Syntheseprogramm, das die VHDL-Beschreibung in Hardware überführt, wird für Integer-Werte zunächst eine Wortbreite von 32 Bit annehmen – unabhängig davon, ob diese Wortbreite für die zu verarbeitenden Daten wirklich benötigt wird. Es besteht daher die Gefahr, dass das Syntheseprogramm nicht erkennt, dass die in VHDL beschriebene Aufgabe auch mit einer geringeren Wortbreite lösbar ist und letztlich eine Schaltung für 32 Bit realisiert, obwohl auch eine weniger komplexe Schaltung ausreichen würde. Um diese Gefahr zu vermeiden können die im Folgenden vorgestellten Datentypen *std_logic_vector, signed* und *unsigned* eingesetzt werden. Sie zeichnen sich dadurch aus, dass man die zu verwendende Wortbreite explizit angibt.

3.3.2 Std_logic

Der Datentyp std_logic wurde bereits weiter vorne in diesem Kapitel zur Beschreibung einzelner Bits eingeführt. Dieser Datentyp repräsentiert ein einzelnes Bit, das die Werte 0 oder 1 annehmen kann. Der Datentyp std_logic bietet darüber hinaus noch weitergehende Möglichkeiten.

So wird zur Beschreibung des Einschaltzustands eines Signals, welcher zufällig 0 oder 1 sein kann, ein weiterer Wert benötigt. Der Datentyp std_logic bietet hierfür den Wert *Undefined* an, welcher mit dem Buchstaben *U* abgekürzt wird.

Neben 0, 1, und *U* bietet der Datentyp noch sechs weitere Werte. Eine Übersicht über die neunwertige Logik des Datentyps std_logic ist in Tab. 3.2 dargestellt.

Nicht alle neun möglichen Werte sind gleichermaßen praxisrelevant. Einige können zum Beispiel verwendet werden, wenn Ausgänge mehrerer Gatter auf eine gemeinsame Leitung geführt werden. Hierzu zählen die Werte *Z, L, H* und *W*. Die Möglichkeit, mehrere Gatterausgänge an eine gemeinsame physikalische Leitung anzuschließen, ist jedoch ein Sonderfall.

Es verbleiben neben der 0 und der 1 also noch die Werte *U, X* und – (*Don't-Care*). Obwohl Sie diese Werte in einer realen Schaltung nicht beobachten werden, da die

Tab. 3.2 Werte des Datentyps std_logic

Wert	Bedeutung
0	Logische 0
1	Logische 1
U	Undefiniert
X	Unbekannt
–	„Don't-Care" (für Eingänge: Wert ist beliebig)
Z	Hochohmig
L	„Schwache" logische 0
H	„Schwache" logische 1
W	„Schwach" unbekannt

Leitungen entweder den Wert 0 oder den Wert 1 besitzen, sind die zusätzlichen Signalzustände hilfreich. Die Werte *U* und *X* werden Ihnen bei der Simulation eines VHDL-Modells begegnen. Der Wert *U* deutet darauf hin, dass sich in der simulierten Schaltung Signale befinden, die noch nicht auf einen definierten Wert initialisiert worden sind. Insbesondere zu Beginn einer Simulation werden Sie viele Signale mit dem Wert *U* beobachten können. Aufgrund von VHDL-Zuweisungen werden diese Signale meist relativ schnell einen definierten Wert (meist 0 oder 1) erhalten. Ist ein Signal mit dem Wert *U* länger zu beobachten, sollte der Grund für dieses Verhalten analysiert werden. Es kann sein, dass die fehlende Zuweisung eines Wertes an dieses Signal einen Fehler darstellt, der zu einem Fehlverhalten der Hardware führen kann.

Der Wert *X* tritt auf, wenn unbeabsichtigt zwei Ausgänge mit unterschiedlichen logischen Werten auf das gleiche Signal geführt werden. Darüber hinaus kann der Wert *X* in der Simulation entstehen, wenn undefinierte oder unbekannte Signale in logischen Verknüpfungen verwendet werden. Werden in einer Simulation Signale mit dem Wert *X* beobachtet, muss die Ursache für dieses Verhalten untersucht werden. In den meisten Fällen liegt ein Fehler im VHDL-Code vor, welcher vor dem Umsetzen der VHDL-Beschreibung in Hardware behoben werden muss.

Mithilfe des Wertes *Don't-Care* kann in einer VHDL-Beschreibung zum Ausdruck gebracht werden, dass der Wert eines bestimmten Signals unerheblich für die Funktion der Schaltung ist und somit dieses Signal für die Berechnung der Ausgangswerte nicht beachtet werden muss. Meist kann diese Information bei der Optimierung der synthetisierten Hardware verwendet werden, sodass eine schnellere oder weniger aufwendige Hardware erzeugt werden kann.

3.3.3 Std_logic_vector

Viele digitale Systeme lassen sich einfacher und übersichtlicher in VHDL beschreiben, wenn man die Möglichkeit nutzt, einzelne Bits zu gruppieren. Hierzu kann der Datentyp *std_logic_vector* (beziehungsweise *std_ulogic_vector*) verwendet werden.

Die Indexgrenzen des Vektors werden in Klammern angegeben. Meist wird hierbei eine absteigende Indizierung verwendet, zum Beispiel (7 *downto* 0).

Nehmen wir an, Sie möchten eine Schaltung realisieren, die vier UND-Gatter mit jeweils zwei Eingängen enthalten soll. Selbstverständlich kann man diese Schaltung mithilfe von 8 Eingängen und 4 Ausgängen vom Datentyp *std_logic* realisieren. Allerdings würde in diesem Fall die Entity-Beschreibung des Moduls 12 Ports enthalten und in der Architecture müssten vier Signalzuweisungen, für jeden der vier Ausgänge der Schaltung, vorgenommen werden.

Die Problemstellung lässt sich bei Verwendung des Datentyps *std_logic_vector* deutlich übersichtlicher lösen:

```
library ieee;
use ieee.std_logic_1164.all;

entity and_2x4 is
   port (a : in  std_logic_vector (3 downto 0);
         b : in  std_logic_vector (3 downto 0);
         q : out std_logic_vector (3 downto 0));
end;

architecture behave of and_2x4 is
begin
   q <= a and b;
end;
```

VHDL unterstützt Operatoren, die auf Vektoren angewendet werden. In der Codezeile *q <= a and b* wird dies ausgenutzt. Diese Zeile führt eine bitweise UND-Verknüpfung der einzelnen Komponenten der Vektoren *a* und *b* aus und weist das Ergebnis den jeweiligen Bits des Ausgangs *q* zu. Es wäre auch möglich, diese Zuweisungen explizit auszuführen, indem auf die einzelnen Elemente der Vektoren zugegriffen wird:

```
architecture behave_2 of and_2x4 is
begin
   q(0) <= a(0) and b(0);
   q(1) <= a(1) and b(1);
   q(2) <= a(2) and b(2);
   q(3) <= a(3) and b(3);
end;
```

Diese Schreibweise würde zum gleichen Ergebnis führen wie die UND-Verknüpfung auf Basis von Vektoren. Es ist eine Frage des „Coding-Styles“ welche der beiden Varianten bevorzugt wird. Im Allgemeinen sollte jedoch aus Gründen der Übersichtlichkeit die vektorielle Schreibweise vorrangig verwendet werden.

Im Zusammenhang mit Vektoren wird häufig die Frage gestellt, ob es möglich ist, die Elemente eines Vektors zu vertauschen indem ein Vektor mit absteigender Indizierung (zum Beispiel *7 downto 0*) einem Vektor mit aufsteigender Indizierung (zum Beispiel *0 to 7*) zugewiesen wird. Obwohl die Elementanzahl in den Vektoren übereinstimmt, ist eine solche Zuweisung nicht zulässig. Die beiden Vektoren besitzen unterschiedliche Datentypen und dürfen daher nicht direkt einander zugewiesen werden.

3.3.4 Signed und Unsigned

Der Datentyp *std_logic_vector* ist eine Zusammenfassung einzelner Bits zu einem Vektor. Welche Information durch den Bitvektor dargestellt wird, ist durch den Datentyp

nicht eindeutig definiert. Es könnten völlig unabhängige Bits sein, die aus Gründen der Übersichtlichkeit gruppiert wurden. Genauso gut könnte die Zusammenfassung der Bits einen Zahlenwert darstellen. Im letzteren Fall wäre es wünschenswert, dass für die Vektoren nicht nur logische Funktionen, sondern auch arithmetische Operationen wie Addition oder Subtraktion definiert wären.

VHDL verwendet im Hinblick auf den Datentyp *std_logic_vector* eine strikte Philosophie: Der Datentyp *std_logic_vector* beschreibt die Zusammenfassung einzelner Bits. Dass diese Bits gemeinsam betrachtet einen Zahlenwert darstellen könnten, wird von VHDL ausgeschlossen und es werden im Sprachstandard keine arithmetischen Operationen dafür zur Verfügung gestellt.

Soll in VHDL die Kombination einzelner Bits als eine Zahl interpretiert werden, werden die Datentypen *signed* und *unsigned* verwendet. Ähnlich wie beim Datentyp *std_logic_vector* können mit *signed* und *unsigned* beliebig große Vektoren gebildet werden. Die Bits werden als eine Zweierkomplementzahl beziehungsweise als vorzeichenlose Dualzahl interpretiert werden.

Diese Datentypen sind ebenfalls vom IEEE standardisiert worden und stehen im Paket *numeric_std* der IEEE-Bibliothek zur Verfügung. Für diese Datentypen sind arithmetische Operationen wie die Addition definiert und eine Addiererschaltung für vorzeichenlose Zahlen mit der Wortbreite 4 bit kann wie folgt implementiert werden:

```
library ieee;
use ieee.numeric_std.all;

entity addu_4 is
   port (a : in  unsigned (3 downto 0);
         b : in  unsigned (3 downto 0);
         q : out unsigned (3 downto 0));
end;

architecture behave of addu_4 is
begin
   q <= a + b;
end;
```

3.3.5 Konstanten

Möchte man einem Signal eine Konstante zuweisen, muss hierbei auf den Datentyp geachtet werden. Bei Signalen vom Datentyp *integer* erfolgt die Zuweisung – wie in einer Software-Programmiersprache – in Form einer dezimalen Zahl. Möchte man dagegen den Zahlenwert in hexadezimaler, binärer oder einer anderen nicht-dezimalen Schreibweise angeben, muss vor der Zahl der Radix der Zahlendarstellung angegeben werden. Die nachfolgende Zahl wird durch Doppelkreuze *(#)* eingerahmt. So würde die Hexadezimalzahl *BEEF* im VHDL-Code als *16#BEEF#* angegeben werden.

Konstanten vom Datentyp *std_logic_vector* oder *signed* bzw. *unsigned* werden in Anführungszeichen in binärer Form angeben. Mit einem vorangestellten *x* lassen sich die Werte auch in hexadezimaler Schreibweise angeben, wobei jede Hexadezimalstelle exakt 4 bit repräsentiert.

Die Zuweisung eines *std_logic*-Wertes erfolgt in einfachen (halben) Anführungszeichen.

Die folgenden Beispiele verdeutlichen die Möglichkeiten zur Angabe von Konstanten.

```
-- Exemplarische Konstantenzuweisungen
i <= 1234;          -- integer, dezimal
i <= 16#ABC#;       -- integer, hexadezimal
i <= 8#175#;        -- integer, oktal
i <= 2#01010111#;   -- integer, dual
sv8 <= "01000111";  -- std_logic_vector
sv8 <= "0UUX0111";  -- std_logic_vector
sv8 <= x"EF";       -- std_logic_vector, hexadezimal
s <= '1';           -- std_logic
b <= true;          -- boolean
```

Sehr nützlich ist die Zuweisung mithilfe der Others-Funktion. Diese ermöglicht es einzelnen Elementen eines Vektors Werte zuzuweisen und den restlichen Elementen *(others)* einen anderen Wert. Die Syntax wird durch die folgenden Beispiele verdeutlicht:

```
-- Diese Zeilen können…
   sv1 <= "01000001";
   sv2 <= "00111101";
   sv3 <= "00000000";
-- … mit Hilfe von "others" auch so formuliert werden:
   sv1 <= (0,6=>'1', others=>'0');
   sv2 <= (7,6,1 =>'0', others=>'1');
   sv3 <= (others=>'0');
```

3.3.6 Umwandlung zwischen Datentypen

Für die Umwandlung zwischen den Datentypen *integer, signed/unsigned* und *std_logic_vector* stehen verschiedene Funktionen zur Verfügung. So lässt sich beispielsweise ein *Unsigned*- bzw. *Signed*-Wert mit der Funktion *to_integer()* in einen Integer-Wert umwandeln. Für die umgekehrte Typumwandlung steht die Funktion *to_unsigned()* beziehungsweise *to_signed()* zur Verfügung. Für eine Umwandlung vom Datentyp *unsigned* bzw. *signed* in den Datentyp *std_logic_vector* kann die Funktion *std_logic_vector()* verwendet werden. Eine Umwandlung in die Datentypen *signed* und *unsigned* kann entsprechend mit den Funktionen *signed()* und *unsigned()* erfolgen.

In Abb. 3.4 sind die Funktionen zur Umwandlung zwischen den Datentypen *std_logic_vector, signed/unsigned* und *integer* grafisch dargestellt.

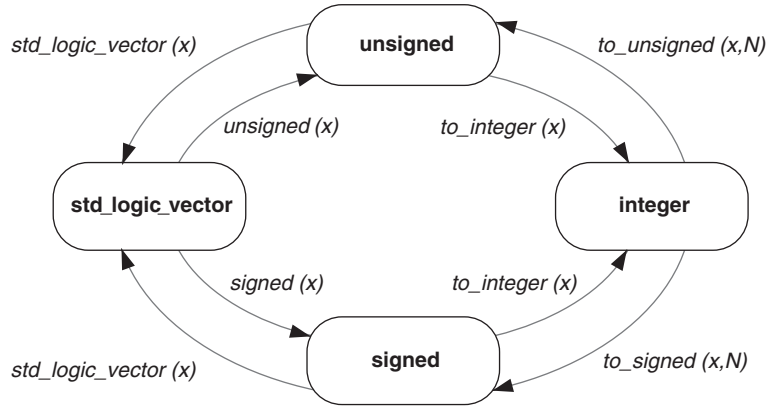


Abb. 3.4 Umwandlung zwischen wichtigen VHDL-Datentypen

Eine Umwandlung vom Datentyp *integer* in den Datentyp *std_logic_vector* kann nicht direkt erfolgen, sondern erfordert immer einen Zwischenschritt über den Datentypen *signed* bzw. *unsigned.*

Einige Beispiele für die Umwandlung der VHDL-Datentypen sind im Folgenden dargestellt.

```
-- Exemplarische Typumwandlungen
i    <= to_integer(s8);       -- signed -> integer
u8   <= to_unsigned(i,8);     -- integer -> unsigned
s8   <= to_signed(-123,8);    -- Ganzzahlige Konstante: Datentyp Integer
slv8 <= std_logic_vector(u8); -- unsigned -> std_logic_vector
i    <= to_integer(unsigned(slv8));       -- std_logic_vector -> integer
slv8 <= std_logic_vector(to_signed(i,8)); -- integer -> std_logic_vector
```

3.3.7 Datentyp Bit

In VHDL existiert auch der Datentyp *bit.* Objekte dieses Typs können die Werte 0 bzw. 1 annehmen, was auf den ersten Blick ausreichend erscheinen mag. In der Praxis besteht jedoch häufig der Wunsch einem Signal noch weitere Zustände, außer 0 oder 1, zuweisen zu können. Ein typisches Beispiel hierfür ist der Zustand eines Signals nach dem Einschalten eines Systems. Ist es 0 oder ist es 1? Möglicherweise „fällt“ das Signal auf einen zufälligen Initialwert, es ist also nach dem Einschalten manchmal 0 und manchmal 1. Der Einschaltzustand des Signals ist also weder eindeutig 0 noch eindeutig 1,

sondern *undefiniert.* Die Modellierung des undefinierten Einschaltzustands ist mithilfe des Datentyps *std_logic* möglich, mit dem Datentyp *bit* dagegen nicht. Daher wird in der Praxis der Typ *std_logic* bevorzugt eingesetzt und hat die Verwendung des Typs *bit* verdrängt.

3.4 Operatoren

Die UND-Verknüpfung wurde bereits in den vorangegangenen Abschnitten eingeführt. In diesem Abschnitt werden nun weitere wichtige Operatoren vorgestellt, die zur Beschreibung der Funktion einer Schaltung eingesetzt werden können. Nicht alle Operatoren lassen sich mit allen Datentypen verwenden. So ist es zum Beispiel nicht möglich zwei Werte vom Datentyp *std_logic_vector* zu addieren.

In den Tab. 3.3, 3.4 und 3.5 folgenden Tabellen sind wichtige VHDL-Operatoren zusammengestellt. Die Datentypen *integer, signed* und *unsigned* werden hierbei unter dem Begriff „numerisch“ zusammengefasst.

Die folgenden Beispiele sollen den Einsatz der Operatoren in VHDL verdeutlichen:

```
-- Beispiele für die Verwendung von VHDL-Operatoren
a    <= b or c;              -- Bitweises ODER
sig1 <= not sig2;             -- Bitweise Invertierung
u8_1 <= u8_2 + "00000011";    -- Addition
u8   <= to_unsigned(2**7,8);  -- Potenzierung

if s8 = to_signed(3,8) then     -- Vergleich
   slv5_1 = slv5_2 nand slv5_3; -- NAND (Nicht-UND)
end if;
```

Bei den arithmetischen Operatoren ist zu beachten, dass die Wortbreite des Ergebnisses mit der Wortbreite der Operanden identisch sein muss. Sie mögen vielleicht spontan einwenden wollen, dass dies zu Problemen führen kann: Wenn beispielsweise zwei 8 Bit breite vorzeichenlose Zahlen (Wertebereich: 0 … 255) addiert werden

Tab. 3.3 Logische VHDL-Operatoren

Schreibweise	Bedeutung	Datentypen	Synthetisierbar?
and	UND-Verknüpfung	std_logic, std_logic_vector, signed, unsigned	Ja
or	ODER-Verknüpfung		
nand	Nicht-UND-Verknüpfung		
nor	Nicht-ODER-Verknüpfung		
xor	Exklusiv-ODER-Verknüpfung		
not	Invertierung		

Tab. 3.4 Arithmetische VHDL-Operatoren

Schreibweise	Bedeutung	Datentypen	Synthetisierbar?
+	Addition	Numerisch	Ja
-	Subtraktion		
*	Multiplikation		
/	Division		Abhängig vom verwendeten Synthese-Programm
mod	Quotient der Ganzzahldivision		
rem	Rest der Ganzzahldivision		
**	Potenzierung	Integer	Falls Konstanten
abs	Absolutwert	Numerisch	Ja

Tab. 3.5 VHDL-Operatoren für Vergleiche

Schreibweise	Bedeutung	Datentypen	Synthetisierbar?
=	Gleich	Beliebig	Ja
/=	Ungleich		
>	Größer	Numerisch	
<	Kleiner		
>=	Größer-gleich		
<=	Kleiner-gleich		

sollen, würde das Ergebnis in einem Bereich von 0 bis 510 liegen können. Es wären also zur Darstellung des Ergebnisses 9 Bit erforderlich. Dieser Einwand ist völlig korrekt und in VHDL würde das 8 Bit breite Ergebnis der Addition tatsächlich nur die untersten Bits des „wahren" Ergebnisses enthalten. Würden beispielsweise die Zahlen 65 und 250 addiert ($65 + 250 = 315 = 100111011_2$), würde dem Ergebnissignal der binäre Wert 00111011 zugewiesen – die führende 1 ginge verloren. Soll bei der Addition das korrekte 9 Bit breite Ergebnis berechnet werden, muss die Addition mit 9 Bit breiten Operanden ausgeführt werden. Dies lässt sich erreichen, indem die Wortbreite der Operanden um 1 bit vergrößert wird. Eine mögliche Realisierung in VHDL zeigt der nachfolgende Code:

```
-- Addition mit vorheriger Erweiterung der Operanden
sum <= '0' & op1 + '0' & op2;       -- für Datentyp unsigned
sum <= op1(7) & op1 + op2(7) & op2; -- für Datentyp signed
```

Dieser VHDL-Code verwendet den Operator & mit dem zwei Vektoren zu einem neuen Vektor mit größerer Wortbreite „zusammengefügt" werden können. Der Operator lässt sich mit allen vektoriellen Datentypen, also *signed, unsigned* und *std_logic_vector* verwenden. Die folgenden Beispiele verdeutlichen die Funktionsweise des Operators:

```
-- Exemplarische Anwendungen des "Zusammenfügeoperators"
sv6  <= "010" & '1' & "100" & '0'; -- Ergebnis: "01011000"
sv10 <= "00" & sv8;   -- Vorzeichenlose Erweiterung 8 bit -> 10 bit
s9   <= s8(7) & s8;   -- Vorzeichenerweiterung eines signed-Wertes

-- "Rotieren" eines 6 bit breiten Wertes um zwei Stellen nach rechts
-- Beispiel: Aus "011001" wird "010110"
sv6   <= sv6(1 downto 0) & sv6(5 downto 2);
```

3.5 Signale

Die Ausgangswerte komplexerer Schaltungen lassen sich normalerweise nicht durch eine ausschließliche Verknüpfung der Eingangssignale beschreiben. Häufig möchte man zunächst Zwischenergebnisse berechnen, deren anschließende Verknüpfung weitere Zwischenergebnisse oder die Werte der Ausgangssignale ergeben. Diese Zwischenergebnisse sind letztlich nichts anderes als digitale Signale, die nur innerhalb des Moduls verwendet werden und nicht von außen sichtbar sind. Für die Definition solcher Signale steht in VHDL das Schlüsselwort *signal* zur Verfügung.

3.5.1 Definition und Verwendung von Signalen

Nehmen wir an, die in Abb. 3.5 dargestellte Schaltung soll in VHDL beschrieben werden.

Die Eingänge *a* und *b* werden durch ein UND-Gatter zum Signal *z* verknüpft, welches nur innerhalb des Moduls sichtbar ist. Mithilfe der ODER-Verknüpfung von *z* und dem Eingangssignal *c* wird das Ausgangssignal *q* berechnet. Die Signale *a*, *b* und *q* sind Ports der Entity dieses Moduls. Das Signal *z* muss dagegen in der Architecture des Moduls definiert werden. VHDL stellt hierfür das Schlüsselwort *signal* zur Verfügung. Hinter dem Schlüsselwort *signal* werden der gewünschte Signalname sowie der Datentyp des Signals angegeben. Signale werden im sogenannten Deklarationsteil der Architecture definiert, welcher sich vor dem *begin* der Architecture befindet.

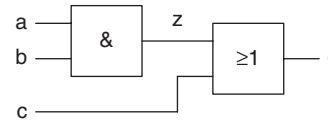


Abb. 3.5 Beispiel einer logischen Funktion

Die VHDL Beschreibung des Moduls würde also wie folgt realisiert werden:

```
library ieee;
use ieee.std_logic_1164.all;

entity and_or is
   port (a : in  std_logic;
         b : in  std_logic;
         c : in  std_logic;
         q : out std_logic);
end;

architecture behave of and_or is
   -- Hier ist der Deklarationsteil der Architecture
   -- Signale werden hier definiert
   -- und können nach "begin" verwendet werden
   signal z : std_logic;
begin
   z <= a and b;
   q <= z or c;
end;
```

3.5.2 Signalzuweisungen

In dem obigen Beispiel ist die Reihenfolge der Zuweisungen an das Signal *z* bzw. den Port *q* unerheblich. Anders als in einer Programmiersprache für die Softwareentwicklung wird der Code innerhalb einer Architecture nicht sequenziell, Zeile für Zeile, ausgeführt, sondern alle Zuweisungen sind zeitgleich aktiv. Der Fachbegriff hierfür ist *nebenläufige Zuweisung*.

Es wäre also ebenso korrekt, den Code wie folgt umzustellen:

```
architecture behave_2 of and_or is
   signal z : std_logic;
begin
   q <= z or c;  -- zuerst die Zuweisung an q
   z <= a and b; -- dann erst an z
end;
```

Hat man bereits Erfahrungen mit Programmiersprachen für die Softwareentwicklung gesammelt, mag dieses Verhalten zunächst ungewöhnlich erscheinen. Aber eine genauere Betrachtung zeigt, dass sich die Zuweisungen genauso verhalten müssen: Ein Gatter in einer digitalen Schaltung reagiert immer auf die Signale an den Gattereingängen, unabhängig davon, ob andere Gatter in der Schaltung existieren oder ob andere Gatter ebenfalls Änderungen ihrer Eingangssignale beobachten. Somit sind die beiden Gatter der Beispielschaltung also immer und unabhängig voneinander aktiv. Das UND-Gatter

wird immer dann einen neuen Wert ausgeben, wenn sich einer der beiden Eingänge *a* oder *b* geändert hat, während eine Änderung von *z* oder *c* zu einer Neuberechnung des Ausgangs *q* führt. Um dieses Verhalten beschreiben zu können, müssen auch die VHDL-Zuweisungen kontinuierlich und unabhängig voneinander aktiv sein. Würde dagegen eine Zuweisung von der Ausführung einer vorangegangenen Zuweisung abhängen, ergäbe sich eine Abhängigkeit, die nicht dem Verhalten der Hardware entspräche.

Selbstverständlich hätte diese recht einfache Schaltung übrigens auch mithilfe einer einzelnen Zuweisung in der Form

```
q <= (a and b) or c;
```

realisiert werden können, wobei dann auf die Definition des Signals *Z* verzichtet werden kann.

In welchem Umfang Signale eingesetzt werden ist auch eine Frage der Übersichtlichkeit des Codes. Werden mehr als zwei oder drei Operatoren in einer Zuweisung verwendet, empfiehlt sich in der Regel der Einsatz von Signalen, um die Lesbarkeit des Codes zu verbessern.

3.6 Prozesse

In den vorangegangenen Beispielen wurden Signalen oder Ports Werte zugewiesen. Hierzu wurden einfache Zuweisungen verwendet. Mithilfe der vorgestellten Operatoren kann man unter Verwendung dieser einfachen Zuweisungen theoretisch beliebig komplexe Schaltungen in VHDL realisieren. Dieses Vorgehen kann allerdings ein recht mühseliges und fehlerbehaftetes Abenteuer werden: Die logische Funktion, die es zu realisieren gilt, müsste zunächst manuell so umgewandelt werden, dass sie mithilfe der vorgestellten Operatoren darstellbar ist. Erst danach kann die Eingabe des VHDL-Codes erfolgen. Selbst wenn die Umwandlung der Funktion fehlerfrei gelingt, wäre der anschließend formulierte VHDL-Code in vielen Fällen schlecht lesbar. Spätere Änderungen der Funktion wären damit schwierig.

Geht es also vielleicht auch etwas eleganter und übersichtlicher? Kann man vielleicht auch in VHDL die aus Programmiersprachen bekannten Konstrukte wie Schleifen oder Verzweigungen zur Beschreibung einer digitalen Funktion verwenden? Alle Signal- oder Portzuweisungen werden zeitgleich (parallel, nebenläufig) ausgeführt. Mit zunehmender Komplexität eines VHDL-Moduls kann dies die Verständlichkeit des Codes weiter verringern. Wäre es daher nicht angenehmer, wenn VHDL-Code sequenziell (wie ein Programm einer Software-Programmiersprache) ausgeführt würde?

Für die Lösung der skizzierten Problematik existiert in VHDL das Sprachkonstrukt eines Prozesses. Prozesse sind eines der wichtigsten Elemente zur Beschreibung von Hardware in VHDL. Ein VHDL-Prozess kann als Erweiterung der Zuweisungen aufgefasst werden. Genauso wie eine nebenläufige Zuweisung beschreibt ein VHDL-Prozess das Verhalten einer Teilschaltung des Systems und wird innerhalb einer Architecture eingesetzt.

Prozesse zeichnen sich unter anderem durch die folgenden Eigenschaften aus:

- Ein Prozess wird nebenläufig zu anderen Prozessen oder Signalzuweisungen ausgeführt.
- VHDL-Code innerhalb eines Prozesses wird sequenziell ausgeführt.
- Innerhalb eines Prozesses können Konstrukte wie sie aus Software-Programmiersprachen bekannt sind, zum Beispiel If-Else-Anweisungen oder Variablen, zur Beschreibung der Funktion des Prozesses eingesetzt werden.
- Genauso wie nebenläufige Signalzuweisungen repräsentiert ein Prozess ein Stück Hardware, welches einen Teil der digitalen Gesamtfunktion des Systems zur Verfügung stellt.

Im Folgenden werden einige wichtige Aspekte von Prozessen näher beleuchtet und vertieft.

3.6.1 Syntaktischer Aufbau von Prozessen

Prozesse werden mithilfe des Schlüsselwortes *process* eingeleitet. Wie bei einer VHDL-Architecture beginnt die eigentliche Beschreibung des Verhaltens nach dem Schlüsselwort *begin*. Zwischen *process* und *begin* befindet sich der Deklarationsteil, welcher zum Beispiel zur Definition von Variablen verwendet werden kann.

Im Gegensatz zu nebenläufigen Signalzuweisungen werden Prozesse nicht automatisch ausgeführt, wenn sich eines der verknüpften Signale ändert. Die im Rahmen dieses Kapitels betrachteten Prozesse besitzen eine sogenannte Sensitivitätsliste, welche die Signale enthält, deren Änderung zu einer Ausführung des Prozesses führen soll. Die Signale werden in Klammern nach dem Schlüsselwort *process* angeben.

Im Folgenden wird die Struktur von Prozessen anhand des Beispiels aus Abb. 3.5 erläutert.

```
architecture and_or_proc of and_or is
begin
   my_process: process (a,b,c)
   begin
      q <= (a and b) or c;
   end process;
end;
```

Es soll eine Schaltung beschrieben werden, welche die Signale bzw. Eingänge a, b und c verknüpft und das Ergebnis q zuweist. Da q von a, b und c abhängt, muss eine Neuberechnung von q immer dann erfolgen, wenn sich eines der Eingangssignale ändert. Daher werden die drei Signale in die Sensitivitätsliste aufgenommen. Zwischen *begin* und *end* wird die Prozessbeschreibung eingefügt, die in diesem einfachen Beispiel nur eine einzelne Zuweisung umfasst.

Es wäre völlig berechtigt, wenn Sie jetzt Zweifel an der Sinnhaftigkeit von Prozessen bekämen: Im Prinzip beschreibt der Prozess keine andere Funktion als die, die man bereits mit einer einfachen Signalzuweisung realisieren kann. Eine nebenläufige Signalzuweisung wäre für dieses Beispiel tatsächlich kürzer und übersichtlicher als die Verwendung eines Prozesses. Aber Prozesse können mehr! Einige Aspekte werden bereits in diesem Kapitel vorgestellt. Andere Aspekte werden Sie beim Lesen der weiteren Kapitel dieses Buches entdecken und sukzessive die Behauptung nachvollziehen können, dass ohne Prozesse eine sinnvolle und übersichtliche Beschreibung digitaler Systeme nicht möglich ist.

Möglicherweise werden Sie bei der Lektüre dieses Buches auch entdecken, dass nebenläufige Signalzuweisungen und Prozesse zwei unterschiedliche Herangehensweisen repräsentieren: Beschreibt man ein digitales Hardware-Modul ausschließlich mit Signalzuweisungen, benötigt man eine gute Vorstellung darüber, wie die Schaltung aus digitalen Grundelementen (UND-, ODER-Gatter, usw.) aufgebaut sein soll. Bei Verwendung von Prozessen steht eher die digitale Funktion im Vordergrund. Wie diese Funktion später durch das Syntheseprogramm mithilfe der verfügbaren Grundelemente realisiert wird, ist von nachrangiger Bedeutung. Daher lassen sich mithilfe von VHDL-Prozessen auch komplexe digitale Funktionen elegant und übersichtlich realisieren.

Der Beispielcode zeigt auch, dass Prozessen Namen erhalten können, wenn dies sinnvoll erscheint. Der Prozessname ist optional und wird vor dem Schlüsselwort *process* eingefügt. Der dem Namen folgende Doppelpunkt ist obligatorisch.

3.6.2 Ausführung von Prozessen

Prozesse besitzen eine gewisse Ähnlichkeit mit Funktionen höherer Programmiersprachen. Allerdings existiert zwischen den Funktionen einer Programmiersprache und den VHDL-Prozessen ein entscheidender Unterschied. Eine Software-Funktion wird vom Programmierer durch einen entsprechenden Aufruf im Code aktiviert und einmalig ausgeführt. Dieses Prinzip kann für Prozesse nicht gelten: Ein Prozess beschreibt eine digitale Hardware-Komponente, die kontinuierlich aktiv ist. Eigentlich müsste also ein Prozess eine Endlosschleife enthalten, die immer wieder den Kern des Prozesses ausführt. Genauso arbeitet ein VHDL-Prozess tatsächlich. Die Endlosschleife ist jedoch im VHDL-Code nicht in Form einer Schleifenanweisung sichtbar, da mit der Verwendung eines VHDL-Prozesses bereits implizit festgelegt ist, dass der Code des Prozesses kontinuierlich ausgeführt wird.

Endlosschleifen in einer Software führen häufig dazu, dass ein Programm nicht mehr reagiert. In VHDL sind dagegen Endlosschleifen bewusst gewollt? Genauso ist es tatsächlich.

Ein Software-Programm wird sequenziell, also Befehl für Befehl, von einem Rechner ausgeführt. Sie haben aber nur einen Rechner zur Ausführung der Software zur Verfügung und wenn dieser mit der Verarbeitung einer Endlosschleife beschäftigt ist, kann er keine anderen Aufgaben ausführen. Wenn Sie dagegen aus einer VHDL-Beschreibung

Hardware generiert haben, existieren sozusagen viele kleine „Rechner“ gleichzeitig. Diese führen kontinuierlich, also im Prinzip in einer Endlosschleife, immer das gleiche „Programm“ aus, welches zuvor durch Prozesse beschrieben wurde.

Aber wie kann dann eine Simulation mehrerer VHDL-Prozesse auf einem nicht-parallelen, sequenziell arbeitenden Rechner ausgeführt werden? Ein PC wäre ja schon mit der Ausführung eines einzelnen VHDL-Prozesses komplett ausgelastet.

Um diese Problematik zu lösen, ist in VHDL die bereits erwähnte Sensitivitätsliste eingeführt worden. In dieser Liste werden alle Signale eingetragen, die innerhalb des jeweiligen Prozesses gelesen werden. Der Prozess wird genau einmal durchlaufen, wenn sich eines der Signale der Sensitivitätsliste ändert. Ändert sich keines der Signale, ruht die Ausführung des jeweiligen Prozesses. Auf diese Weise wird *in der Simulation* einer VHDL-Beschreibung zu einem beliebigen Zeitpunkt immer maximal ein Prozess aktiv sein. Die Aktivierung eines Prozesses führt zu Signaländerungen, die dann wiederum die Ausführung weiterer Prozesse zur Folge haben. Auf diese Weise kann sukzessive das gesamte Verhalten der parallelen Hardware auf einem sequenziell arbeitenden PC nachgebildet werden.

Wird beim Anlegen der Sensitivitätsliste ein Signal übersehen, ist dies für die Hardwaregenerierung mittels Synthese relativ unbedeutend. Die meisten Syntheseprogramme würden zwar Warnungen ausgeben, aber dennoch eine funktionstüchtige Hardware erzeugen.

Für die Simulation ist die korrekte Angabe der Sensitivitätsliste dagegen sehr wichtig: Würde bei dem in Abschn. 3.6.1 gezeigten Beispiel das Signal *b* nicht in der Sensitivitätsliste aufgeführt sein, würde der Prozess bei Änderungen von *b* nicht aktiviert werden. Somit würde trotz einer Änderung von *b* das Ausgangssignal *q* seinen Wert behalten und die Simulation der Schaltung ein anderes Ergebnis liefern als die zugehörige Hardware. Eine umfassende Überprüfung der VHDL-Beschreibung mithilfe einer Simulation wäre also nicht möglich.

3.6.3 Variablen

Als Alternative zu Signalen können in Prozessen auch Variablen eingesetzt werden. VHDL-Variablen sind mit statischen Variablen vergleichbar, wie sie zum Beispiel in der Programmiersprache C zur Verfügung stehen: Sie sind nur in dem Prozess sichtbar, in dem sie definiert wurden und behalten den zugewiesenen Wert auch dann, wenn der Prozess unterbrochen wird.

Die Definition einer Variablen geschieht im Deklarationsteil des Prozesses (vor *begin*) und werden mit dem Schlüsselwort *variable* eingeleitet. Für Zuweisungen an Variablen wird *:=* verwendet, während bei Signalen die bereits erwähnte Zeichenkombination <= zum Einsatz kommt.

Ein einfaches Beispiel verdeutlicht die Verwendung von Variablen in VHDL-Prozessen:

```
proc_with_variable : process (a,b)
   variable my_var : std_logic;
begin
   my_var := a and b; -- Variablenzuweisung
   q <= my_var;       -- Signalzuweisung
end process;
```

Aufgrund der sequenziellen Ausführung eines Prozesses sind die im Beispielcode gezeigten Zuweisungen nicht vertauschbar.

Im Prinzip wird eine Zuweisung an eine Variable zunächst komplett durchgeführt, bevor die nächste Zeile des Prozesses abgearbeitet wird. Dies ist genau das Verhalten, das auch für Variablen in Programmiersprachen wie C/C++ oder Java gilt.

Zuweisungen an Signale blockieren den Prozessablauf dagegen nicht. Der Prozess läuft also weiter, ohne dass die Zuweisung eine Wirkung auf den Wert des Signals hat. Das Signal behält bis zu einer Prozessunterbrechung bzw. dem Prozessende seinen alten Wert. Erst bei einer Beendigung oder Unterbrechung des Prozesses werden die zuvor ausgeführten Signalzuweisungen wirksam und die Signale erhalten neue Werte.

Würde also die Zuweisung an *q* vor der Zuweisung an *my_var* stehen, würde der VHDL-Code im Gegensatz zum obigen Beispiel kein einfaches UND-Gatter mehr beschreiben.

Da insbesondere das oben erwähnte Verhalten von Signalzuweisungen innerhalb von Prozessen für viele VHDL-Einsteiger etwas gewöhnungsbedürftig ist, wird dieses Verhalten im nachfolgenden Abschnitt ausführlicher erläutert.

3.6.4 Signalzuweisungen in Prozessen

Für die Zuweisungen von Signalen innerhalb von VHDL-Prozessen gelten zwei wichtige Regeln:

1. Die an ein Signal zugewiesenen Werte werden erst nach einer Unterbrechung des Prozesses sichtbar.
2. Wird ein Signal mehrfach in einem Prozess zugewiesen, zeigt nur die zuletzt ausgeführte Zuweisung Wirkung. Alle vorangegangenen Zuweisungen werden verworfen.

Da Signale allen Prozessen einer VHDL-Architecture zur Verfügung stehen, muss sichergestellt werden, dass die Änderung eines Signals in allen Prozessen gleichzeitig sichtbar wird. Dieser Forderung wird durch die erste Regel Rechnung getragen.

Wird eine VHDL-Beschreibung simuliert, werden alle Zuweisungen an Signale zunächst „gesammelt“. Die eigentliche Zuweisung an das Signal und die damit verbundene Sichtbarmachung eines Signalwechsels geschieht erst mit der Unterbrechung des Prozesses oder mit der Beendigung des Prozessdurchlaufs. Dies bedeutet auch, dass

ein Lesezugriff auf ein Signal vor einer Prozessunterbrechung den „alten“ Wert liefern wird – unabhängig davon, ob der Prozess das Signal zuvor beschrieben hat oder nicht.

Nicht wenige, die VHDL lernen, haben zuvor eine Programmiersprache erlernt. In diesen Sprachen gilt die Regel, dass eine Zuweisung sofort Wirkung zeigt. Wird einer Variablen ein neuer Wert zugewiesen, kann bereits mit dem nächsten Befehl auf den neuen Wert zugegriffen werden. Auch hier gilt: VHDL hat zwar viele Ähnlichkeiten mit klassischen Programmiersprachen, aber VHDL ist nicht für die Entwicklung eines sequenziellen Rechnerprogramms, sondern für die Beschreibung von parallel arbeitenden Hardwarekomponenten gedacht.

Die zweite Regel ergibt sich als Konsequenz aus der ersten. Es ist erlaubt einem Signal in einem Prozess mehrfach einen Wert zuzuweisen. Wenn die Signalzuweisungen aber zunächst gesammelt werden und erst bei einer Prozessunterbrechung wirklich ausgeführt werden, kann hierbei nur der zuletzt zugewiesene Wert Berücksichtigung finden.

Ihnen wird der nachfolgende VHDL-Code vorgelegt. Es handelt sich um ein Modul mit den Eingängen *a* und *b* sowie dem Ausgang *q*.

```
signal s : std_logic;
   -- Hier ggf. weiterer Code
process (a,b,s)
begin
   s <= a and b;
   s <= a or b;
   s <= a;
   q <= s;
   s <= a xor b;
end process;
```

Welche Hardware wird durch diesen Code beschrieben? Ein UND-Gatter oder ein ODER-Gatter? Oder ist es nur ein einfacher Draht; wird also *q* immer direkt der Wert von *a* zugewiesen? Oder handelt es sich um ein Exklusiv-ODER-Gatter?

Analysiert man dieses Beispiel Schritt für Schritt, kann man sich der, in diesem Beispiel recht verklausulierten, Funktion des Codes nähern.

Offensichtlich ist, dass die ersten beiden Zuweisungen an das Signal *s* keine Wirkung haben, da sie durch spätere Zuweisungen überschrieben werden. Diese kann man also aus dem Code streichen und der Prozess kann auch wie folgt formuliert werden.

```
process (a,b,s)
begin
   s <= a;
   q <= s;
   s <= a xor b;
end process;
```

Werfen wir in dem verbleibenden Code einen Blick auf die Zuweisung an den Ausgang *q*. *q* wird der Wert von *s* zugewiesen. Aber was liefert der Lesezugriff auf *s* zurück? Vielleicht sind Sie geneigt ad hoc „*a*“ zu sagen, da vor der Zuweisung an *q* dem Signal *s* der Wert von *a* zugewiesen wird. Dies wäre die korrekte Antwort, wenn es sich bei *s* um eine VHDL-Variable handeln würde. Da *s* jedoch ein Signal ist, muss der Code noch etwas genauer analysiert werden.

Bei der Ausführung des Prozesses wird die Zuweisung des Wertes von *a* an das Signal *s* noch nicht sofort ausgeführt. Die Zuweisung an *q* würde also den Wert des Signals *s* sehen, der bei einem vorangegangenen Aufruf des Prozesses zugewiesen wurde. Da das Signal *s* in dem Prozess zweimal geschrieben wird und nur die letzte Signalzuweisung zur Ausführung kommt, wird *s* also die Exklusiv-ODER-Verknüpfung der Eingänge *a* und *b* zugewiesen. Also beschreibt der Prozess letztlich eine Exklusiv-ODER-Verknüpfung.

Die Reihenfolge der Zuweisungen an *s* und *q* ist, wie bei nebenläufigen Signalzuweisungen irrelevant. Der Prozess kann daher auch wie folgt formuliert werden.

```
process (a,b,s)
begin
   s <= a xor b;
   q <= s;
end process;
```

Diese Variante ist deutlich besser lesbar, da sie auch bei einer sequenziellen Interpretation des Codes auf das korrekte Verständnis der beschriebenen Funktionalität führt.

Sofern das Signal *s* nicht in anderen Prozessen der Architecture verwendet wird, kann der Code auf die Zuweisung *q* <= *a xor b* reduziert werden.

Ein nicht seltener Fehler, der bei Signalzuweisungen in Prozessen auftritt, ist die Zuweisung eines Signals aus unterschiedlichen Prozessen heraus. Dies würde bedeuten, dass zwei Prozesse gleichzeitig den Wert des Signals festlegen könnten. Abgesehen von wenigen Spezialfällen, ist dies in der Regel nicht gewollt und würde auch beim Synthesevorgang zu Fehlermeldungen führen. Daher müssen bei der Erstellung von VHDL-Prozessen die beiden folgenden Regeln beachtet werden:

1. Signale dürfen in **beliebig vielen** Prozessen **gelesen** werden.
2. Signale dürfen nur in **einem** Prozess **geschrieben** werden.

3.6.5 Wichtige Sprachkonstrukte in VHDL-Prozessen

VHDL-Prozesse bieten vielfältige Sprachkonstrukte zur Beschreibung einer Hardware-Komponente. In diesem Abschnitt werden die gebräuchlichsten und wichtigsten Elemente zur Beschreibung von Prozessen vorgestellt.

3.6.5.1 If-Anweisung

Die If-Anweisung ermöglicht die bedingte Ausführung von Code innerhalb eines VHDL-Prozesses. Zwischen den Schlüsselwörtern *if* und *then* wird eine Bedingung, beispielsweise ein Vergleich zweier Signale, eingefügt. Anschließend folgt der Code, der ausgeführt werden soll, wenn die Bedingung wahr ist. Abgeschlossen wird die Anweisung mit *end if;*

Optional können zusätzlich mit *elsif* weitere Bedingungen eingefügt werden, die dann überprüft werden, wenn die voranstehenden Bedingungen unwahr waren.

Mit dem Schlüsselwort *else* wird der Code eingeleitet, der ausgeführt werden soll, wenn alle Bedingungen der If-Anweisung unwahr waren. Auch dies ist eine Option, die bei Bedarf weggelassen werden kann.

Bei der Verwendung von *elsif* ist die Schreibweise als ein einzelnes Wort zu beachten. Viele VHDL-Anfänger, insbesondere wenn sie bereits Programmierkenntnisse besitzen, neigen dazu, statt *elsif* die Formulierung *else if* zu wählen. Die beiden Varianten sind nicht äquivalent. Mit *else if* wird in dem Else-Zweig der Anweisung eine neue If-Anweisung geöffnet, die ihrerseits durch *end if* geschlossen werden muss.

Der folgende Pseudocode zeigt den prinzipiellen Aufbau der If-Anweisung, wobei optionale Elemente in geschweiften Klammern dargestellt sind.

```
if <Bedingung> then
   <Anweisungen>
{elsif <Bedingung> then
   <Anweisungen>}
{else
   <Anweisungen>}
end if;
```

Ein Beispiel für die Anwendung der If-Anweisung zeigt der folgende Code.

```
if a = b then
   q <= a and c;
   v := '1';
elsif a = c and b = '1' then
   q <= d;
   v := '1';
else
   q <= '0';
   v := '0';
end if;
```

3.6.5.2 Case-Anweisung

Wie die If-Anweisung ermöglicht auch die Case-Anweisung die bedingte Ausführung von Codeteilen. Nach dem Schlüsselwort *case* wird ein auszuwertender Ausdruck

angegeben. Mit dem Schlüsselwort *when* wird angegeben, welcher Code für ein konkretes Ergebnis des Ausdrucks ausgeführt werden soll. Durch die Verwendung von „|“ können mehrere Werte angegeben werden, die zur Ausführung des nachfolgenden Codes führen sollen. Ist keiner der angegebenen Werte identisch mit dem Ergebnis des Ausdrucks, können Default-Anweisungen spezifiziert werden, die in diesem Fall ausgeführt werden sollen. Hierzu wird statt eines Wertes das Schlüsselwort *others* angegeben.

Der folgende Pseudocode zeigt den Aufbau der Case-When-Anweisung.

```
case <Ausdruck> is
   when <Wert>  => <Anweisungen>
   {when <Wert> => <Anweisungen>}
      …
   {when <Wert> => <Anweisungen>}
   {when others => <Anweisungen>}
end case;
```

Ein Anwendungsbeispiel der Case-When-Anweisung wird durch den folgenden Code dargestellt.

```
case a_vec is -- a_vec ist vom Typ std_logic_vector(2 downto 0)
   when "000" =>
      q <= a and c;
      r <= a;
   when "001"|"010" =>
      q <= b;
      r <= a and c;
   when "111" =>
      q <= '1';
      r <= d;
   when others =>
      q <= '0';
      r <= '0';
end case;
```

Mit einer Case-Anweisung kann ein einzelner Ausdruck mit verschiedenen möglichen (konstanten) Werten verglichen werden. In vielen Fällen kann mit der Case-Anweisung ein sehr kompakter und übersichtlicher Code erzielt werden. Sind die Vergleichswerte nicht konstant oder sind Vergleiche mit unterschiedlichen Ausdrücken gewünscht, kann die If-Anweisung verwendet werden.

3.6.5.3 For-Schleife

VHDL unterstützt auch Schleifen. Zuerst wird hier die For-Schleife vorgestellt.

Nach dem Schlüsselwort *for* wird ein Bezeichner für die Schleifenvariable eingefügt. Der Schleifenbereich folgt nach dem Schlüsselwort *in*. Der Bereich kann aufsteigend (zum Beispiel *„1 to 8“*) oder absteigend (zum Beispiel *„15 downto 0“*) durchlaufen werden.

Nach der Angabe des Schleifenbereichs folgt das Schlüsselwort *loop*, welches von den Anweisungen des Schleifenkerns gefolgt wird. Die Schleife wird mit *end loop* abgeschlossen.

Schleifen dürfen optional mit einem Namen *(Label)* versehen werden.

```
{loop_label:} for <Bezeichner> in <Bereich> loop
   <Anweisungen>
end loop {loop_label};
```

Ein Beispiel für die Verwendung einer For-Schleife zeigt das nachfolgende Codefragment, das den Vektor x „spiegelt“ und das Ergebnis dem Vektor y zuweist. *y(0)* erhält den Wert von *x(9), y(1)* den Wert von *x(8)* usw.

```
my_loop: for i in 0 to 9 loop
   y(9-i) := x(i); -- x und y sind Vektoren
end loop my_loop;
```

Die For-Schleifen sind abweisende Schleifen. Beispielsweise würde der Kern der nachfolgenden Schleife nie ausgeführt werden, da es sich um eine abwärtszählende Schleife handelt, deren untere Grenze größer ist als die obere.

```
another_loop: for i in 0 downto 5 loop
   y(i) := x(i); -- was auch immer hier steht - es wird nicht ausgeführt!
end loop;
```

Schleifen sind synthesefähig, wenn die Schleifengrenzen statisch sind, sich die Schleifengrenzen also nicht erst zur Laufzeit des VHDL-Codes ergeben.

Darüber hinaus ist zu beachten, dass Schleifen von Syntheseprogrammen „ausgerollt“ werden. Man kann sich dies so vorstellen, dass die Schleife aufgelöst wird und der Schleifenkern wiederholt in den Code eingefügt wird. Für jedes Durchlaufen des Schleifenkerns wird also eine eigene Hardwarekomponente generiert.

3.6.5.4 While-Schleife

Neben For-Schleifen können in VHDL auch While-Schleifen eingesetzt werden. Hierbei wird zunächst die nach dem Schlüsselwort *while* angegebene Bedingung geprüft. Ergibt diese den Wert *true,* wird der Schleifenkern ausgeführt und anschließend die Bedingung erneut geprüft. Auch die While-Schleifen sind also abweisende Schleifen.

Die Struktur einer while-Schleife zeigt der folgende Pseudocode.

```
{loop_label:} while <Bedingung> loop
   <Anweisungen>
end loop {loop_label};
```

Ein Beispiel für die Verwendung einer While-Schleife zeigt das nachfolgende Codefragment.

```
i := 0;
while i < 8 loop
   a(i) := b(i) xor c(7-i);
   i    := i + 1;
end loop;
```

3.7 Hierarchie

Werden komplexere Schaltungen entworfen, ist es sinnvoll, die gesamte Schaltung in kleinere Module aufzuspalten, die zunächst separat in VHDL beschrieben werden. In einer weiteren VHDL-Beschreibung können diese Module dann zur gewünschten Gesamtschaltung kombiniert werden. Um dieses Vorgehen zu unterstützen, bietet VHDL die Möglichkeit Module innerhalb von Modulen „aufzurufen“. In der Praxis spricht man hierbei nicht von „aufrufen“, sondern von „instanziieren“. Ein instanziiertes Modul wird auch als „Instanz“ dieses Moduls bezeichnet.

Es ist auch möglich eine neu geschaffene Komponente, welche Instanzen enthält, wiederum in einem anderen Modul zu instanziieren und so eine hierarchische Beschreibung einer Schaltung in mehreren Stufen/Ebenen zu realisieren.

Im Folgenden wird die Vorgehensweise zur Instanziierung von Modulen in VHDL anhand des Beispiels einer Komponente beschrieben, die drei 8-Bit-Operanden addiert.

Nehmen wir an, dass bereits das folgende Entity-Architecture-Paar zur Beschreibung eines 8-Bit-Addierers für zwei Operanden in VHDL beschrieben wurde.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity add_2 is
   port (op1 : in  std_logic_vector(7 downto 0);
         op2 : in  std_logic_vector(7 downto 0);
         sum : out std_logic_vector(7 downto 0));
end;
```

```
architecture struct of add_2 is
begin
   process (op1,op2)
   begin
      sum <= std_logic_vector(unsigned(op1) + unsigned(op2));
   end process;
end;
```

Um diese Beschreibung des Addierers in einer anderen VHDL-Architecture zu instanziieren, wird die Entity angegeben, die für diese Instanziierung verwendet werden soll. Darüber hinaus muss die Bibliothek angegeben werden, in der das Modul abgelegt wurde.

Die Instanziierung eines Moduls beginnt mit einem eindeutigen Namen für diese Instanz. Nach einem Doppelpunkt folgen das Schlüsselwort *entity,* die Bibliothek (im nachfolgenden Beispiel die Arbeitsbibliothek *work*) und der Name des zu instanziierenden Moduls. Abschließend wird die Zuordnung der Anschlüsse der Instanz zu den Ein- und Ausgängen oder den Signalen der instanziierenden Architecture angegeben. Die Zuordnung wird mit den Schlüsselwörtern *port map* eingeleitet.

Auf Basis des Addierers für zwei Operanden kann ein Addierer für 3 Operanden realisiert werden. Das Blockschaltbild dieses 3-Operanden-Addierers ist in Abb. 3.6 dargestellt.

Dieser Addierer lässt sich in VHDL wie folgt beschreiben:

```
library ieee;
use ieee.std_logic_1164.all;

entity add_3 is
   port (a : in std_logic_vector (7 downto 0);
         b : in std_logic_vector (7 downto 0);
```

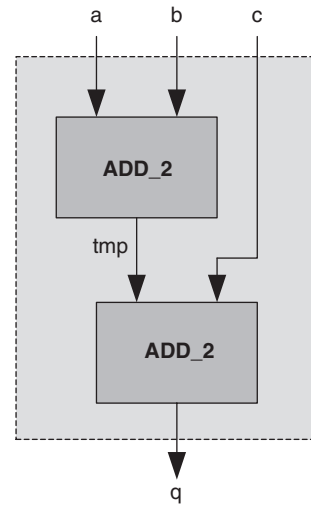


Abb. 3.6 Blockschaltbild eines Addierers für 3 Operanden

```
        c : in  std_logic_vector (7 downto 0);
        q : out std_logic_vector (7 downto 0));
end;

architecture struct of add_3 is
   signal tmp : std_logic_vector (7 downto 0);
begin
   a1: entity work.add_2 port map (op1 => a, op2 => b, sum => tmp);
   a2: entity work.add_2 port map (op1 => tmp, op2 => c, sum => q);
end;
```

Das Beispiel zeigt die Zuordnung der Anschlüsse der *add_2*-Module zu den Ein- und Ausgängen des Moduls *add_3,* wobei eine namensbasierte Zuordnung (engl. *named association*) mithilfe des Zuordnungsoperators => verwendet wird. Eher selten findet man positionsbasierte Zuordnung (engl. *positional association*), bei der lediglich die Ports und Signale der instanziierenden Architecture angegeben werden. Das erste angegebene Signal wird dann an den ersten Port der instanziierten Architecture angeschlossen. Das zweite Signal an den zweiten Port, usw.

3.8 Übungsaufgaben

Haben Sie den Inhalt des Kapitels verstanden? Prüfen Sie sich mit den folgenden Aufgaben. Die Antworten finden Sie am Ende des Buches.

Sofern nicht anders vermerkt, ist nur eine Antwort richtig.

Aufgabe 3.1
Welche der folgenden Aussagen zum VHDL-basierten Entwurfsprozess ist richtig?

a) Eine Testbench ist eine VHDL-Datei, die nur in der Simulation zum Einsatz kommt.
b) Wurde mithilfe von Simulationen die korrekte Funktionsweise einer VHDL-Beschreibung nachgewiesen, müssen im weiteren Verlauf des Entwurfsprozesses keine Änderungen an dem VHDL-Code vorgenommen werden.
c) Ein digitales System muss immer in einer einzelnen VHDL-Datei beschrieben werden.
d) Eine syntaktisch korrekt beschriebenes Entity-/Architecture-Paar ist sowohl simulierbar als auch synthetisierbar.

Aufgabe 3.2
Welche Aussagen zu VHDL-Bibliotheken sind richtig? *(Mehrere Antworten sind richtig)*

a) Das Ergebnis der Übersetzung einer VHDL-Datei wird immer in einer Bibliothek abgelegt.

b) Zur Verwendung der Inhalte einer Bibliothek muss diese mithilfe einer Library-Anweisung bekannt gemacht werden (Ausnahmen *work*, *std*).
c) Die Bibliothek *work* enthält wichtige vordefinierte Datentypen.
d) Bei der Verwendung des Datentyps *std_logic* muss die Bibilothek *ieee* bekannt gemacht werden.
e) Die Datentypen *signed, unsigned* und *integer* sind vordefinierte Datentypen, die auch ohne Angabe einer Bibliothek verwendet werden können.

Aufgabe 3.3
Welche Aussagen sind richtig? *(Mehrere Antworten sind richtig)*

a) In VHDL wird Groß- und Kleinschreibung nicht unterschieden: *MY_SIG* und *my_sig* bezeichnen das gleiche Signal.
b) Anhand der Entity einer VHDL-Beschreibung können die Ein- und Ausgänge eines Moduls identifiziert werden.
c) Signale vom Datentyp std_logic können nur die Werte ‚0', ‚1' und ‚U' annehmen.
d) Im Deklarationsteil einer Architecture (= vor *begin*) können sowohl Signale als auch Variablen definiert werden.
e) Numerische Konstanten können nicht in hexadezimaler Darstellung angegeben werden.

Aufgabe 3.4
Welche Aussagen zu VHDL-Prozessen sind richtig? *(Mehrere Antworten sind richtig)*

a) Der Code innerhalb eines Prozesses wird sequenziell ausgeführt.
b) Alle Signale auf die innerhalb eines Prozesses schreibend zugegriffen wird, müssen in der Sensitivitätsliste erscheinen.
c) Innerhalb eines Prozesses ist nur die zuletzt ausgeführte Zuweisung an ein Signal relevant. Alle vorangegangenen Zuweisungen an das gleiche Signal haben keine Wirkung.
d) Für die Zuweisung eines Wertes an eine Variable wird die Zeichenkombination „<=" verwendet.

Aufgabe 3.5
Der nachfolgend dargestellte VHDL-Code ist syntaktisch nicht korrekt. Korrigieren Sie die Fehler.

```
library ieee.std_logic_1164.all;

entity my_module is
   port (a : in  std_logic_vector;
         b : in  integer;
         c : in  std_logic;
```

```
        q : out std_logic_vector;)
end;

architecture of my_module is
begin
   signal tmp : unsigned (7 downto 0);
   process (a,b,tmp)
      variable vi : unsigned (7 downto 0);
   begin
      tmp <= to_unsigned(A);
      vi  <= to_unsigned(B,8);
      if c == 1
         q <= vi - tmp;
      else
         q <= vi + tmp;
      end;
   end process;
end;
```

Aufgabe 3.6
Erstellen Sie ein VHDL-Modul (Entity und Architecture), das die im Folgenden beschriebene Funktion realisiert:

- Das Modul besitzt die Eingänge a (Wortbreite 8 bit), b (8 Bit) und c (2 Bit) und den Ausgang q (8 Bit)
- Der Ausgang q wird in Abhängigkeit vom Eingang c aus den Werten der Eingänge a und b berechnet. Es soll gelten:
  $c = 00$: $q = a$
  $c = 01$: $q = a \,\&\, b$
  $c = 10$: $q = a \vee b$
  $c = 11$: $q = a \oplus b$ ($\oplus$ bezeichnet eine Exklusiv-ODER-Verknüpfung)
- Verwenden Sie für die Fallunterscheidung (Werte von c) eine If-Anweisung

Aufgabe 3.7
Ersetzen Sie die If-Anweisung aus Aufgabe 3.6 durch eine Case-Anweisung. Welche Codeänderungen sind erforderlich?

Aufgabe 3.8
Auf Basis des Moduls aus Aufgabe 3.6 soll ein Modul entworfen werden, das für eine Wortbreite von 16 Bit ausgelegt ist (Ports *a,b* und q) aber ansonsten die identische Funktion ausführt.

Schreiben Sie ein geeignetes Entity/Architecture-Paar in VHDL. Instanziieren Sie das Modul aus Aufgabe 3.6.

Kombinatorische Schaltungen

4

Digitalschaltungen, deren Ausgänge nur von den aktuellen Eingangswerten abhängen, nennt man *kombinatorische Schaltungen*. Eine solche Schaltung arbeitet nur mit Logikgattern und enthält weder Rückkopplungen noch Flip-Flops. Die Eingangswerte werden durch die Schaltung kombiniert (daher der Name) und ein Ergebnis berechnet. Da keine Flip-Flops verwendet werden, können keine Informationen gespeichert werden.

Kombinatorische Schaltungen sind normalerweise Teil einer größeren Schaltung. Sie werden zusammen mit Flip-Flops eingesetzt, wobei der kombinatorische Teil die Berechnungen vornimmt und die Flip-Flops die Ergebnisse speichern. Die gesamte Schaltung mit den Flip-Flops ist dann eine sequenzielle Schaltung, also eine Schaltung deren Ergebnis von der zeitlichen Abfolge (der Sequenz) ihrer Eingänge abhängt. In diesem Kapitel werden zunächst die Funktion und der Entwurf kombinatorischer Schaltungen erläutert. Flip-Flops und sequenzielle Schaltungen werden im nächsten Kapitel vorgestellt.

Als Beispiel für eine kombinatorische Schaltung ist in Abb. 4.1 eine einfache Alarmanlage dargestellt. Dabei sollen eine Tür *(T)* und zwei Fenster *(F1, F2)* überwacht werden. Mit einem Schalter *(S)* wird die Alarmanlage ein- oder ausgeschaltet. Diese vier Eingangssignale sollen binäre Werte also 0 oder 1 sein. Die 1 bedeutet dabei jeweils „aktiv", das heißt Tür oder Fenster ist offen, beziehungsweise Anlage ist eingeschaltet.

Die kombinatorische Schaltung wertet die vier Eingangssignale aus und berechnet, ob ein Alarm ausgelöst werden soll oder nicht. Dafür gibt es einen Ausgang *A,* der mit einer 1 einen Alarmfall anzeigt. Andernfalls ist der Ausgang 0. Am Ausgang *A* ist eine Alarmhupe angebracht.

Wie man systematisch die kombinatorische Schaltung entwirft, wird später in diesem Kapitel erläutert. Für dieses einfache Beispiel kann man die Schaltung direkt aus der Aufgabenstellung ableiten. Der Alarm soll überwachen, ob Tür oder Fenster offen sind und dabei melden, wenn einer oder mehrere der Kontakte auf 1 sind. Dies entspricht der ODER-Verknüpfung der drei Signale *T, F1, F2*. Dieses Zwischenergebnis führt zu einem Alarm, wenn die Anlage eingeschaltet ist, also muss das Ergebnis der ODER-Verknüpfung

W. Gehrke et al., *Digitaltechnik,* Springer-Lehrbuch,
DOI 10.1007/978-3-662-49731-9_4

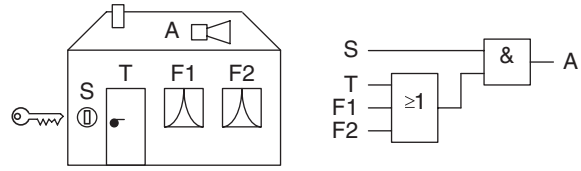


Abb. 4.1 Kombinatorische Schaltung als einfache Alarmanlage

noch mit dem Schalter *S* UND-verknüpft werden. Nur wenn der Schalter auf 1 ist, wird der Alarm *A* ausgelöst. Die kombinatorische Schaltung ist ebenfalls in Abb. 4.1 dargestellt.

4.1 Schaltalgebra

Die Rechenregeln der Digitaltechnik werden als *Schaltalgebra* bezeichnet. Der Begriff Algebra ist aus der Schulmathematik bekannt und beschreibt dort die Rechenregeln für Zahlen. Die Zahlen in der elementaren Algebra, also der Schulmathematik, können dabei unendlich viele Werte einnehmen, also eins, zwei, drei, siebenundvierzig, fünftausend und so weiter.

Die Schaltalgebra ist eine besondere Form der Algebra, bei der Variablen nur zwei mögliche Werte haben, nämlich 0 und 1. Das heißt für alle Eingangswerte und das Ergebnis einer Rechenoperation sind nur diese beiden Werte möglich. Manchmal werden für die Werte auch die Begriffe *Falsch* (entspricht 0) und *Wahr* (entspricht 1) verwendet.

Funktionen, bei denen Eingangs- und Ausgangswerte nur die Werte 0 und 1 annehmen können, bezeichnet man als *binäre Schaltfunktionen, boolesche Schaltfunktionen* oder einfach *Schaltfunktionen*. Die Bezeichnung boolesch weist darauf hin, dass die Funktion nach der Booleschen Algebra berechnet wird, die nach dem englischen Mathematiker George Boole benannt ist.

Die Schaltalgebra ist also die mathematische Beschreibung der Funktionen in der Digitaltechnik. Die Schaltung selber wird dann als kombinatorische Schaltung bezeichnet. Darin führen *Schaltglieder* eine logische Verknüpfung von Eingangswerten zu einem Ausgangswert durch. Die Schaltglieder bezeichnet man auch als *Gatter*.

Physikalische Eigenschaften wie Spannungspegel oder Umschaltzeiten werden in der Schaltalgebra nicht berücksichtigt. Ob ein digitales Signal den Wert 0 V oder 0,1 V hat ist unbedeutend. Beide Spannungspegel werden durch den Wert 0 dargestellt. Somit ist die Schaltalgebra eine Abstrahierung zur vereinfachten Schaltungsbeschreibung.

4.1.1 Schaltfunktion und Schaltzeichen

Bei der Beschreibung von *Schaltfunktionen* werden die Eingangsvariablen meist mit den Buchstaben *A, B, C,* … und die Ausgangsvariable mit dem Buchstaben *Y* bezeichnet. *Y* ist damit eine Funktion von *A, B, C,* … und kann durch ein *Schaltzeichen* dargestellt werden (Abb. 4.2).

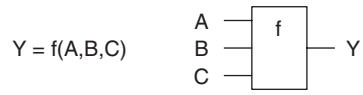


Abb. 4.2 Schaltfunktion und Schaltzeichen

4.1.2 Funktionstabelle

Da jede Eingangsvariable nur zwei mögliche Werte haben kann, ist es möglich, sämtliche Kombinationen der Eingangswerte aufzuzählen und als Funktionstabelle anzugeben. Bei n Eingängen sind 2^n Kombinationen möglich. Für die *Funktionstabelle* wird auch der Begriff *Wahrheitstabelle* benutzt; er bezieht sich auf die Bezeichnungen *Falsch* und *Wahr*.

Somit gibt es bei zwei Eingangsvariablen A und B vier verschiedene Kombinationen der Eingangswerte, nämlich 00, 01, 10, 11. Drei Eingangsvariablen ergeben acht, vier Eingangsvariablen 16 Kombinationen.

Für die elementare Algebra wäre eine Funktionstabelle nicht möglich, da unendlich viele Eingangswerte möglich sind. Die Tabelle würde also unendlich groß werden. Trotzdem gibt es auch dort ein Beispiel für eine Funktionstabelle, nämlich das „Kleine Einmaleins“. Für das Produkt zweier Zahlen von 1 bis 10 gibt es 100 Möglichkeiten und die 100 Ergebnisse werden in der Grundschule auswendig gelernt.

Funktionstabellen dienen zum Beschreiben vorhandener Schaltungen oder zur Spezifikation einer Schaltung, die entworfen werden soll. Beim Schaltungsentwurf, der Schaltungssynthese wird die Aufgabe meist als Text beschrieben und daraus die Funktionstabelle erstellt.

Als Beispiel soll eine Schaltung spezifiziert werden, welche die Mehrheit aus drei Eingangswerten bildet. Die Eingänge *A, B, C* sind digitale Werte und können die Werte 0 und 1 annehmen. Wenn zwei oder drei Eingänge 1 sind, soll auch der Ausgang Y 1 sein. Ansonsten ist der Ausgang 0.

Eine solche Mehrheitsschaltung oder *Majoritätsschaltung* kann als Sicherheitsschaltung für redundante Systeme dienen. Eine Fabrikhalle hat drei Rauchmelder und nur wenn zwei Rauchmelder auslösen, wird ein Alarm gemeldet und die Fabrikation gestoppt. Ein Fehler in einem Rauchmelder kann also keinen Alarm auslösen.

Die Funktionstabelle der Majoritätsschaltung ist in Abb. 4.3 angegeben. Für drei Variablen gibt es 2^3, also 8 Kombinationen und die Tabelle gibt an, welche der Kombinationen eine 1 am Ausgang ergeben sollen.

4.1.3 Funktionstabelle mit Don’t-Care

Als Besonderheit kann es bei Funktionstabellen vorkommen, dass für eine oder mehrere Eingangskombinationen keine Ausgabe spezifiziert werden muss. Dies ist dann der Fall, wenn bestimmte Eingangskombinationen laut Aufgabenstellung nicht vorkommen

Abb. 4.3 Funktionstabelle der Majoritätsschaltung

A	B	C	Y
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Abb. 4.4 Primzahlerkennung für Zahlen 0 bis 9 als Funktionstabelle mit Don't-Care

A(3:0)				Y	*Zahlenwert*
0	0	0	0	0	0
0	0	0	1	0	1
0	0	1	0	1	2
0	0	1	1	1	3
0	1	0	0	0	4
0	1	0	1	1	5
0	1	1	0	0	6
0	1	1	1	1	7
1	0	0	0	0	8
1	0	0	1	0	9
1	0	1	0	-	
1	0	1	1	-	
1	1	0	0	-	
1	1	0	1	-	
1	1	1	0	-	
1	1	1	1	-	

können. Oder das Ergebnis bestimmter Eingangskombinationen wird in der späteren Verarbeitung nicht verwendet.

Der nicht definierte Ausgang wird als *Don't-Care* bezeichnet und in der Funktionstabelle mit einem Strich ‚-' gekennzeichnet. Beim Schaltungsentwurf können die Don't-Care-Einträge benutzt werden, um eine möglichst kleine und damit kostengünstige Schaltung zu entwerfen.

Eine Schaltung soll für die Zahlen 0 bis 9 ausgeben, ob es sich um eine Primzahl handelt. Die Zahlen sind als vierstellige Dualzahl *A(3:0)* angegeben. Von den 16 Kombinationen der vier Stellen werden 6 Kombinationen nicht benutzt. Die Ausgabe für diese Kombinationen ist beliebig, also Don't-Care. Abb. 4.4 zeigt die Funktionstabelle.

4.2 Funktionen der Schaltalgebra

Die Grundfunktionen der Schaltalgebra sind UND-Verknüpfung, ODER-Verknüpfung und Negation. Alle anderen Schaltfunktionen lassen sich aus Kombinationen dieser Grundfunktionen darstellen. Zusammengesetzte Funktionen sind NAND-Verknüpfung, NOR-Verknüpfung, XOR-Verknüpfung (Antivalenz) und XNOR-Verknüpfung (Äquivalenz).

4.2.1 UND-Verknüpfung

Die *UND-Verknüpfung* wurde in Kapitel 1 schon kurz vorgestellt. Der Ausgang Y ist 1, wenn alle Eingangsvariablen 1 sind. Ansonsten ist der Ausgang 0. Das Funktionszeichen ist nicht eindeutig definiert. Meist wird ‚&' (Kaufmanns-Und) verwendet. Daneben sind ‚∧' (umgekehrtes v), der Multiplikationspunkt ‚·' sowie das direkte Aneinanderfügen der Operatoren möglich. In der Übersicht aller Funktionen in Tab. 4.1 finden sich für alle Funktionen die verschiedenen Schreibweisen.

Das Verhalten der UND-Verknüpfung ist in der Funktionstabelle in Abb. 4.5 dargestellt. Alle vier Kombinationsmöglichkeiten für die beiden Eingänge sind aufgezählt; nur wenn A und B gleich 1 sind, ist auch Y gleich 1. Abb. 4.5 zeigt auch das Schaltzeichen der UND-Verknüpfung.

Eine UND-Verknüpfung ist auch für mehr als zwei Eingangsvariablen möglich. Abb. 4.6 zeigt Funktionstabelle und Schaltzeichen bei drei Eingangsvariablen. Genauso sind Funktionen mit vier, fünf oder mehr Eingangsvariablen möglich und werden auch in der Praxis verwendet.

Tab. 4.1 Funktionen für zwei Eingangsvariablen

Ausgabe für AB =				Logische Funktion		Bezeichnung
11	01	10	00			
0	0	0	0	$Y = 0$		Konstante 0
0	0	0	1	$Y = \overline{A \vee B}$		NOR
0	0	1	0	$Y = A \,\&\, \bar{B}$		Inhibition
0	0	1	1	$Y = \bar{B}$	Oder: $Y = \neg B$	Negation *(B)*
0	1	0	0	$Y = \bar{A} \,\&\, B$		Inhibition
0	1	0	1	$Y = \bar{A}$	Oder: $Y = \neg A$	Negation *(A)*
0	1	1	0	$Y = A \oplus B$		XOR, Antivalenz
0	1	1	1	$Y = \overline{A \,\&\, B}$		NAND
1	0	0	0	$Y = A \,\&\, B$	Oder: $Y = A \wedge B = A \cdot B = AB$	UND
1	0	0	1	$Y = \overline{A \oplus B}$	(Selten: $Y = A \leftrightarrow B$)	XNOR, Äquivalenz
1	0	1	0	$Y = A$		Identität *(A)*
1	0	1	1	$Y = A \vee \bar{B}$	(Selten: $Y = B \rightarrow A$)	Implikation
1	1	0	0	$Y = B$		Identität *(B)*
1	1	0	1	$Y = \bar{A} \vee B$	(Selten: $Y = A \rightarrow B$)	Implikation
1	1	1	0	$Y = A \vee B$	(Selten: $Y = A + B$)	ODER
1	1	1	1	$Y = 1$		Konstante 1

Abb. 4.5 Funktionstabelle und Schaltzeichen der UND-Verknüpfung

A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

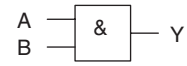
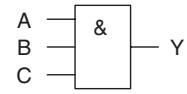


Abb. 4.6 Funktionstabelle und Schaltzeichen einer UND-Verknüpfung mit drei Eingangsvariablen

A	B	C	Y
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1



4.2.2 ODER-Verknüpfung

Auch die *ODER-Verknüpfung* wurde in Kapitel 1 kurz vorgestellt. Der Ausgang *Y* ist 1, wenn ein oder mehrere Eingangsvariablen 1 sind. Nur wenn alle Eingangsvariablen 0 sind ist auch der Ausgang 0. Die Funktionstabelle und das Schaltzeichen sind in Abb. 4.7 dargestellt. Auch die ODER-Funktion kann mehr als zwei Eingänge verknüpfen. Als Symbol in der Schaltfunktion wird ‚≥1' verwendet. In Formeln wird ‚∨' (mathematisches Symbol) oder ‚v' (Buchstabe) benutzt, auch das Plus-Zeichen ‚+' wird manchmal verwendet.

4.2.3 Negation, Inverter

Die *Negation* gibt das „Gegenteil" des Eingangswerts aus, also bei einer 0 eine 1 und bei einer 1 eine 0 (Abb. 4.8). In Formeln wird die Negation durch einen Strich über der Variablen oder Voranstellen des Zeichens ‚¬' dargestellt. Auch ganze Ausdrücke können durch einen Strich oberhalb negiert werden. Ein Beispiels dafür ist das XNOR in Tab. 4.1.

Das Schaltungselement wird *Inverter* genannt. In Schaltzeichen wird die Negation durch einen Kreis dargestellt. Als Schaltzeichen für den Inverter werden drei verschiedene Varianten verwendet (Abb. 4.8). Das untere Schaltsymbol mit dem Dreieck ist am prägnantesten und wird in der Praxis meist benutzt.

Das Sonderzeichen ‚¬' ist etwas umständlich zu erzeugen, darum wird auch der Schrägstrich ‚/' als Präfix oder die Raute ‚#' als Suffix zum Kennzeichen einer Negation verwendet. Die Invertierung des Wertes *A* schreibt sich dann also */A* oder *A#*.

Abb. 4.7 Funktionstabelle und Schaltzeichen der ODER-Verknüpfung

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1

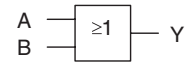
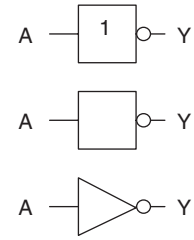


Abb. 4.8 Funktionstabelle der Negation und drei Variationen des Schaltzeichens für einen Inverter

A	Y
0	1
1	0



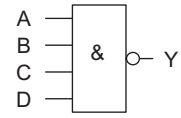
4.2.4 NAND-Verknüpfung

Durch Kombination einer UND-Verknüpfung und einer Negation am Ausgang ergibt sich die *NAND-Verknüpfung*. Der Name leitet sich aus dem englischen „not and“ ab. Das Schaltbild entspricht der UND-Verknüpfung mit einem Kreis am Ausgang für die Negation. Die Funktion ist für zwei oder mehr Eingangsvariablen definiert und Abb. 4.9 zeigt Funktionstabelle und Schaltzeichen für vier Variablen.

Formeln verwenden das UND-Symbol ‚&‘ und negieren den ganzen Ausdruck durch einen Strich oberhalb (siehe Tab. 4.1). Dies gilt auch für NOR und XNOR.

Abb. 4.9 Funktionstabelle und Schaltzeichen der NAND-Verknüpfung

A	B	C	D	Y
0	0	0	0	1
0	0	0	1	1
0	0	1	0	1
0	0	1	1	1
0	1	0	0	1
0	1	0	1	1
0	1	1	0	1
0	1	1	1	1
1	0	0	0	1
1	0	0	1	1
1	0	1	0	1
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	0



4.2.5 NOR-Verknüpfung

Durch Kombination einer ODER-Verknüpfung und einer Negation am Ausgang ergibt sich die *NOR-Verknüpfung*. Der Name leitet sich aus dem englischen „not or“ ab. Das Schaltbild entspricht der ODER-Verknüpfung mit einem Kreis am Ausgang für die Negation (Abb. 4.10). Auch diese Funktion ist für zwei oder mehr Eingangsvariablen definiert.

4.2.6 XOR-Verknüpfung

Die *XOR-Verknüpfung* ist in der Grundform zunächst für zwei Eingangsvariablen definiert und ergibt eine 1 wenn genau eine Variable 1 ist, aber nicht beide gemeinsam. Dies kann man als „ausschließendes oder“, englisch „exclusive or“ bezeichnen, daher XOR. Manchmal wird die Funktion auch als *Antivalenz* bezeichnet. Dies meint, dass beide Eingänge unterschiedlichen Wert haben müssen, damit der Ausgang 1 wird. Eine XOR-Verknüpfung mit mehr als zwei Eingängen ist 1, wenn die Anzahl der 1-Werte am Eingang ungerade ist.

In Formeln wird das XOR durch das Symbol ‚⊕‘ dargestellt. In Schaltzeichen wird die Bezeichnung ‚=1‘ verwendet (Abb. 4.11).

4.2.7 XNOR-Verknüpfung

Die *XOR-Verknüpfung* mit negiertem Ausgang wird als XNOR-Verknüpfung bezeichnet („exclusive not or“). Funktion und Schaltzeichen sind in Abb. 4.12 dargestellt.

Abb. 4.10 Funktionstabelle und Schaltzeichen der NOR-Verknüpfung

A	B	Y
0	0	1
0	1	0
1	0	0
1	1	0

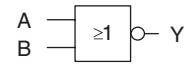


Abb. 4.11 Funktionstabelle und Schaltzeichen der XOR-Verknüpfung

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	0

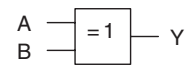
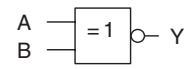


Abb. 4.12 Funktionstabelle und Schaltzeichen der XNOR-Verknüpfung

A	B	Y
0	0	1
0	1	0
1	0	0
1	1	1



Manchmal wird die Funktion auch als *Äquivalenz* bezeichnet. Dies meint, dass in der Grundform mit zwei Eingängen beide Eingänge den gleichen Wert haben müssen, damit der Ausgang 1 wird. Bei mehr als zwei Eingängen ist die XNOR-Verknüpfung 1, wenn die Anzahl der 1-Werte am Eingang gerade ist.

4.2.8 Weitere Verknüpfungen

Neben den genannten Verknüpfungen sind weitere Funktionen möglich. Bei nur einem Eingang gibt es noch die *Identität,* bei der der Ausgang gleich dem Eingang ist.

Alle möglichen Verknüpfungen mit zwei Eingängen sind in Tab. 4.1 aufgeführt. Eine Funktionstabelle für zwei Eingänge hat vier Einträge und für jeden Eintrag sind zwei Werte 0 und 1 möglich. Also sind $2^4 = 16$ Funktionen theoretisch denkbar. Einige dieser Funktionen sind trivial, beispielsweise Ausgang ist immer 0 oder Ausgang ist identisch Eingang *A*. Einige Funktionen sind die oben genannten Verknüpfungen, also UND, ODER, XOR und so weiter.

Daneben gibt es noch *Implikation* und *Inhibition* als weitere Verknüpfungen. Die Funktionen selbst werden verwendet, aber die Begriffe sind in der Praxis nicht üblich. Stattdessen wird die Funktion über eine Grundfunktion beschrieben, also beispielsweise „*A* und nicht *B*“ für Eintrag drei der Tabelle.

4.2.9 Logikstufen

Alle Verknüpfungen können auch in einer mehrstufigen Funktion verwendet werden, bei der das Ergebnis einer Verknüpfung die Eingabe einer weiteren Verknüpfung ist. Die Anzahl der aufeinander folgenden Verknüpfungen wird als Stufigkeit bezeichnet. Der Begriff bezieht sich sowohl auf die Logik als auch auf deren Umsetzung als Schaltung.

- **Einstufige Logik:** Eine Logik und digitale Schaltung wird als einstufig bezeichnet, wenn zwischen Eingang und Ausgang nur eine Verknüpfung vorhanden ist.
- **Zweistufige Logik:** Eine Logik und digitale Schaltung wird als zweistufig bezeichnet, wenn zwischen Eingang und Ausgang zwei Verknüpfungen in Kette geschaltet sind.
- ***n*-stufige Logik:** Eine Logik und digitale Schaltung wird als *n*-stufig bezeichnet, wenn zwischen Eingang und Ausgang *n* Verknüpfungen in Kette geschaltet sind.

Bei der Anzahl der Stufen wird eine Negation am Eingang oder Ausgang nicht als separate Stufe gezählt.

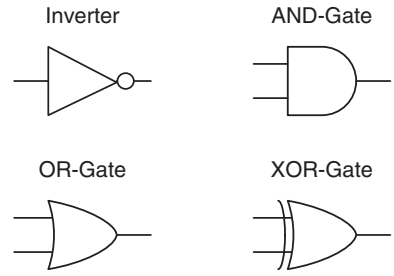


Abb. 4.13 US-amerikanische Logiksymbole

Beispiele für Logikfunktionen mit verschiedenen Stufen:

- **Einstufige Logik:** $Y = A \vee \bar{B}$
- **Zweistufige Logik:** $Y = \overline{(\bar{A} \,\&\, B) \vee (C \,\&\, \bar{D})}$
- **4-stufige Logik:** $Y = A \,\&\, (\bar{B} \vee (C \,\&\, (\bar{D} \vee E)))$

Bedeutung hat die Stufenzahl insbesondere für eine Schaltungsrealisierung. Jede Verknüpfung entspricht einem Logikgatter in der Schaltung. Dabei addieren sich die Verzögerungszeiten sämtlicher Stufen. Deshalb sollte für zeitkritische Entwürfe die Anzahl der Stufen so klein wie möglich sein.

4.2.10 US-amerikanische Logiksymbole

In englischsprachiger Literatur und in Datenblättern finden Sie auch Logiksymbole in US-amerikanischer Darstellungsweise. Diese sind in Abb. 4.13 dargestellt. Durch einen Kreis am Ausgang werden die Varianten mit invertiertem Ausgang gekennzeichnet, also aus AND wird NAND, aus XOR wird XNOR.

Man kann sich die Symbole merken, indem man bei der geraden linken Kante des AND an die vertikalen Striche des A und bei der gebogenen linken Kante des OR an die Rundungen des O denkt.

4.3 Rechenregeln der Schaltalgebra

4.3.1 Vorrangregeln

Genau wie in der elementaren Algebra hat auch die Schaltalgebra *Vorrangregeln*. In der elementaren Algebra gilt „Punktrechnung vor Strichrechnung“, also hat die Multiplikation Vorrang vor der Addition.

In der Schaltalgebra hat das Negationszeichen den größten Vorrang und es kann für eine einzelne Variable oder für einen gesamten Ausdruck stehen. An nächster Stelle sind nach DIN die Verknüpfungszeichen für UND, ODER, NAND und NOR gleichrangig. Danach folgen im Vorrang die Symbole für Implikation, Äquivalenz und Antivalenz, die

untereinander wiederum gleichrangig sind. Da die Verknüpfungszeichen für UND sowie ODER die gleiche Priorität haben, müssen innerhalb einer Gleichung mit UND- und ODER-Verknüpfungen also die einzelnen Terme in Klammern gesetzt werden.

Allerdings wird der Vorrang in der Praxis anders gehandhabt. Den stärksten Vorrang hat weiterhin das Negationszeichen. Dann gilt allerdings „UND vor ODER“, das heißt die UND-Verknüpfung hat Vorrang vor der ODER-Verknüpfung. Dies spart oftmals Schreibarbeit und Klammern. Alle anderen Verknüpfungen werden üblicherweise per Klammer geordnet, um Missverständnisse zu vermeiden.

Auch in diesem Buch wird die Praxisregel „UND vor ODER“ benutzt. Abb. 4.14 zeigt die verschiedenen Schreibweisen. Alle drei Ausdrücke sind gleichwertig.

4.3.2 Rechenregeln

Rechenregeln zum Umformen von Funktionen gelten in der Schaltalgebra ähnlich wie in der elementaren Algebra. Die Rechenregeln gelten für UND- sowie ODER-Verknüpfungen. Für alle Rechenregeln wird auf mathematische Beweise verzichtet. Die meisten Regeln können verifiziert werden, indem alle möglichen Werte eingesetzt werden.

4.3.2.1 Vereinfachungsregeln für eine Variable

Es gibt eine Reihe von *Vereinfachungsregeln,* die gelten, wenn nur eine Variable und eventuell eine Konstante vorhanden ist.
Eine Variable ODER die Konstante 0 ergibt die Variable:

$$A \vee 0 = A$$

Eine Variable ODER die Konstante 1 ergibt 1:

$$A \vee 1 = 1$$

Eine Variable UND die Konstante 0 ergibt 0:

$$A \,\&\, 0 = 0$$

Eine Variable UND die Konstante 1 ergibt die Variable:

$$A \,\&\, 1 = A$$

Eine Variable ODER sich selbst ergibt die Variable:

$$A \vee A = A$$

Y	= (A & B) v (C & D)	korrekt nach DIN
	= A & B v C & D	„UND vor ODER“
	= A B v C D	verkürzt ohne ‚&‘

Abb. 4.14 Vorrangregeln der Schaltalgebra

Eine Variable UND sich selbst ergibt die Variable:

$$A \,\&\, A = A$$

Eine Variable ODER ihre Negation ergibt 1:

$$A \vee \bar{A} = 1$$

Eine Variable UND ihre Negation ergibt die 0:

$$A \,\&\, \bar{A} = 0$$

Eine Variable doppelt negiert ergibt wieder die Variable:

$$\bar{\bar{A}} = A$$

Einige dieser Rechenregeln haben Ähnlichkeit zur elementaren Algebra, also der Schulmathematik.

- Eine Zahl plus Null ergibt wieder die Zahl.
- Eine Zahl mal Null ergibt Null.
- Eine Zahl mal Eins ergibt wieder die Zahl.

Für andere Rechenregeln gibt es jedoch keine Entsprechung.

- Eine Zahl mal oder plus sich selbst ergibt keine Konstante.

4.3.2.2 Kommutativgesetz

Das *Kommutativgesetz,* oder Vertauschungsgesetz, besagt, dass die Reihenfolge der Operanden vertauscht werden darf. Es gilt also:

$$A \,\&\, B = B \,\&\, A$$

$$A \vee B = B \vee A$$

4.3.2.3 Assoziativgesetz

Das *Assoziativgesetz,* oder Verbindungsgesetz, besagt, dass Rechenoperationen mit dem gleichen Operator in beliebiger Reihenfolge durchgeführt werden dürfen. Es gilt also:

$$A \,\&\, B \,\&\, C = (A \,\&\, B) \,\&\, C = A \,\&\, (B \,\&\, C) = (A \,\&C) \,\&\, B$$

$$A \vee B \vee C = (A \vee B) \vee C = A \vee (B \vee C) = (A \vee C) \vee B$$

4.3.2.4 Distributivgesetz

Das *Distributivgesetz,* oder Verteilungsgesetz, besagt, dass ein Operand vor einer Klammer auf Operatoren in einer Klammer verteilt werden darf. Dies wird in der elementaren Algebra als Ausmultiplizieren und Ausklammern bezeichnet. Es gilt also:

$$A \,\&\, (B \vee C) = (A \,\&\, B) \vee (A \,\&\, C)$$

$$A \vee (B \,\&\, C) = (A \vee B) \,\&\, (A \vee C)$$

4.3.2.5 De Morgansche Gesetze

Die *de Morganschen Gesetze* sind zwei Regeln, die besagen:

- Eine NAND-Verknüpfung kann ersetzt werden durch eine ODER-Verknüpfung mit negierten Operatoren.
- Eine NOR-Verknüpfung kann ersetzt werden durch eine UND-Verknüpfung mit negierten Operatoren.

$$\overline{A \,\&\, B \,\&\, C \,\&\, \ldots \,\&\, X} = \bar{A} \vee \bar{B} \vee \bar{C} \vee \ldots \vee \bar{X}$$
$$\overline{A \vee B \vee C \vee \ldots \vee X} = \bar{A} \,\&\, \bar{B} \,\&\, \bar{C} \,\&\, \ldots \,\&\, \bar{X}$$

Anschaulich gesagt, kann also die Negation des gesamten Ausdrucks ersetzt werden durch Negation der einzelnen Operanden und Tauschen von UND nach ODER beziehungsweise ODER nach UND. Diese Gesetze gelten für beliebig viele Operatoren.

Zu den de Morganschen Gesetzen gibt es kein Äquivalent in der elementaren Algebra, sodass diese Regeln eventuell etwas überraschend aussehen.

4.3.2.6 Shannonsches Gesetz

Das *Shannonsche Gesetz* ist eine Erweiterung der de Morganschen Gesetze. Es besagt, dass in einer Funktion, die aus UND- sowie ODER-Verknüpfungen besteht, alle Variablen negiert und die Operatoren UND sowie ODER vertauscht werden können. Die so entstehende Funktion ergibt dann die Negation der ursprünglichen Funktion. Als Formel schreibt sich dies:

$$\overline{f(A, B, \ldots, X;\ \&,\ \vee)} = f\left(\bar{A}, \bar{B}, \ldots, \bar{X};\ \vee,\ \&\right)$$

Das Shannonsche Gesetz erscheint zunächst sehr theoretisch, hat aber praktische Bedeutung. Mit ihm können logische Ausdrücke umgeformt werden, damit sie besser als Schaltung umgesetzt werden können. In der CMOS-Technologie sind beispielsweise NAND- und NOR-Verknüpfungen einfacher als UND-, ODER-Verknüpfungen. Mit dem Shannonschen Gesetz kann dann umgeformt werden:

$$\bar{\mathrm{A}} \vee (B \,\&\, C) = \overline{A \,\&\, \left(\bar{B} \vee \bar{C}\right)} = \overline{A \,\&\, \overline{(B \,\&\, C)}}$$

Die Funktion kann somit durch zwei NAND-Schaltungen mit jeweils zwei Operatoren implementiert werden.

4.4 Schaltungsentwurf durch Minimieren

Beim Schaltungsentwurf wird für eine bestimmte Aufgabenstellung eine Schaltung entworfen. Aus der Spezifikation wird die logische Funktion erstellt. Diese logische Funktion entspricht einer Schaltung aus Gattern, welche die Funktion ausführt.

In diesem Abschnitt wird die prinzipielle Vorgehensweise erläutert. Für den praktischen Entwurf ist das grafische Verfahren mit *Karnaugh-Diagramm* gut geeignet, welches im nächsten Abschnitt beschrieben wird. Des Weiteren kann die Minimierung rechnergestützt erfolgen. Dabei können, je nach Algorithmus, auch mehrstufige Logikfunktionen entstehen. Mit dem hier vorgestellten Verfahren wird stets eine zweistufige Logik erzeugt.

4.4.1 Minterme

Für den Schaltungsentwurf werden sogenannte *Minterme* verwendet. Ein Minterm ist eine UND-Verknüpfung, die jede Variable genau einmal benutzt. Die Variable kann dabei nicht-negiert oder negiert verwendet werden. Bei n Eingangsvariablen existieren 2^n verschiedene Minterme. Bei drei Variablen *A, B, C* wären also acht verschiedene Minterme möglich. Alle drei Variablen werden nicht-negiert oder negiert verwendet, beispielsweise:

$$A \,\&\, B \,\&\, C\,;\; A \,\&\, \bar{B} \,\&\, C\,;\; \bar{A} \,\&\, \bar{B} \,\&\, \bar{C}$$

Das Besondere am Minterm ist, dass er bei genau einer Kombination der Eingangsvariablen den Ausgangswert 1 ergibt und sonst 0 ist. Dies ergibt sich dadurch, dass die UND-Bedingung ja nur bei einer Kombination erfüllt ist. Abb. 4.15 zeigt für drei Minterme die Funktionstabelle. Wenn die nicht-negierten Eingänge gleich 1 und die negierten Eingänge gleich 0 sind, ist der Ausgang gleich 1.

4.4.2 Schaltungsentwurf mit Mintermen

Mit den Mintermen kann direkt eine kombinatorische Schaltung entworfen werden. Dazu werden die Minterme ausgewählt, welche eine 1 ausgeben sollen. Die Minterme

Abb. 4.15 Funktionstabelle für drei Minterme

A	B	C	A&B&C	A&$\bar{B}$&C	$\bar{A}$&$\bar{B}$&$\bar{C}$
0	0	0	0	0	1
0	0	1	0	0	0
0	1	0	0	0	0
0	1	1	0	0	0
1	0	0	0	0	0
1	0	1	0	1	0
1	1	0	0	0	0
1	1	1	1	0	0

werden dann ODER-verknüpft, damit die 1-Werte der Minterme auch in der Gesamtschaltung eine 1 ausgeben. Diese Beschreibung wird als *disjunktive Normalform* (DNF) bezeichnet. Disjunktion ist dabei eine andere Bezeichnung für die ODER-Funktion.

Betrachten wir als Beispiel die Majoritätsschaltung dessen Funktionstabelle in Abb. 4.3 dargestellt ist. Die Schaltung soll für vier Eingangskombinationen eine 1 ausgeben. Die Minterme für diese vier Kombinationen werden ausgewählt und ODER-verknüpft. Dies ergibt die Funktion:

$$Y = \left(\bar{A} \,\&\, B \,\&\, C\right) \vee \left(A \,\&\, \bar{B} \,\&\, C\right) \vee \left(A \,\&\, B \,\&\, \bar{C}\right) \vee (A \,\&\, B \,\&\, C)$$

4.4.3 Minimierung von Mintermen

Die disjunktive Normalform, also die ODER-Verknüpfung der Minterme ist eine logische Gleichung, welche die geforderte Funktion ausführt. Allerdings kann die Normalform meist noch vereinfacht werden. Diese Vereinfachung wird als Minimierung bezeichnet. Dabei werden Terme anhand der Rechenregeln der Schaltalgebra zusammengefasst. Wenn ein Term nicht mehr weiter zusammengefasst werden kann, wird er als *Primterm* bezeichnet.

Bei der Majoritätsschaltung können unter anderem die Terme $\left(\bar{A} \,\&\, B \,\&\, C\right)$ sowie $(A \,\&\, B \,\&\, C)$ zusammengefasst werden. In beiden Termen müssen B und C den Wert 1 haben. A soll im ersten Term 0, im zweiten Term 1 sein. Das heißt, beide mögliche Werte für A sind erlaubt und daher braucht A nicht beachtet zu werden. Die Terme können deshalb zum Primterm $(B \,\&\, C)$ zusammengefasst werden.

Diese anschauliche Erklärung lässt sich auch über die Rechenregeln herleiten:

- Assoziativgesetz: $\left(\bar{A} \,\&\, B \,\&\, C\right) \vee (A \,\&\, B \,\&\, C) = \left(\bar{A} \,\&\, (B \,\&\, C)\right) \vee (A \,\&\, (B \,\&\, C))$
- Distributivgesetz: $\left(\bar{A} \,\&\, (B \,\&\, C)\right) \vee (A \,\&\, (B \,\&\, C)) = \left(\bar{A} \vee A\right) \,\&\, (B \,\&\, C)$
- Vereinfachungsregel: $\left(\bar{A} \vee A\right) = 1$
- Vereinfachungsregel: $1 \,\&\, (B \,\&\, C) = (B \,\&\, C)$

Auf die gleiche Weise können die Terme $\left(A \,\&\, \bar{B} \,\&\, C\right)$ sowie $\left(A \,\&\, B \,\&\, \bar{C}\right)$ mit dem Term $(A \,\&\, B \,\&\, C)$ zusammengefasst werden. Dabei fällt die Variable $\bar{B}$ beziehungsweise $\bar{C}$ weg. Die minimierte Majoritätsfunktion lautet:

$$Y = (B \,\&\, C) \vee (A \,\&\, C) \vee (A \,\&\, B)$$

Diese Minimierung ist allerdings rechnerisch sehr aufwendig. Man muss genau aufpassen, welche Terme miteinander kombiniert werden können. Für die Ermittlung der Primterme ist das grafische Verfahren nach Karnaugh wesentlich einfacher, welches in Abschn. 4.6 erläutert wird.

4.4.4 Maxterme

Für den Schaltungsentwurf können auch sogenannte *Maxterme* verwendet werden. Ein Maxterm ist eine ODER-Verknüpfung, die jede Variable genau einmal verwendet. Wie bei Mintermen kann jede Variable wiederum nicht-negiert oder negiert sein. Für den Maxterm gilt dann, dass er bei genau einer Kombination der Eingangsvariablen den Ausgangswert 0 ergibt und sonst 1 ist. Maxterme sind:

$$A \vee B \vee C;\ A \vee \bar{B} \vee C;\ \bar{A} \vee \bar{B} \vee \bar{C}$$

Abb. 4.16 zeigt für drei Maxterme die Funktionstabelle. Nur wenn die nicht-negierten Eingänge gleich 0 sowie die negierten Eingänge gleich 1 sind, ist der Ausgang gleich 0.

Maxterme sind also das Gegenstück zu Mintermen. Eine Funktion benutzt die UND-, die andere die ODER-Verknüpfung. Bei einer Funktion gibt es eine einzige 1, bei der anderen eine einzige 0.

4.4.5 Schaltungsentwurf mit Maxtermen

Auch aus den Maxtermen kann direkt eine kombinatorische Schaltung entworfen werden. Dazu werden die Maxterme ausgewählt, welche eine 0 ausgeben und dann UND-verknüpft, damit diese Nullen in Kombinationen der Gesamtschaltung eine 0 ergeben. Diese Beschreibung wird als *konjunktive Normalform* (KNF) bezeichnet. Konjunktion ist dabei eine andere Bezeichnung für die UND-Funktion.

Die Majoritätsschaltung gibt für vier Eingangskombinationen eine 0 aus. Die Maxterme für diese vier Kombinationen werden ausgewählt und UND-verknüpft. Dies ergibt die Funktion:

$$Y = (A \vee B \vee C)\ \&\ \left(\bar{A} \vee B \vee C\right)\ \&\ \left(A \vee \bar{B} \vee C\right)\ \&\ \left(A \vee B \vee \bar{C}\right)$$

4.4.6 Minimierung von Maxtermen

Auch die konjunktive Normalform, also die UND-Verknüpfung von Maxtermen kann meist noch vereinfacht werden.

Abb. 4.16 Funktionstabelle für drei Maxterme

A	B	C	AvBvC	Av$\bar{B}$vC	$\bar{A}$v$\bar{B}$v$\bar{C}$
0	0	0	0	1	1
0	0	1	1	1	1
0	1	0	1	0	1
0	1	1	1	1	1
1	0	0	1	1	1
1	0	1	1	1	1
1	1	0	1	1	1
1	1	1	1	1	0

Für die Majoritätsschaltung kann der Maxterm $(A \vee B \vee C)$ jeweils mit den drei anderen Termen zusammengefasst werden. Die einzelnen Rechenschritte sollen hier jedoch nicht einzeln aufgeführt werden. Die minimierte Majoritätsfunktion lautet:

$$Y = (B \vee C) \,\&\, (A \vee C) \,\&\, (A \vee B)$$

4.5 Schaltungsminimierung mit Karnaugh-Diagramm

Im vorherigen Abschnitt wurde gezeigt, dass disjunktive und konjunktive Normalformen durch Minimierung vereinfacht werden können. Ein Karnaugh-Diagramm (Aussprache „Karnoh“) führt diese Vereinfachung grafisch durch. Durch die Darstellung kann direkt erkannt werden, welche Terme miteinander verbunden werden können. Eine Minimierung mit Karnaugh-Diagramm ist sehr gut für Funktionen mit zwei bis vier Variablen geeignet. Für fünf oder sechs Variablen ist das Verfahren ebenfalls möglich, erfordert dann aber etwas Übung und gutes räumliches Vorstellungsvermögen.

Das Verfahren kann sowohl für die disjunktive als auch für die konjunktive Normalform durchgeführt werden. Hier soll hauptsächlich die disjunktive Normalform vorgestellt werden. Das Verfahren ist auch unter dem Namen *Venn-Diagramm, Karnaugh-Veitch-Diagramm* (Aussprache „Karnoh-Fietsch“) oder *KV-Diagramm* bekannt.

Das Karnaugh-Diagramm ist insbesondere wichtig, da es die Zusammenhänge von Schaltalgebra, logischen Verknüpfungen und Schaltungsimplementierung verdeutlicht. In der Praxis erfolgt die Schaltungsminimierung heutzutage meist durch Computerprogramme zur Schaltungssynthese.

4.5.1 Grundsätzliche Vorgehensweise

Das Karnaugh-Diagramm ist im Prinzip eine andere Anordnung der Wahrheitstabelle. Die Eingangsvariablen werden am horizontalen und vertikalen Rand eines schachbrettartig unterteilten Rechtecks angeordnet. Für n Eingangsvariablen erhält man somit 2^n Felder. Dabei sind sie so angeordnet, dass jedes Feld einem Minterm entspricht und sich zwei horizontal oder vertikal benachbarte Felder nur in einer Eingangsvariablen unterscheiden. In die Felder werden die Werte 0 und 1 der Ausgangsvariablen eingetragen. Benachbarte 1-Felder werden dann wie im Abschn. 4.5.4 zusammengefasst:

$$(A \,\&\, B) \vee \left(A \,\&\, \bar{B}\right) = A \,\&\, \left(B \vee \bar{B}\right) = A$$

Im Karnaugh-Diagramm sind auch Felder am rechten und linken bzw. oberen und unteren Rand benachbart, denn auch sie unterscheiden sich nur in einer Variablen. Es müssen möglichst viele benachbarte 1-Felder zu einem Block zusammengefasst werden. Die logische Gleichung wird dann minimal, wenn die Blöcke möglichst viele Felder enthalten und die Anzahl der Blöcke minimal ist.

Die Vorgehensweise zum Aufstellen der disjunktiven Minimalform lautet:

1. Ausgehend von der Wahrheitstabelle wird die benötigte Anzahl der Eingangsvariablen ermittelt und das entsprechende Karnaugh-Diagramm aufgestellt. Die logischen Variablen werden am Rand des KV-Diagramms angeordnet.
2. Anhand der Wahrheitstabelle werden die Werte der Ausgangsvariablen 0 oder 1 in die entsprechenden Felder des Karnaugh-Diagramms eingetragen.
3. Benachbarte 1-Felder werden zu einem Block zusammengefasst.
4. Zwei Blöcke, die sich nur in einer Variablen unterscheiden, sind ebenfalls benachbart; sie dürfen zu einem größeren Block zusammengefasst werden. Ein Block enthält immer 2^n Felder. Eine Zusammenfassung von zum Beispiel drei oder fünf Feldern ist nicht erlaubt.
5. Ein 1-Feld darf in mehreren Blöcken integriert sein.
6. Jeder Block repräsentiert einen UND-Term (UND-Verknüpfung der Eingangsvariablen).
7. Aus den möglichen Termen (den Blöcken im Diagramm) werden die erforderlichen Terme so gewählt, dass alle 1-Felder berücksichtigt sind.
8. Die logische Gleichung ergibt sich aus der ODER-Verknüpfung der ausgewählten UND-Terme.
9. Die logische Gleichung wird nur dann minimal, falls die Blöcke so groß wie möglich sind und die Anzahl der ausgewählten Blöcke minimal ist.

4.5.2 Karnaugh-Diagramm für zwei Variablen

Bei zwei Variablen hat die Funktionstabelle vier Einträge. Im Karnaugh-Diagramm in Abb. 4.17 werden diese Einträge in vier Feldern dargestellt. Jeder Eintrag entspricht einem Feld und die horizontale Richtung unterscheidet zwischen verschiedenen Werten der Variable *B,* die vertikale Richtung unterscheidet zwischen verschiedenen Werten der Variable *A*. Die Buchstaben *p* bis *s* zeigen die Korrespondenz zwischen Tabelle und Diagramm.

Um eine Funktion zu minimieren, werden die Ausgabewerte der Funktionstabelle in das Diagramm eingetragen. Die beispielhaft gewählte Funktionstabelle in Abb. 4.18 hat einen Eintrag mit Funktionswert 0 und drei Einträge mit Funktionswert 1 und diese Werte finden sich im Karnaugh-Diagramm wieder. Jede 1 entspricht einem Minterm, das heißt, die Funktion könnte durch drei Minterme dargestellt werden.

Im Diagramm kann man jetzt erkennen, welche 1-Einträge, also welche Minterm nebeneinander liegen. Diese benachbarten Minterme können zu einem Term

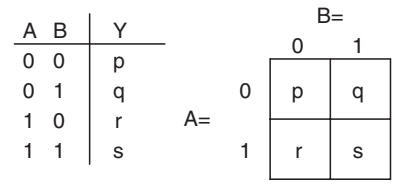


Abb. 4.17 Zuordnung im Karnaugh-Diagramm

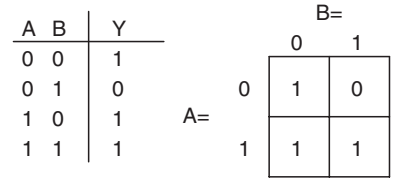


Abb. 4.18 Einträge im Karnaugh-Diagramm

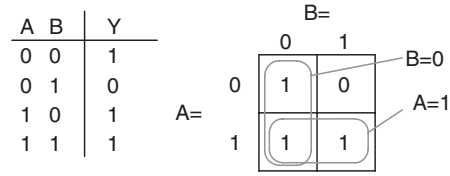


Abb. 4.19 Zusammenfassung von 1-Einträgen

zusammengefasst werden. In Abb. 4.18 sind dies die beiden Einsen in der linken Spalte und die beiden Terme in der unteren Zeile. Eine 1, also ein Minterm darf dabei mehrfach für die Minimierung verwendet werden.

Abb. 4.19 zeigt die zusammengefassten Einträge. Für die linke Spalte ist die Variable B gleich 0. Die Variable A kann 0 oder 1 sein, denn das abgerundete Rechteck der verbundenen Einträge liegt über der oberen und unteren Zeile. Damit entspricht dieser Term der Funktion $B = 0$ gleichbedeutend mit $\bar{B}$. Der andere verbundene Eintrag läuft über die untere Zeile, also A gleich 1. B kann 0 oder 1 sein, denn das Rechteck liegt über den Spalten für beide Werte von B. Der Term ist also $A = 1$ gleichbedeutend mit A. Die minimierte Funktion ergibt sich aus der ODER-Verknüpfung der Terme, also:

$$Y = A \vee \bar{B}$$

4.5.3 Karnaugh-Diagramm für drei Variablen

Für drei Variablen wird das Diagramm auf acht Felder erweitert (Abb. 4.20). An der langen Kante werden dafür zwei Variablen angeordnet. Die Reihenfolge der beiden Variablen ist so zu wählen, dass sich benachbarte Felder weiterhin in nur einer Variable unterscheiden. Diese Reihenfolge entspricht dadurch dem Gray-Code. Beachten Sie, dass linker und rechter Rand benachbart sind.

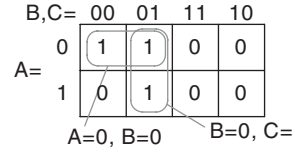
Das Diagramm enthält somit zwei Terme, die sich aus jeweils zwei Mintermen, also zwei 1-Stellen zusammensetzen. Die minimierte Funktion ergibt sich zu:

$$Y = (\bar{A} \,\&\, \bar{B}) \vee (\bar{B} \,\&\, C)$$

Auch Gruppen von vier Funktionswerten können zu einer Vierergruppe zusammengefasst werden. Dies entspricht einer Zusammenfassung von zwei Zweiergruppen, die sich auch nur in einer Variablen unterscheiden. Wenn sich somit weniger Terme und größere Terme ergeben, spart dies Schaltungsaufwand. Die Vierergruppen können quadratisch

Abb. 4.20 Karnaugh-Diagramm mit drei Variablen

A	B	C	Y
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	0



oder über eine ganze Zeile gehen. Abb. 4.21 zeigt ein Karnaugh-Diagramm mit zwei Vierergruppen. Die resultierende Funktion ist:

$$Y = A \vee C$$

Die linke Spalte des Karnaugh-Diagramms enthält die Terme für $B,C = 00$ und die rechte Spalte die Terme für $B,C = 10$. Daher unterscheiden sich diese Terme auch nur in einer Variable (Variable B) und sind benachbart. Zweier- und Vierergruppen können daher über den Rand hinaus verbunden sein. Abb. 4.22 zeigt dies für zwei Karnaugh-Diagramme.

4.5.4 Karnaugh-Diagramm für vier Variablen

Für vier Eingangsvariablen wird das Diagramm auf 16 Felder erweitert, so dass auch die vertikale Achse zwei Variablen enthält, wiederum mit der Reihenfolge in Gray-Codierung. Abb. 4.23 zeigt die Anordnung und zwei Gruppen.

Abb. 4.21 Karnaugh-Diagramm mit Vierergruppen

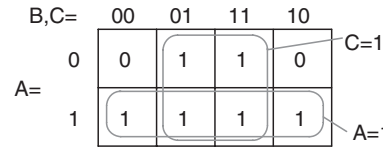
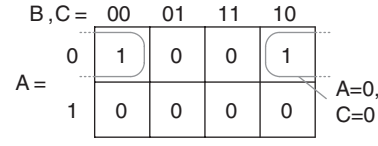
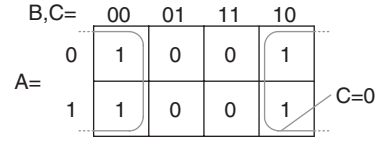


Abb. 4.22 Linker und rechter Rand sind im Karnaugh-Diagramm benachbart





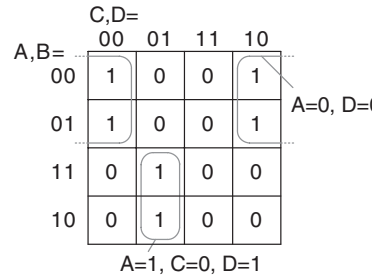


Abb. 4.23 Karnaugh-Diagramm mit vier Variablen

In den 16 Feldern können Gruppen mit zwei, vier oder acht 1-Feldern gebildet werden. Die Gruppengröße muss aber eine Zweierpotenz sein, das heißt eine Gruppe aus sechs Feldern ist nicht möglich. Dies ergibt sich daraus, dass bei einer Zusammenfassung eine Variable aus dem Term entfällt und dadurch die Gruppe jeweils doppelt so groß wird. Für ein Karnaugh-Diagramm mit vier Variablen gibt es also folgende mögliche Gruppen:

- Einzelnes 1-Feld mit allen vier Variablen
- Zweiergruppe mit drei Variablen
- Vierergruppe mit zwei Variablen
- Achtergruppe mit einer Variablen (siehe Abb. 4.24)

Theoretisch kann es dann auch eine 16er-Gruppe ohne Variable geben, das heißt die Funktion ist immer 1.

Wie schon beim Karnaugh-Diagramm für drei Variablen sind hier wieder die Ränder benachbart. Dies gilt natürlich auch für Vierergruppen und zwar auch in der Kombination von oberer, unterer und linker, rechter Rand. Mit anderen Worten, auch die vier Ecken können zu einer Vierergruppe zusammengefasst werden (Abb. 4.25). Dazu müssen aber alle vier Eckfelder eine 1 eingetragen haben. Eine diagonale Zweiergruppe, also Feld links-unten und Feld rechts-oben wäre nicht möglich, da ja auch ansonsten keine diagonalen Felder erlaubt sind.

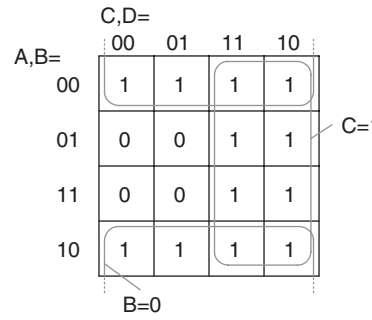


Abb. 4.24 Karnaugh-Diagramm mit Achtergruppen

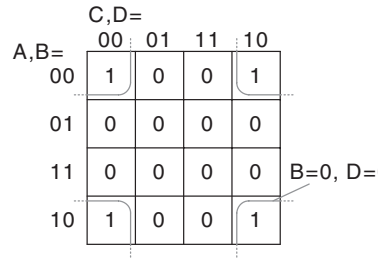


Abb. 4.25 Die vier Ecken können eine Vierergruppe bilden

4.5.5 Auswahl der erforderlichen Terme

Nachdem die 1-Felder zu möglichst großen Gruppen, den Primtermen, zusammengefasst sind, muss im nächsten Schritt überprüft werden, welche Terme erforderlich sind. Dabei sind manchmal alle Primterme erforderlich und manchmal werden Primterme nicht benötigt, sind also redundant. Die Bedingung für die Auswahl der Primterme ist, dass alle 1-Felder in mindestens einem Primterm enthalten sein müssen. Je weniger Primterme benötigt werden und je größer die Terme sind, umso günstiger ist die entstehende Schaltung.

Als Beispiel wird eine Funktion mit sieben 1-Feldern betrachtet, die in Abb. 4.26 im Karnaugh-Diagramm eingetragen sind. Es lassen sich vier Primterme bilden, und zwar eine Vierergruppe und drei Zweiergruppen. Da alle 1-Felder in einem der Primterme enthalten sein müssen, ist Term 1 erforderlich, denn nur er enthält die 1-Felder in der linken Spalte. Auch Term 2 und Term 4 sind erforderlich, denn nur sie enthalten die 1-Felder in der dritten Spalte (für $C,D = 11$). Term 3 hingegen ist nicht erforderlich, denn sein oberes 1-Feld ist bereits in Term 1, das untere 1-Feld in Term 4 enthalten.

4.5.6 Ermittlung der minimierten Funktion

Wenn die erforderlichen Primterme bekannt sind, müssen die logischen Funktionen für diese Terme bestimmt werden. Die Terme sind dabei eine UND-Verknüpfung von nicht-negierten und negierten Eingangsvariablen. Welche Eingangsvariablen im Term enthalten sind, ergibt sich durch die Position des Primterms im Karnaugh-Diagramm. Drei Fälle sind möglich:

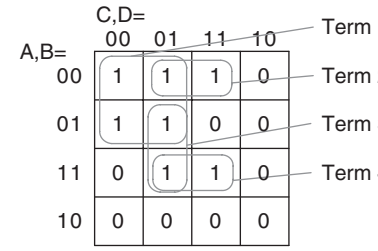


Abb. 4.26 Auswahl der Primterme

- Der Primterm überdeckt nur Zeilen oder Spalten, für die eine Eingangsvariable 1 ist. Dann wird die Variable nicht-negiert in der UND-Verknüpfung verwendet.
- Der Primterm überdeckt nur Zeilen oder Spalten, für die eine Eingangsvariable 0 ist. Dann wird die Variable negiert in der UND-Verknüpfung verwendet.
- Der Primterm überdeckt Zeilen oder Spalten, für die eine Eingangsvariable sowohl 1 als auch 0 sind. Dann wird die Variable nicht in der UND-Verknüpfung verwendet.

Die Formel für die minimierte Funktion ergibt sich dann als ODER-Verknüpfung aller UND-Terme.

Als Beispiel wird die Funktion für Term 1 in Abb. 4.26 ermittelt. Für die vier Eingangsvariablen gilt:

- Der Term überdeckt nur Zeilen in denen die Variable A gleich 0 ist. A wird negiert verwendet.
- In den oberen beiden Zeilen ist die Variable B sowohl 0 als auch 1. B wird nicht verwendet.
- In den überdeckten linken Spalten ist C beides mal 0. C wird negiert verwendet.
- In den beiden linken Spalten ist D sowohl 0 als auch 1. D wird nicht verwendet.
- Die Funktion für Term 1 ist also: $\bar{A} \,\&\, \bar{C}$

Für Term 2 gilt, dass die Variablen A und B gleich 0 sind und daher negiert verwendet werden. D ist gleich 1 und wird nicht-negiert verwendet. C kann 0 und 1 sein und darum in der Funktion nicht enthalten. Der Primterm lautet also: $\bar{A} \,\&\, \bar{B} \,\&\, D$

Für Term 4 sind die Variablen A, B und D gleich 1 und daher nicht-negiert. C ist wie bei Term 2 nicht enthalten und daher lautet der Primterm: $A \,\&\, B \,\&\, D$

Somit ergibt sich die minimierte Funktion für Abb. 4.26 als ODER-Verknüpfung von Term 1, 2 und 4:

$$Y = \bar{A} \,\&\, \bar{C} \vee \bar{A} \,\&\, \bar{B} \,\&\, D \vee A \,\&\, B \,\&\, D$$

4.5.7 Karnaugh-Diagramm mit Don't-Care

Wenn für bestimmte Kombinationen von Eingangswerten keine Ausgabe spezifiziert ist, kann diese Freiheit benutzt werden, um die minimierte Funktion möglichst einfach zu erstellen. Ein Beispiel für Funktionen mit Don't-Care wurde am Anfang des Kapitels in Abschn. 4.2.4 erläutert.

Die Behandlung von Don't-Care-Einträgen bei der Minimierung nach Karnaugh ist relativ einfach. Zunächst werden die Don't-Cares als Strich ‚-' in das Karnaugh-Diagramm eingetragen. Bei der Ermittlung der Primterme werden die Don't-Cares einbezogen, um möglichst große Primterme zu bilden. Bei der Auswahl der erforderlichen Primterme werden die Don't-Cares dann nicht berücksichtigt, denn sie müssen nicht in einem Primterm enthalten sein.

Anschaulich gesprochen werden die Don't-Cares zur Bildung der Primterme wie 1-Werte, bei der Auswahl der erforderlichen Primterme wie 0-Werte behandelt. Primterme die nur aus Don't-Cares bestehen, werden nicht eingetragen. In der resultierenden minimierten Funktion ergeben sich dann für manche Don't-Cares eine 1, für andere eine 0.

Als Beispiel für die Behandlung von Don't-Cares soll die in Abschn. 4.2.3 beschriebene Primzahlerkennung für die Zahlen 0 bis 9 minimiert werden. Die Funktionstabelle findet sich in Abb. 4.4 und hat sechs Don't-Cares. Der Eingang ist die vierstellige Dualzahl *A(3:0),* so dass die Eingangsvariablen hier nicht *A* bis *D* heißen. Abb. 4.27 zeigt auf der linken Seite die Zuordnung zwischen Feldern im Karnaugh-Diagramm und Dezimalzahlen.

Im Karnaugh-Diagramm in Abb. 4.27 (rechts) können drei Vierergruppen als Primterme gebildet werden. In der dritten Zeile wäre eine weitere Vierergruppe nur aus Don't-Cares möglich, die aber nicht eingetragen wird, da sie ohne 1-Felder nicht erforderlich sein kann.

Term 1 ist erforderlich, da nur er das 1-Feld rechts oben abdeckt. Term 2 ist erforderlich, da nur er das 1-Feld für ‚0101' abdeckt. Mit diesen beiden Termen sind sämtliche 1-Felder abgedeckt, sodass Term 3 nicht erforderlich ist.

Zur Bestimmung der Terme werden wieder die Eingangsvariablen betrachtet. Für Term 1 ist *A(2)* stets 0 und *A(1)* stets 1, also werden sie negiert beziehungsweise nicht-negiert berücksichtigt. Die Variablen *A(3)* und *A(0)* treten sowohl als 0 und 1 auf, entfallen also. Term 1 lautet somit $\overline{A(2)}\,\&\,A(1)$. Term 2 berechnet sich als $A(2)\,\&\,A(0)$. Die minimierte Funktion für die Primzahlerkennung ist die ODER-Verknüpfung der Terme und lautet:

$$Y = \overline{A(2)}\,\&\,A(1) \vee A(2)\,\&\,A(0)$$

Durch die gewählten Terme werden vier der sechs Don't-Care-Felder umfasst. Für diese Felder ergibt sich also eine 1 als Ausgabe, für die anderen beiden Don't-Care-Felder eine 0. Da laut Aufgabenstellung diese Eingangskombinationen nicht auftreten, konnten sie frei belegt werden.

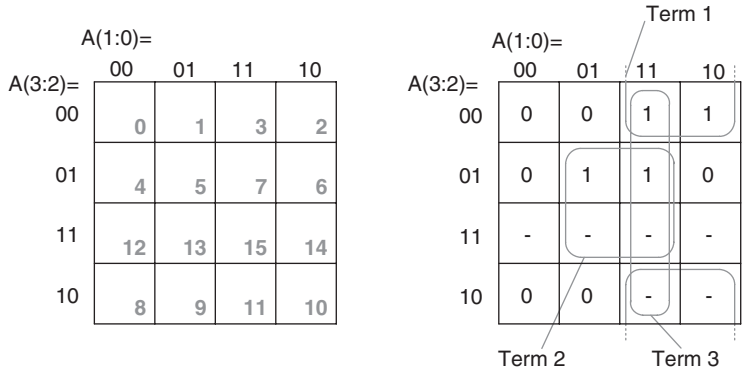


Abb. 4.27 Karnaugh-Diagramm für Primzahlerkennung mit Don't-Cares

Vielleicht fragen Sie sich beim Betrachten von Abb. 4.27, ob die Terme nicht auch kleiner gewählt werden könnten. Term 1 beispielsweise könnte auch als Zweiergruppe mit den beiden 1-Feldern aus der ersten Zeile eingetragen werden. Ein solcher Term würde tatsächlich eine korrekte logische Funktion ergeben. Er wäre aber aufwendiger als die Vierergruppe. Term 1 als Zweiergruppe entspricht $\overline{A(3)} \,\&\, \overline{A(2)} \,\&\, A(1)$, während die Vierergruppe durch den einfacheren Term $\overline{A(2)} \,\&\, A(1)$ umgesetzt wird.

4.5.8 Karnaugh-Diagramm für mehr als vier Variablen

Auch Funktionen mit fünf oder sechs Variablen können prinzipiell mit dem Karnaugh-Diagramm minimiert werden. Allerdings sind dafür mehr als zwei Dimensionen erforderlich und man muss sich die Felder räumlich hintereinander oder übereinander vorstellen. Abb. 4.28 zeigt eine Darstellung mit 32 Feldern für fünf Variable, bei der die beiden Hälften gedanklich an der mittleren, dickeren Linie geknickt werden. Felder aus rechter und linker Hälfte liegen dadurch übereinander. Eine mögliche Vierergruppe ist zur Verdeutlichung eingetragen. Für sechs Variable müsste man in einem 64er-Feld auch eine obere und untere Hälfte übereinander legen.

Diese Darstellung ist allerdings unübersichtlich und daher fehleranfällig. Ein rechnergestütztes Verfahren wäre daher sinnvoll.

4.5.9 Karnaugh-Diagramm der konjunktiven Normalform

Bisher wurde stets die disjunktive Normalform beschrieben, aber in ähnlicher Weise kann auch die konjunktive Normalform aufgestellt werden. Entsprechend der Symmetrie der Schaltalgebra (siehe de Morgansche Gesetze in Abschn. 4.3.2.5) ist die Vorgehensweise praktisch spiegelbildlich. Es werden also anstatt der 1-Felder die 0-Felder verbunden, gegebenenfalls mithilfe der Don't-Care-Felder. Dann werden die ODER-Terme UND-verknüpft.

Als Beispiel soll die Primzahlerkennung auch in der konjunktiven Normalform minimiert werden. In Abb. 4.29 werden aus den 0-Feldern mithilfe der Don't-Cares eine Achtergruppe und drei Vierergruppen gebildet. Term 1 und 3 sind erforderlich, da es

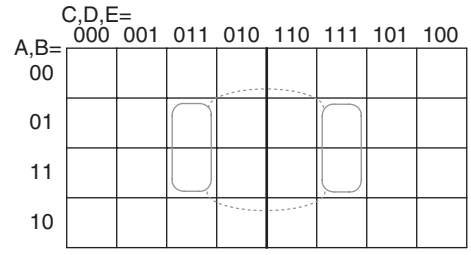


Abb. 4.28 Karnaugh-Diagramm für fünf Variablen

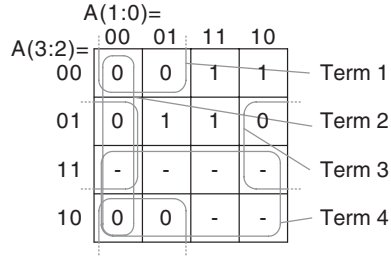


Abb. 4.29 Primzahlerkennung in der konjunktiven Normalform

jeweils ein 0-Feld gibt, welches nur in ihnen enthalten ist. Damit sind alle 0-Felder abgedeckt, so dass Term 2 und 4 redundant sind.

Die minimierte Funktion ergibt sich zu:

$$Y = (A(2) \vee A(1)) \,\&\, \left(\overline{A(2)} \vee A(0)\right)$$

Für die Primzahlerkennung sind die minimierten Funktionen der disjunktiven und konjunktiven Normalform praktisch gleich aufwendig, denn beide Funktionen nutzen drei Verknüpfungen mit jeweils zwei Eingängen. Je nach Funktionstabelle kann eine der Varianten aber auch günstiger als die andere sein. Es gibt Entwurfsprogramme die beide Varianten ausprobieren und die günstigere verwenden.

4.6 VHDL für kombinatorische Schaltungen

4.6.1 Beschreibung logischer Verknüpfungen

Im Kapitel 3 haben Sie die Schaltungsbeschreibung mit VHDL kennengelernt. Die logischen VHDL-Operatoren können verwendet werden, um eine Funktion zu beschreiben. Die gerade in Abschn. 4.5.9 berechnete Logikfunktion würde dann wie folgt lauten:

```
y <= (a(2) or a(1)) and ((not a(2)) or a(0));
```

Die Reihenfolge der Operationen wird durch Klammern vorgegeben. Dies empfiehlt sich, um zweifelsfrei zu definieren, wie die Funktion gemeint und interpretiert werden soll. Es ist besser einige Sekunden für eine weitere Klammer zu investieren, als mehrere Stunden oder länger nach einem Fehler im Code zu suchen.

Die direkte Beschreibung der Logikfunktion ist möglich und wird auch von Programmen verstanden. Viel sinnvoller ist es jedoch, die Funktion zu beschreiben und die Generierung der Logikfunktion dem Programm zu überlassen.

4.6.2 Beschreibung der Funktion

Bei der Funktionsbeschreibung in VHDL wird die Spezifikation durch If- und Case-Anweisungen sowie Zuweisungen beschrieben. Die Funktion soll ja die Primzahl aus dem 4-Bit-Wert *A* erkennen. Eine VHDL-Beschreibung würde darum *A* zunächst in einen Unsigned umwandeln und dann eine Case-Anweisung aufrufen.

```
signal a_u : unsigned (3 downto 0);
…
a_u     <= unsigned(a);
process (a_u)
   begin
   case a_u is
      when  0 => y <= '0';
      when  1 => y <= '0';
      when  2 => y <= '1';
      when  3 => y <= '1';
      when  4 => y <= '0';
      when  5 => y <= '1';
      when  6 => y <= '0';
      when  7 => y <= '1';
      when  8 => y <= '0';
      when  9 => y <= '0';
      when others => y <= '0';
   end case;
end process;
```

Diese Beschreibung benötigt zwar etwas mehr Text, dafür spart man sich die manuelle Schaltungsminimierung für die Logikfunktion. Außerdem ist beim Betrachten des Codes schneller deutlich, welche Funktion ausgeführt wird.

Man kann die Beschreibung auch noch vereinfachen, indem nur die Primzahlen in der Case-Anweisung genannt werden. Alle anderen Werte sind durch den Others-Fall berücksichtigt. Die Case-Anweisung würde dann lauten:

```
   case a_u is
      when  2 => y <= '1';
      when  3 => y <= '1';
      when  5 => y <= '1';
      when  7 => y <= '1';
      when others => y <= '0';
   end case;
```

Im Unterschied zu der Funktionstabelle in Abb. 4.4 werden bei beiden VHDL-Beschreibungen die Don't-Care-Fälle nicht berücksichtigt, sondern die Ausgabe für

Werte größer 10 zu 0 gesetzt. Prinzipiell könnte für ein Don't-Care der Wert ‚-' zugewiesen werden. Dies wird in der Praxis jedoch selten gemacht, da die Einsparungen meist relativ gering sind.

4.7 Übungsaufgaben

Haben Sie den Inhalt des Kapitels verstanden? Prüfen Sie sich mit den Aufgaben und Fragen am Kapitelende. Die Lösungen und Antworten finden Sie am Ende des Buches.

Bei den Auswahlfragen ist immer genau eine Antwort korrekt.

Aufgabe 4.1
Was ist ein Minterm?

a) Eine Logikfunktion die nur für eine Eingangskombination 1 ist
b) Eine Logikfunktion die mit geringst möglicher Geschwindigkeit arbeitet
c) Eine Logikfunktion die nur für eine Eingangskombination 0 ist
d) Eine Logikfunktion die nur aus einem Inverter besteht
e) Eine Logikfunktion die nur aus einem XOR-Gatter besteht

Aufgabe 4.2
Was ist ein Maxterm?

a) Eine Logikfunktion die nur für eine Eingangskombination 0 ist
b) Eine Logikfunktion die nur für eine Eingangskombination 1 ist
c) Eine Logikfunktion die nur aus einem XOR-Gatter besteht
d) Eine Logikfunktion die nur aus einem Inverter besteht
e) Eine Logikfunktion die mit konstanter Geschwindigkeit arbeitet

Aufgabe 4.3
Für eine Stereoanlage soll die eingestellte Lautstärke auf einer vertikalen Skala mit sieben LEDs *(L1* bis *L7)* dargestellt werden. Die Lautstärke ist als 3-Bit-Dualzahl *D2, D1, D0* verfügbar.

Je höher die eingestellte Lautstärke, umso mehr LEDs sollen durch Ausgabe einer 1 leuchten. Bei Lautstärke 0 ($D2, D1, D0 = 000$) sind alle LEDs aus, bei 1 (001) leuchtet nur L1, und so weiter. Abb. 4.30 zeigt den Wert 4 (100) bei dem *L1* bis *L4* leuchten.

Stellen Sie die Funktionstabelle der Schaltung auf.

Aufgabe 4.4
Für einen Spielautomaten soll die Eingabe eines Joysticks akustisch ausgegeben werden. Der Joystick hat vier Schalter *O* (oben), *U* (unten), *L* (links), *R* (rechts). In der Mittelstellung

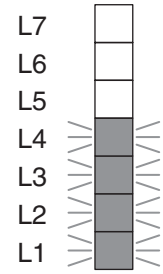


Abb. 4.30 Lautstärkeanzeige einer Stereoanlage

geben alle Schalter 0 aus, bei Auslenkung sind die entsprechenden Schalter 1. Der Joystick kann schräg gehalten werden, sodass ein horizontaler und ein vertikaler Schalter gleichzeitig gedrückt sein können. Die beiden horizontalen bzw. vertikalen Schalter *O* und *U* bzw. *L* und *R* können nicht gleichzeitig gedrückt sein.

Wenn der Joystick aus der Mittelstellung heraus, horizontal oder vertikal gedrückt wird, soll durch Setzen des Ausgangs $T1 = 1$ ein bestimmter Ton ausgegeben werden. Wenn der Joystick schräg gehalten wird und zwei Schalter drückt, soll durch Setzen des Ausgangs $T2 = 1$ ein anderer Ton ausgegeben werden. *T1* ist dann 0.

Stellen Sie die Funktionstabelle der Schaltung zur Erzeugung der Tonansteuerung *T1* und *T2* aus den Schaltern *O, U, R, L* des Joysticks auf. Für Eingangskombinationen, die nicht auftreten können, soll für die Ausgänge ein Don't-Care (‚-') eingetragen werden.

Aufgabe 4.5
Eine kombinatorische Schaltung hat vier Eingänge *A(3:0),* die unten als Dezimalzahl *A* angegeben ist. Es gibt einen Ausgang *Y* der folgende Werte hat:

- 1 bei $A = 5, 7, 10, 11, 14, 15$
- 0 sonst

Hinweis: Die Zuordnung von Dezimalzahl zu Feldern im Karnaugh-Diagramm ergibt sich aus der Zahlendarstellung, ist aber auch in Abb. 4.27 (links) angegeben.

a) Stellen Sie das Karnaugh-Diagramm auf.
b) Ermitteln Sie die Produktterme. Welche Produktterme sind erforderlich?
c) Geben Sie die Funktion für die Ausgangsvariable an.

Aufgabe 4.6
Eine kombinatorische Schaltung hat vier Eingänge *A(3:0),* die unten als Dezimalzahl *A* angegeben ist. Es gibt einen Ausgang *Y* der folgende Werte hat:

- 1 bei $A = 0, 1, 2, 3, 4, 7, 8, 9, 10, 11, 15$
- 0 sonst

a) Stellen Sie das Karnaugh-Diagramm auf.
b) Ermitteln Sie die Produktterme. Welche Produktterme sind erforderlich?
c) Geben Sie die Funktion für die Ausgangsvariable an.

Aufgabe 4.7
Eine kombinatorische Schaltung hat vier Eingänge *A(3:0),* die unten als Dezimalzahl *A* angegeben ist. Es gibt einen Ausgang *Y* der folgende Werte hat:

- 1 bei $A = 1, 3, 8, 11, 13, 14$
- 0 bei $A = 0, 2, 4, 5, 6$
- Don't-Care sonst

a) Stellen Sie das Karnaugh-Diagramm auf.
b) Ermitteln Sie die Produktterme mit Nutzung der undefinierten Ausgänge. Welche Produktterme sind erforderlich?
c) Geben Sie die Funktion für die Ausgangsvariable an.

Aufgabe 4.8
Eine kombinatorische Schaltung hat vier Eingänge *A(3:0),* die unten als Dezimalzahl *A* angegeben ist. Es gibt einen Ausgang *Y* der folgende Werte hat:

- 1 bei $A = 0, 5, 14, 15$
- 0 bei $A = 1, 2, 3, 6, 7, 8, 12, 13$
- Don't-Care sonst

a) Stellen Sie das Karnaugh-Diagramm auf.
b) Ermitteln Sie die Produktterme mit Nutzung der undefinierten Ausgänge. Welche Produktterme sind erforderlich?
c) Geben Sie die Funktion für die Ausgangsvariable an.

Sequenzielle Schaltungen

5

Während kombinatorische Schaltungen nur die aktuellen Werte der Eingangssignale verwenden, können sich *sequenzielle Schaltungen* Informationen merken. Die Ausgangswerte einer sequenziellen Schaltung können damit von den aktuellen und den vorangegangenen Werten der Eingangssignale abhängen. Dieses Gedächtnis wird durch Flip-Flops als Speicherelemente erreicht.

Beispielsweise kann der Kanal eines Fernsehers durch Zifferntasten sowie durch ‚+' und ‚−'-Taste ausgewählt werden. Durch Drücken der Taste ‚4' wird der Kanal Vier ausgewählt. Der Fernseher hat hierfür eine sequenzielle Schaltung, die sich den aktuellen Kanal merkt, auch wenn keine Taste mehr gedrückt ist. Durch Drücken von ‚+' wechselt der Fernseher auf Kanal Fünf. Wird ‚−' gedrückt, geht der Fernseher wieder auf Kanal Vier. Der Kanal kann also auf verschiedene Arten angewählt werden. Wie die Kanalauswahl erfolgte, ist nicht wichtig. Wenn der Kanal Vier gewählt ist, braucht sich die sequenzielle Schaltung nicht zu merken, ob dies durch die Taste ‚4' oder ‚−' oder ‚+' geschah.

Sequenzielle Schaltungen werden in der Digitaltechnik sehr oft eingesetzt und dabei meist durch einen Takt angesteuert. Dieser Takt erreicht für eine Hochleistungs-CPU Frequenzen von über 3 GHz, während für viele Anwendungen eine Geschwindigkeit im Bereich 10 bis 100 MHz ausreicht. Sequenzielle Schaltungen werden beispielsweise als Flankendetektor, als Zähler oder als Steuerung eingesetzt.

- Ein Flankendetektor erkennt die Änderung eines Eingangswertes und gibt einmalig ein Signal weiter. Wenn beim Fernseher die ‚+'-Taste gedrückt wird, soll nur ein Kanal weitergeschaltet werden, selbst wenn die Taste etwas länger gedrückt wird.
- Ein Zähler ist beispielsweise in einer CPU enthalten und zählt pro Takt jeweils einen Wert weiter, um den nächsten Befehl auszuführen. Bei einer Verzweigung kann der Zähler auch auf einen bestimmten Wert gesetzt werden.

In diesem und dem nächsten Kapitel werden einige Schaltungen ausführlich erläutert.

W. Gehrke et al., *Digitaltechnik,* Springer-Lehrbuch,
DOI 10.1007/978-3-662-49731-9_5

5.1 Speicherelemente

5.1.1 RS-Flip-Flop

Die Grundform eines Speicherelements ist das *RS-Flip-Flop* (RS-FF), auch als *Latch* bezeichnet. Es arbeitet ohne Takt und hat die beiden Eingänge *R* und *S* sowie den Ausgang *Q*. Das Schaltsymbol ist in Abb. 5.1 dargestellt.

5.1.1.1 Funktion

Die beiden Eingänge haben die Bedeutung Reset *(R)* und Set *(S),* also rücksetzen und setzen. Entsprechend dieser Namen ist auch die Funktion des RS-Flip-Flops.

- Mit *R* auf 1 wird der Ausgang *Q* auf 0 gesetzt (rücksetzen), *S* ist dabei 0.
- Mit *S* auf 1 wird der Ausgang *Q* auf 1 gesetzt (setzen), *R* ist dabei 0.
- Sind beide Eingänge 0, bleibt der Wert von *Q* unverändert (speichern).
- Beide Eingänge dürfen nicht gleichzeitig auf 1 sein. Man kann nicht gleichzeitig setzen und rücksetzen.

Der Zeitverlauf in Abb. 5.2 verdeutlicht die Funktion. In der Digitaltechnik wird der Zeitverlauf üblicherweise etwas vereinfacht dargestellt, da vor allem der logische Zusammenhang gezeigt werden soll. Auf der horizontalen Achse ist die Zeit aufgetragen. Die vertikale Achse zeigt die Pegel für die Eingangs- und Ausgangssignale. Die Zeitachse hat keinen Maßstab, da keine konkreten Zeiten sondern die Abläufe wichtig sind. Ebenso hat die vertikale Achse keinen Maßstab, sondern gibt nur die Pegel L und H für die Werte 0 und 1 an. Die Signalübergänge werden leicht schräg dargestellt, um den Übergang von 0 nach 1 oder umgekehrt anzudeuten. Die Zeitverzögerung, die in jeder Schaltung enthalten ist, wird dadurch angedeutet, dass die Signalübergänge von Eingang und Ausgang leicht versetzt sind.

Für das RS-FF sind in Abb. 5.2 die Eingänge *R* und *S* sowie der im Flip-Flop gespeicherte Ausgangswert *Q* dargestellt. Die eingezeichneten Zeitpunkte haben folgende Bedeutung:

1. Der Eingang *R* ist 1, das RS-FF wird rückgesetzt und *Q* ist 0.
2. Beide Eingänge sind 0 und das RS-FF speichert den vorherigen Wert 0 für *Q*.
3. S wird 1 und setzt das RS-FF. Der Ausgang *Q* wird 1 und speichert diesen Wert auch wenn *S* wieder auf 0 geht.
4. Mit Aktivierung von *R* wird das RS-FF wieder auf 0 gesetzt.

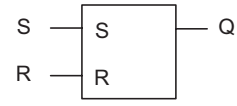


Abb. 5.1 Schaltsymbol eines RS-Flip-Flops (RS-FF)

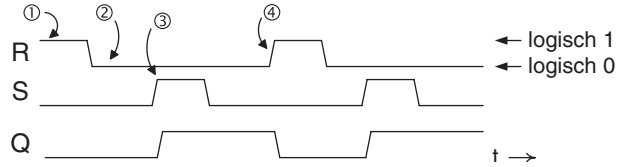


Abb. 5.2 Zeitverlauf der Ansteuerung eines RS-Flip-Flops

Beachten Sie: Wenn *R* und *S* 0 sind, kann der Ausgang sowohl 0 als auch 1 sein. Der Ausgangswert hängt also davon ab, ob zuletzt *R* oder *S* auf 1 war. Dies ist der wesentliche Unterschied zu einer kombinatorischen Schaltung, die bei gleichen Eingangswerten immer den gleichen Ausgangswert ergeben, unabhängig von vorherigen Werten.

5.1.1.2 Aufbau

Die Speicherung im RS-FF erfolgt durch eine *Rückkopplung* des Ausgangs *Q*. Es werden zwei NOR-Gatter benötigt, die wie in Abb. 5.3 verschaltet sind. Der Ausgang des zweiten NOR-Gatters wird an einen Eingang des ersten Gatters zurückgeführt und speichert so den Wert des Ausgangs *Q*. Da nur zwei Gatter benötigt werden, ist der Schaltungsaufwand für das RS-FF relativ klein.

Die NOR-Gatter des RS-FF können im Schaltplan auch nebeneinander geschoben werden, so dass sich die in Abb. 5.4 gezeigte Anordnung ergibt. Während ein NOR-Gatter den Ausgang *Q* erzeugt, hat das andere NOR-Gatter den invertierten Speicherwert als Ausgang.

5.1.1.3 Herleitung des Aufbaus

Der Aufbau des RS-Flip-Flops könnte auch mit den bereits bekannten Methoden aus dem vorherigen Kapitel hergeleitet werden. Abb. 5.5 zeigt, dass die Rückführung zur Speicherung des Flip-Flop-Wertes als separate Leitung angesehen werden kann. Der

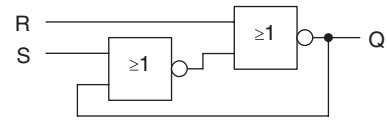


Abb. 5.3 Aufbau eines RS-Flip-Flops

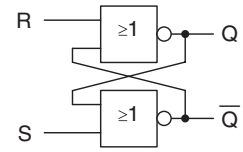


Abb. 5.4 Alternative Darstellung des RS-Flip-Flop-Aufbaus

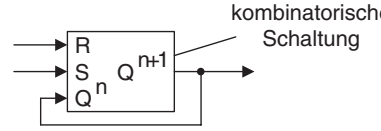


Abb. 5.5 Entwurf des RS-Flip-Flops

Rest des Flip-Flops ist dann eine normale kombinatorische Schaltung. Sie hat die Eingänge R und S sowie den alten Wert von Q, der hier als Q^n bezeichnet wird. Mit diesen drei Werten berechnet die kombinatorische Schaltung dann den neuen Wert von Q, bezeichnet als Q^{n+1}. Die Bezeichner n und $n+1$ stellen Zeitschritte dar; n ist der aktuelle, $n+1$ der nächste Wert.

Für die kombinatorische Schaltung aus Abb. 5.5 kann eine Funktionstabelle erstellt und mit dem Verfahren nach Karnaugh minimiert werden. Abb. 5.6 zeigt die Funktionstabelle und das Karnaugh-Diagramm dieser kombinatorischen Schaltung. Zur Minimierung können die disjunktive und die konjunktive Normalform verglichen werden, also Einsen oder Nullen zusammengefasst werden. Die konjunktive Normalform mit dem in Abb. 5.6 dargestellten Termen ergibt die Funktion

$$Q^{n+1} = \bar{R} \,\&\, (Q^n \vee S)$$

Mit dem De Morganschen Gesetz wird die UND-Verknüpfung durch eine NOR-Verknüpfung mit negierten Operatoren ersetzt. Aus dem ODER in der Klammer wird dann ein NOR und die Negierung von R entfällt. Somit ergibt sich die in Abb. 5.3 gezeigte Struktur mit zwei NOR-Gattern.

$$Q^{n+1} = \bar{R} \,\&\, (Q^n \vee S) = \overline{R \vee \overline{(Q^n \vee S)}}$$

5.1.1.4 Verwendung

In der Praxis wird das RS-Flip-Flop in der einfachen Grundform nur selten verwendet, da es kein Taktsignal benutzt. Es ist jedoch als Teilschaltung in getakteten Flip-Flops enthalten und dadurch eine wichtige Grundlage für die Datenspeicherung in sequenziellen Schaltungen.

5.1.2 Taktsteuerung von Flip-Flops

5.1.2.1 Takt

Fast alle in der Realität eingesetzten Schaltungen benutzen einen *Takt* zur Ansteuerung der Speicherelemente. Der Takt ist ein periodisches Signal, welches in gleichmäßigem Rhythmus zwischen 0 und 1 wechselt. Ein 0-1-Zyklus wird als *Taktzyklus, Taktschritt* oder *Taktperiode* bezeichnet.

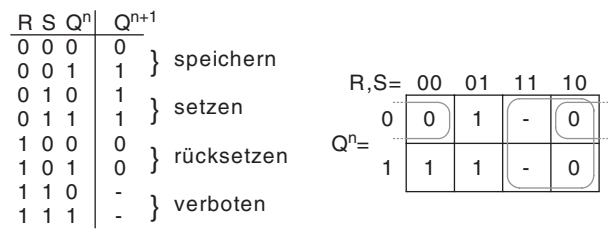


Abb. 5.6 Minimierung nach Karnaugh für den Entwurf des RS-FF

Der besondere Vorteil der Taktsteuerung ist die *Synchronisierung der Speicherelemente.* Durch den Takt schalten alle Flip-Flops gemeinsam und führen einen Rechenschritt aus. Mit dem nächsten Taktzyklus wird der nächste Rechenschritt ausgeführt.

Kennzeichnend für einen Takt sind die *Periodendauer* T_{per} und die *Taktfrequenz f,* die der Kehrwert der Periodendauer ist:

$$f = {}^{1}/_{T_{\text{per}}}$$

Taktfrequenzen für digitale Schaltungen sind typischerweise im Bereich zwischen 10 MHz für eine einfache Schaltung bis zu über 3 GHz für aktuelle CPUs in Computern. Die Periodendauer ergibt sich nach der genannten Formel.

Zur Verdeutlichung zwei Zahlenbeispiele:

- Für die Taktfrequenz 10 MHz beträgt die Periodendauer

$$T_{\text{per}} = {}^{1}/_{f} = {}^{1}/_{10MHz} = {}^{1}\Big/_{10 \cdot 10^{6}Hz} = {}^{1}/_{10^{7}Hz} = 10^{-7}s = 100 \cdot 10^{-9}s = 100\,ns$$

- Für die Taktfrequenz 3 GHz beträgt die Periodendauer

$$T_{\text{per}} = {}^{1}/_{3GHz} = 0,333\,ns$$

Je höher die Taktfrequenz ist, umso leistungsfähiger ist eine Schaltung. Allerdings steigen auch der Schaltungsaufwand, die Störanfälligkeit und die benötigte Leistung. Darum haben netzbetriebene stationäre Computer normalerweise höhere Taktraten als batteriebetriebene Laptops und Smartphones.

Eine weitere Kenngröße des Takts ist das *Tastverhältnis D* (englisch *Duty Cycle*), also die Dauer der 1-Phase bezogen auf die Periodendauer:

$$D = {}^{T_1}/_{T_{\text{per}}}$$

Der Duty Cycle sollte möglichst etwa 50 %, also 0- und 1-Phase etwa gleich lang sein. Dies ist insbesondere für hohe Taktfrequenzen wichtig, damit das Taktsignal ausreichend Zeit hat, auch wirklich die Low- und High-Pegel zu erreichen.

Abb. 5.7 zeigt den Taktverlauf eines Taktsignals mit Periodendauer und Zeiten für die Taktphasen. Die englische Bezeichnung für Takt ist *Clock;* das Signal wird daher oft als *CLK* oder *C* abgekürzt.

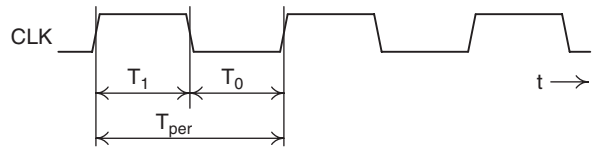


Abb. 5.7 Zeitverlauf eines Taktsignals

5.1.2.2 Taktpegelsteuerung

Als einfache Taktsteuerung kann der *Taktpegel,* also der Wert 0 oder 1, benutzt werden. Ein taktpegelgesteuertes Flip-Flop ist nur aktiv, wenn der Takt auf 1 ist. Die Grundform des RS-Flip-Flop kann mit wenig Aufwand um eine Taktpegelsteuerung erweitert werden. Wie in Abb. 5.8 gezeigt, werden dazu die Eingänge mit jeweils einem UND-Gatter erweitert. Nur wenn der Takt auf 1 ist, werden die beiden Steuereingänge *R* und *S* durch die UND-Funktion an *R** und *S** weitergegeben. Ist der Takt auf 0 sind auch *R** und *S** auf 0 und das RS-FF behält seinen Wert.

Der Zeitablauf in Abb. 5.9 verdeutlicht das Verhalten. Nur wenn der Takt *CLK* auf 1 ist, werden die Steuereingänge *R* und *S* ausgewertet. Dies sind die mit ✓ gekennzeichneten Impulse. Wenn der Takt auf 0 ist, führen an den mit × gekennzeichneten Zeiten die Eingangssignale zu keiner Änderung am Ausgang.

Die Taktpegelsteuerung hat jedoch einen großen Nachteil. Eigentlich sollte die Verarbeitung so ablaufen, dass pro Taktzyklus die Informationen genau ein Flip-Flop weitergegeben werden. Allerdings dauert die 1-Phase eine gewisse Zeit und die Flip-Flops sind während dieser 1-Phase aktiviert. Es wird also vorkommen, dass Informationen durch mehrere Flip-Flops „rutschen".

Um dies zu vermeiden, werden bei taktpegelgesteuerten Flip-Flops zwei Takte verwendet, die sich nicht überlappen. Dies ist in Abb. 5.10 dargestellt. Oben im Bild ist zu sehen, wie aufeinander folgende Flip-Flops abwechselnd an eines der Taktsignale angeschlossen werden. Unten ist der Zeitverlauf der beiden Takte skizziert. Immer abwechselnd, mit einer Pause dazwischen, ist ein Takt aktiv. Damit werden die Daten immer genau einen Schritt, also ein Flip-Flop weitergereicht.

Das Prinzip des Zweiphasentakts ähnelt einer Kanalschleuse, bei der ein Schiff durch zwei Tore fahren muss. Erst fährt das Schiff durch ein Tor und das Tor wird geschlossen.

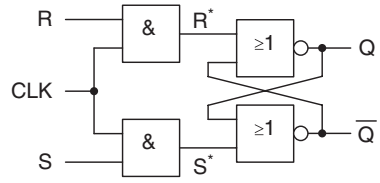


Abb. 5.8 Taktpegelgesteuertes RS-Flip-Flop

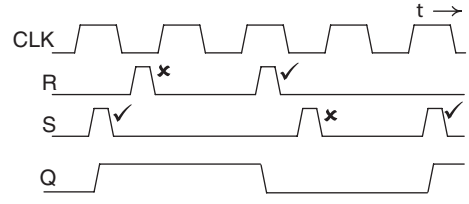


Abb. 5.9 Zeitverlauf beim taktpegelgesteuerten RS-Flip-Flop

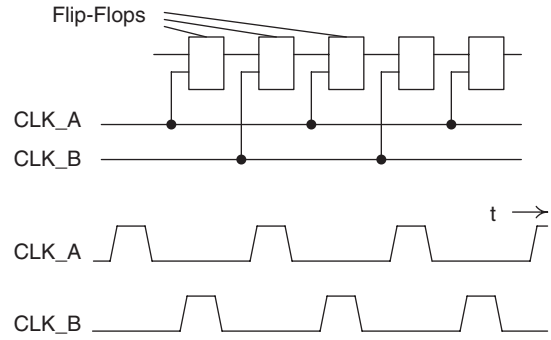


Abb. 5.10 Schaltungsprinzip und Zeitdiagramm eines Zweiphasentakts

Nach dem Ändern des Wasserstands wird das andere Tor geöffnet und das Schiff fährt weiter zur nächsten Schleuse. Es sind jedoch nie beide Tore gleichzeitig offen.

Ein solcher Zweiphasentakt mit taktpegelgesteuerten Flip-Flops wurde früher in vielen Schaltungen eingesetzt. Allerdings sind zwei Taktleitungen erforderlich, was einen höheren Aufwand bedeutet. Auch kann die Taktperiode nicht so gut ausgenutzt werden, sodass Zeit verloren geht. Darum werden heute kaum noch taktpegelgesteuerte Flip-Flops verwendet.

5.1.2.3 Taktflankensteuerung

Heutzutage wird praktisch immer eine *Taktflankensteuerung* verwendet. Nur bei einer Taktflanke ist das Flip-Flop aktiv, das heißt der Zeitpunkt des Schaltens ist sehr genau vorgegeben. Dies hat den Vorteil, dass alle Flip-Flops einer Schaltung wirklich gleichzeitig arbeiten können. Somit wird eine Verarbeitung immer genau einen Schritt von Flip-Flop zu Flip-Flop weitergeführt.

Für die Taktflankensteuerung kann entweder die steigende Taktflanke, also der Übergang von 0 nach 1, oder die fallende Taktflanke, also der Übergang von 1 nach 0, benutzt werden. Meist wird die steigende Taktflanke verwendet, da dies anschaulicher ist. Alle Flip-Flops einer Schaltung sind dann nur beim Übergang des Takts von 0 nach 1 aktiv. Genauso gut könnten auch Flip-Flops eingesetzt werden, die bei der fallenden Taktflanke aktiv sind. Dann sollten alle Flip-Flops der Schaltung so aufgebaut sein. Im Schaltsymbol wird die Taktflankensteuerung durch ein Dreieck am Takteingang dargestellt. Abb. 5.11 zeigt die Steuerung durch die Taktflanke und das Schaltsymbol.

Es gibt keine Flip-Flops, die bei beiden Flanken aktiv sind. Eine Mischung von Flip-Flops mit steigender und fallender Taktflanke wird nur bei Spezialschaltungen benötigt; ein Beispiel findet sich in einem späteren Kapitel bei der Ansteuerung von Speichern.

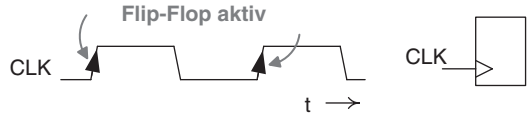


Abb. 5.11 Taktflankensteuerung und Schaltsymbol

Bei der Taktflankensteuerung erfolgt üblicherweise keine Ansteuerung mit *R* und *S* wie beim RS-Flip-Flop. Stattdessen gibt es einen Dateneingang D, dessen Wert direkt gespeichert wird. Dieses taktflankengesteuerte D-Flip-Flop wird im nächsten Abschnitt erläutert.

5.1.3 D-Flip-Flop

Das *taktflankengesteuerte D-Flip-Flop,* oder kurz *D-Flip-Flop (D-FF)* ist das heutzutage am häufigsten verwendete Flip-Flop. Wenn in der Praxis von einem Flip-Flop gesprochen wird, ist so gut wie immer das taktflankengesteuerte D-Flip-Flop gemeint. Zwei oder mehr D-FFs, die von einem gemeinsamen Takt angesteuert werden, bezeichnet man auch als *Register*.

5.1.3.1 Funktion

Beim D-Flip-Flop wird der Eingang *D* bei einer steigenden Flanke des Takts übernommen und am Ausgang *Q* ausgegeben. Das Schaltungssymbol in Abb. 5.12 zeigt auf der linken Seite den Dateneingang *D* und den Takteingang *C* mit dem Dreieck zur Kennzeichnung der Taktflankensteuerung. An der rechten Seite ist der Datenausgang *Q*. Wenn das D-FF auf die negative Taktflanke reagiert, wird dies durch einen Inverterkreis am Takteingang dargestellt. Im Symbol kennzeichnet die Ziffer 1 die Abhängigkeit der Signale voneinander. Der Dateneingang *1D* wird abhängig vom Taktsignal *C1* ausgewertet.

Das Verhalten des D-Flip-Flops wird durch die Funktionstabelle in Abb. 5.13 beschrieben. Die Form der Tabelle ist ähnlich zu den Funktionstabellen der kombinatorischen Schaltungen. Das Zeitverhalten wird durch das Taktflankensymbol und Indizes an den Werten beschrieben. Q^n meint dabei wieder den jetzigen Wert des Ausgangs *Q* und Q^{n+1} ist der zeitlich darauffolgende Wert. Die Indizes bezeichnen also aufeinanderfolgende Taktperioden oder Zeitschritte n und $n+1$.

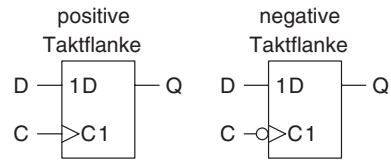


Abb. 5.12 Taktsymbol des D-FFs

positive Taktflanke

D	C	Q^{n+1}
0	↑	0
1	↑	1
X	0	Q^n
X	1	Q^n

negative Taktflanke

D	C	Q^{n+1}
0	↓	0
1	↓	1
X	0	Q^n
X	1	Q^n

Abb. 5.13 Funktionstabelle des D-FFs

Die Zeilen der linken Funktionstabelle (positive Taktflanke) haben die Bedeutung:

1. Bei D gleich 0 und positiver Taktflanke an C wird der Ausgang Q zu 0.
2. Bei D gleich 1 und positiver Taktflanke an C wird der Ausgang Q zu 1.
3. Wenn der Takt konstant auf 0 ist, behält Q seinen Wert. Der Wert von D ist irrelevant (‚X'). Das neue Q^{n+1} ist also gleich dem alten Q^n.
4. Wenn der Takt konstant auf 1 ist, behält Q seinen Wert. Q^{n+1} ist gleich Q^n.

Die rechte Funktionstabelle zeigt das entsprechende Verhalten für die negative Taktflanke.

Das Zeitverhalten des D-FFs zeigt Abb. 5.14. Bei jeder steigenden Taktflanke wird der Eingang von D übernommen und am Ausgang Q ausgegeben. Änderungen von D zwischen den steigenden Taktflanken haben keine Auswirkungen.

Die eingezeichneten Zeitpunkte haben folgende Bedeutung:

1. Der Eingang D wird 1.
2. Bei der nächsten steigenden Taktflanke speichert das D-Flip-Flop den Eingangswert und gibt ihn am Ausgang aus. Q wird 1.
3. Der Eingang D wird 0.
4. Bei der nächsten steigenden Taktflanke speichert das D-Flip-Flop wieder den Eingangswert. Q wird 0.
5. D wird 1 und vor der nächsten steigenden Taktflanke wieder 0. Der gespeicherte Wert im Flip-Flop und der Ausgang Q ändern sich nicht.
6. D wird wieder kurz 1, dann 0. Da in dieser Zeit eine steigende Taktflanke auftritt, wird der Ausgang für einen Takt gleich 1.

Das Zeitverhalten und auch alle weiteren Erklärungen sind im Folgenden nur für Flip-Flops mit positiver Taktflanke dargestellt. Flip-Flops mit negativer Taktflanke verhalten sich entsprechend.

5.1.3.2 Reales Zeitverhalten

Wie erläutert, übernimmt das D-Flip-Flop den Eingangswert bei der positiven Taktflanke. Natürlich braucht die Schaltung eine kurze Zeit, um den Wert zu übernehmen. Der Eingangswert darf sich darum zum Zeitpunkt der Taktflanke nicht ändern, sondern muss kurz vor und kurz nach der Taktflanke stabil sein. Abb. 5.15 zeigt einen zulässigen und unzulässigen Zeitverlauf.

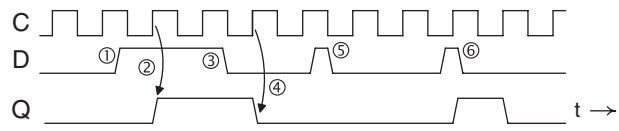


Abb. 5.14 Zeitverhalten eines D-Flip-Flops

1. Der Dateneingang *D* wechselt vor der Taktflanke.
2. Kurz vor und nach der Taktflanke ist *D* stabil und wird korrekt übernommen (✓).
3. Nach der Taktflanke kann *D* wieder wechseln.
4. Während der nächsten Taktflanke ist *D* nicht stabil und wird nicht korrekt übernommen (✗). Der Ausgang des Flip-Flops ist undefiniert. Er kann 0, 1 oder sogar einen unzulässigen Zwischenzustand haben.
5. Bei der nächsten Taktflanke ist *D* stabil. Dennoch kann das Flip-Flop einige Zeit benötigen, um sich zu „fangen". Dies wird als *Metastabilität* bezeichnet. Im Bild ist angenommen, dass der Ausgang bei dieser Taktflanke wieder normal den Eingangswert übernimmt.

Die benötigten Zeiten vor und nach der Taktflanke werden als *Setup-* und *Hold-Zeit* bezeichnet. Das Eingangssignal *D* muss vor der Taktflanke für die Setup-Zeit t_{setup} und nach der Taktflanke für die Hold-Zeit t_{hold} stabil sein.

Abb. 5.16 zeigt die Zeiten und verwendet die in der Digitaltechnik übliche Darstellung. Der horizontale Strich in der Mitte zwischen 0 und 1 gibt an, dass der Wert beliebig wechseln darf. Zwei parallele Striche bei 0 und 1 geben einen konstanten Wert 0 oder 1 an.

Die benötigten Zeiten von t_{setup} und t_{hold} hängen von der verwendeten Technologie ab und sind in Datenblättern angegeben. Bei modernen integrierten Schaltungen sind die Zeiten im Bereich von 0,1 ns oder kleiner. Die Hold-Zeit wird oft zu Null angestrebt, damit sich der Eingangswert direkt nach der Taktflanke ändern darf.

5.1.3.3 Aufbau

Für den Aufbau eines D-Flip-Flops gibt es mehrere Möglichkeiten, die sich in Größe, Zeitverhalten und Stromverbrauch unterscheiden. Abb. 5.17 zeigt eine Möglichkeit zum Aufbau eines D-Flip-Flops. Auf der rechten Seite ist ein RS-Flip-Flop zur Datenspeicherung (vgl. Abb. 5.3). Auf der linken Seite ist eine Vorstufe, in der sich ebenfalls die Struktur zweier RS-FFs findet. Diese Vorstufe erkennt die steigende Taktflanke und steuert dann das RS-FF auf der rechten Seite an.

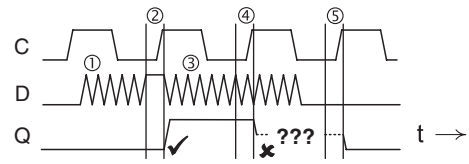


Abb. 5.15 Datenspeicherung bei Taktflanken

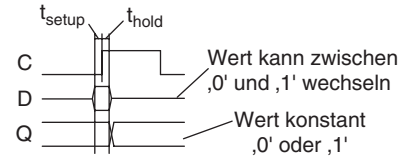


Abb. 5.16 Setup- und Hold-Zeiten beim D-FF

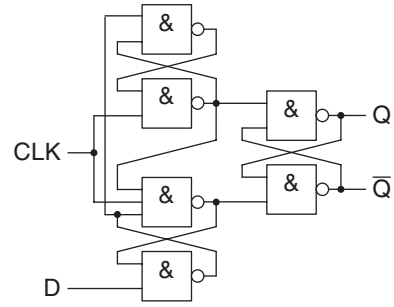


Abb. 5.17 Möglichkeit zum Aufbau eines D-Flip-Flops (nach Datenblatt TI SN7474)

Eine weitere Schaltung zur Implementierung eines Flip-Flops wird später im Kapitel Halbleitertechnik vorgestellt (Kapitel 10).

5.1.4 Erweiterung des D-Flip-Flops

Die Grundfunktion des D-Flip-Flops kann durch weitere Steuereingänge erweitert werden.

5.1.4.1 Asynchroner Reset und Set

Der Dateneingang des D-FF wird nur bei der Taktflanke ausgewertet. Manchmal ist es jedoch erforderlich, dass der Wert eines D-FFs sofort geändert wird. Hierzu dient ein *asynchroner Reset* oder *Set*. Der Begriff *asynchron* meint dabei „nicht synchron“, also „nicht mit dem Takt gekoppelt“. Normalerweise hat ein D-FF entweder Reset oder Set, je nachdem welchen Wert das D-FF bei Aktivierung einnehmen soll.

- Ein asynchroner Reset setzt das D-FF sofort auf 0.
- Ein asynchroner Set setzt das D-FF sofort auf 1.

Mit „sofort“ ist hierbei gemeint, dass nicht auf die nächste Taktflanke gewartet werden muss. Natürlich hat das Flip-Flop eine kurze Verzögerungszeit, in der die Gatter umschalten.

Reset und Set sind normale Eingänge des Flip-Flops und werden an der linken Kante des Schaltsymbols eingezeichnet (Abb. 5.18). Negative Polarität wird wieder durch den Inverterkreis symbolisiert. Abb. 5.18 zeigt beispielhaft den Set mit negativer Polarität. Genauso wäre ein Reset mit negativer Polarität möglich.

Das Zeitverhalten eines D-Flip-Flops mit asynchronen Reset zeigt Abb. 5.19. Bei den steigenden Taktflanken sind Hilfslinien eingezeichnet, um die Taktzyklen zu verdeutlichen.

1. Mit der steigenden Taktflanke wird der Wert 1 des Eingangs *D* gespeichert.
2. Durch eine 1 am Reset wird das D-FF sofort auf 0 gesetzt, also ohne auf eine Taktflanke zu warten.

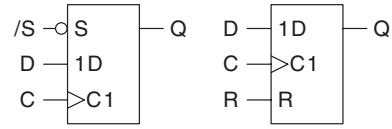


Abb. 5.18 Schaltsymbole von D-FFs mit asynchronem Set (hier mit negativer Polarität) und Reset

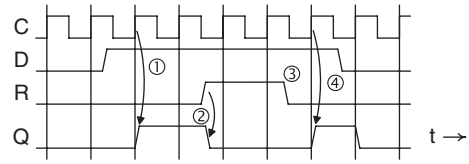


Abb. 5.19 Zeitverhalten eines D-Flip-Flops mit asynchronem Reset

3. Reset wird wieder 0, also inaktiv. Dies hat aber noch keine Auswirkung auf den gespeicherten Wert.
4. Erst mit der nächsten steigenden Taktflanke wird der Wert von *D* wieder ausgewertet und der Ausgang *Q* wird 1.

Beachten Sie insbesondere, dass nach dem Ende des Resets, zum Zeitpunkt ③ das Flip-Flop noch auf 0 bleibt. Der Eingang *D* ist synchron, wird also erst bei der nächsten steigenden Taktflanke wieder ausgewertet.

Praktische Verwendung finden asynchroner Reset und Set insbesondere bei der Initialisierung. Beim Einschalten einer Digitalschaltung haben die Flip-Flops einen unbekannten Speicherzustand und können durch Reset und Set auf den gewünschten Startwert gesetzt werden.

Auch für die Erkennung kurzer Impulse können asynchroner Reset und Set verwendet werden. Ein Eingangssignal ist eventuell sehr kurz und schon vor der nächsten Taktflanke beendet. Ein solcher Impuls würde von einer synchronen Schaltung, die nur bei den Taktflanken arbeitet, nicht erkannt. Zur Erkennung solcher Impulse wird ein Flip-Flop durch den Dateneingang ständig auf 0 gesetzt und der Impuls wird am asynchronen Set angeschlossen. Wenn das Flip-Flop auf 1 ist, lag ein Impuls am Set-Eingang vor.

5.1.4.2 Synchroner Reset und Set

Alternativ kann Reset und Set auch ganz normal mit der Taktflanke ausgewertet werden, also *synchron*. Wie in Abb. 5.20 gezeigt, hat der Steuereingang dann die Ziffer 1, als Kennzeichnung der Abhängigkeit vom Takt.

Der synchrone Set ist prinzipiell ein weiterer Dateneingang, das heißt, der Ausgang des Flip-Flops wird 1, wenn während der Taktflanke *D* oder *S* auf 1 sind. Deswegen könnte die Schaltung auch durch ein normales D-FF und ein ODER-Gatter implementiert werden (Abb. 5.20, rechts). Entwurf und Darstellung als synchroner Set sind jedoch übersichtlicher und der Set kann direkt in die Flip-Flop-Schaltung integriert werden.

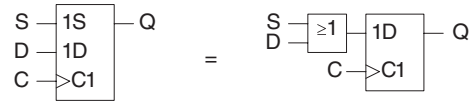


Abb. 5.20 Symbol und Schaltung eines D-FFs mit synchronem Set

In ähnlicher Weise gibt es D-FFs mit synchronem Reset. Auch synchroner Reset und Set werden für die Initialisierung von Digitalschaltungen verwendet.

5.1.4.3 Enable

Ein weiterer Steuereingang der für D-FFs verwendet wird, ist der *Enable-Eingang* (EN). Bei einer Taktflanke wird der D-Eingang nur übernommen, wenn Enable gleich 1 ist. Ansonsten wird der Ausgang Q^n beibehalten. Abb. 5.21 zeigt Symbol und Zeitverhalten, wobei die Ziffern wieder die Abhängigkeit anzeigen. Das Enable *EN1* gibt die Gültigkeit von Takt *1C2* an, welcher dann den Dateneingang *2D* übernimmt.

Im Zeitverhalten sind folgende Fälle gekennzeichnet:

1. *EN* ist 0 und das Flip-Flop behält seinen Wert.
2. *EN* ist 1 und bei jeder steigenden Taktflanke wird der Wert von *D* übernommen.
3. *EN* ist 0 und das Flip-Flop behält seinen Wert.

Ein Enable-Steuereingang wird in der Praxis eingesetzt, wenn eine Teilschaltung nur zu bestimmten Zeiten oder bei bestimmten Bedingungen aktiv ist.

5.1.4.4 Kompakte Darstellung von D-Flip-Flops

Für die Darstellung von D-Flip-Flops in einer größeren Schaltung wird in der Praxis häufig eine kompakte Form gewählt und die Ziffern der Eingangsabhängigkeit weggelassen (Abb. 5.22, links).

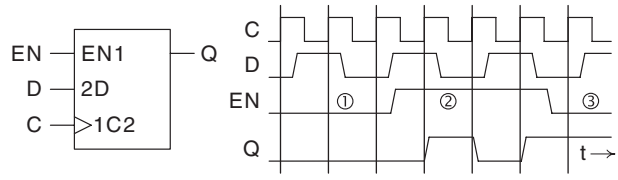


Abb. 5.21 Symbol und Zeitverhalten eines D-FF mit Enable

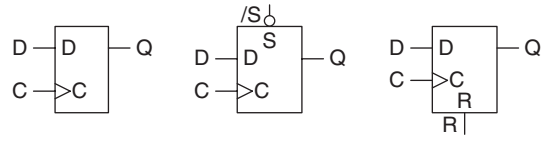


Abb. 5.22 Kompakte Darstellung eines D-FF in der Grundform sowie mit asynchronem Set und Reset

Asynchroner Set und Reset können dann an der unteren oder oberen Kante des Symbols eingezeichnet sein, um darzustellen, dass sie unabhängig vom Takteingang sind. Der Set liegt in dieser Darstellung an der oberen Kante, denn er zieht den Wert „nach oben“, zur 1. Reset wird entsprechend an der unteren Kante dargestellt, denn er zieht den Wert „nach unten“, zur 0. Abb. 5.22 zeigt auch diese Darstellung, wobei das Set wieder beispielhaft negative Polarität hat (vgl. Abb. 5.18).

5.1.5 Weitere Flip-Flops

Es gibt neben D-Flip-Flops und ihren Erweiterungen auch andere taktflankengesteuerte Flip-Flops. Diese werden allerdings in der Praxis nur selten eingesetzt und darum hier nur kurz erwähnt.

5.1.5.1 JK-Flip-Flop

Das *JK-Flip-Flop* (JK-FF) hat einen Takteingang und die beiden Steuereingänge *J* und *K*. Diese haben folgende Bedeutung:

- Beide Eingänge auf 0: Flip-Flop behält seinen Wert.
- *J* auf 1 (und *K* auf 0): Flip-Flop geht auf 1
- *K* auf 1 (und *J* auf 0): Flip-Flop geht auf 0
- Beide Eingänge auf 1: Flip-Flop invertiert seinen Wert, geht also von 0 auf 1 oder von 1 auf 0.

Dieses Verhalten ähnelt dem RS-FF, mit *J* als Set und *K* als Reset. Die Bedeutung kann man sich merken als *J* wie *Jump* (auf 1) und *K* wie *Kill* (auf 0). Die beim RS-FF verbotene Kombination, dass beide Steuereingänge auf 1 sind, ist hier erlaubt und dreht den gespeicherten Wert um.

Auch dieses Flip-Flop kann durch asynchronen Reset oder Set erweitert werden.

JK-Flip-Flops wurden früher eingesetzt, als Digitalschaltungen noch durch einzelne diskrete Bausteine aufgebaut wurden. Durch geschickte Ansteuerung von *J* und *K* konnten Logikgatter eingespart werden. Heutzutage werden praktisch keine diskreten Flip-Flops und darum auch keine JK-FFs mehr verwendet.

5.1.5.2 Toggle-Flip-Flop

Das *Toggle-Flip-Flop* (T-FF) hat, neben dem Takt, nur einen Steuereingang *T*. Wenn *T* gleich 1 ist, invertiert das Flip-Flop seinen Wert, es „toggled“. Bei T gleich 0 bleibt der gespeicherte Wert unverändert.

Auch das T-FF kann durch asynchronen Reset oder Set erweitert werden. Wie beim JK-FF wurde das T-FF eingesetzt, um durch geschickte Ansteuerung Logikgatter einzusparen. Es wird heutzutage praktisch nicht mehr verwendet.

5.1.6 Kippstufen

Flip-Flops werden auch als *bistabile Kippstufen* bezeichnet. Bistabil meint, dass beide „Kippwerte“, also 0 und 1 stabil sind. Diese Bezeichnung legt nahe, dass es auch andere Kippstufen gibt.

5.1.6.1 Monostabile Kippstufe

Eine *monostabile Kippstufe*, auch als *Monoflop* bezeichnet, hat nur einen stabilen Zustand; der instabile Zustand geht nach einer Verzögerungszeit in den stabilen Zustand über. Das Monoflop reagiert auf eine positive Taktflanke am Eingang mit einem 1-Impuls am Ausgang. Aus dieser instabilen Lage kippt es nach einer einstellbaren Zeit T_D zurück in den stabilen Zustand mit einer 0 am Ausgang. Erst wenn der Ausgang wieder in seinen ursprünglichen Logik-Zustand zurückgekippt ist, kann ein neuer Eingangsimpuls mit seiner Flanke wirksam werden.

Als Variante sind *nachtriggerbare* Monoflops möglich. Falls die Impulsdauer T_D noch nicht abgelaufen ist, verlängert eine Taktflanke des Eingangssignals den Impuls bis wiederum die Zeit T_D nach der Flanke abgelaufen ist.

Dieses Verhalten entspricht der Treppenhausbeleuchtung in einem Mehrfamilienhaus. Nach Schalterdruck ist das Licht für zwei Minuten an (instabiler Zustand) und geht danach wieder aus (stabiler Zustand). Bei einer nachtriggerbaren Treppenhausbeleuchtung verlängert ein weiterer Schalterdruck die Beleuchtungsdauer.

Monostabile Kippstufen sind als diskrete Bauelemente verfügbar. Die Verzögerungszeit kann über ein RC-Glied eingestellt werden. Eingesetzt werden diese Bauelemente, um das Zeitverhalten von Signalen zu kontrollieren. Beispielsweise kann so sichergestellt werden, dass ein Reset eine bestimmte Mindestdauer hat.

5.1.6.2 Astabile Kippstufe

Eine *astabile Kippstufe* hat keinen stabilen Zustand, sondern wechselt periodisch zwischen den beiden Zuständen, also 0 und 1. Sie wird auch als *Oszillator* bezeichnet und als Taktgenerator eingesetzt.

Es gibt verschiedene Schaltungen, die als astabile Kippstufe eingesetzt werden können. Einfache Schaltungen nutzen RC-Glieder, um zwischen den Zuständen umzuschalten. Hierbei ist die Frequenz meist nicht sehr stabil, aber für einfache Anwendungen kann dies ausreichend sein.

Für hohe Ansprüche in Hinblick auf Frequenzstabilität werden quarzgesteuerte Oszillatoren eingesetzt. Für den Einsatz in der Digitaltechnik stehen integrierte Schaltkreise zur Verfügung, die über einen Schwingquarz auf eine bestimmte Frequenz eingestellt werden.

5.2 Endliche Automaten

Eine sequenzielle Schaltung, die aus Speicherelementen und Logikgattern besteht, wird als Automat, oder genauer als *endlicher Automat* bezeichnet.

5.2.1 Automatentheorie

Ein Automat ist dadurch gekennzeichnet, dass sein Verhalten durch aktuelle Eingangsvariablen und interne Zustandsvariablen bestimmt ist. Die Zustandswerte, oder auch Zustände, beschreiben die „Vorgeschichte“ des Automaten. Daraus ergibt sich auch die englische Bezeichnung *Finite State Machine* (FSM), also frei übersetzt Automat mit endlicher Anzahl an Zuständen.

Vielleicht fragen Sie sich jetzt, ob es überhaupt Automaten mit unendlicher Anzahl an Zuständen gibt. Als reale Implementierung ist ein unendlich großer Speicher natürlich nicht möglich, aber in der Theorie ist dies denkbar. In der theoretischen Informatik wird die *Turingmaschine* verwendet, die einen unendlich großen Speicher hat und somit ein unendlicher Automat ist. Mit dem Gedankenmodell der Turingmaschine wird die Berechenbarkeit von mathematischen Problemen analysiert.

5.2.1.1 Mealy-Automat

Eine Grundform der endlichen Automaten ist der Mealy-Automat. Er wird durch drei Gruppen an Variablen und zwei Funktionen definiert.

Die drei Gruppen an Variablen sind:

- **Eingangsvariablen,** also Eingangswerte, die in die Schaltung hineingehen. Sie werden als *X(0), X(1), X(2), …* sowie gemeinsam als Gruppe *X* bezeichnet.
- **Ausgangsvariablen,** also Ausgangswerte, die aus der Schaltung herausgehen. Sie werden als *Y(0), Y(1), Y(2), …* sowie gemeinsam *Y* bezeichnet.
- **Zustandsvariablen,** also interne Werte der Schaltung, die den Zustand speichern. Sie werden als *Z(0), Z(1), Z(2), …* sowie gemeinsam Z bezeichnet.

Die zwei Funktionen beschreiben die Zusammenhänge zwischen den Variablen:

- Die *Zustandsübergangsfunktion* benutzt die Eingangsvariablen *X* und die aktuellen Zustandsvariablen Z^n, also Z vom aktuellen Zeitschritt *n.* Hiermit berechnet sie die neuen Zustandsvariablen Z^{n+1} für den nächsten Zeitschritt $n+1$. Als Funktion ausgedrückt lautet dies: $Z^{n+1}=f(X,Z^n)$
- Die *Ausgangsfunktion* benutzt ebenfalls die Eingangsvariablen *X* und die aktuellen Zustandsvariablen Z^n, um die Ausgangsvariablen *Y* zu berechnen. Die Funktion lautet: $Y=g(X,Z^n)$

Diese Struktur ist in Abb. 5.23 dargestellt. Eingangsvariable *X* und aktuelle Zustandsvariablen Z^n gehen in die Zustandsübergangsfunktion. Dieser Block ist eine kombinatorische Schaltung aus UND-Gattern, ODER-Gattern und so weiter. Sie berechnet den nächsten Zustand Z^{n+1}. Die Speicherglieder sind D-Flip-Flops, die zurzeit noch den aktuellen Zustand Z^n speichern und bei der Taktflanke den neuen Zustand übernehmen.

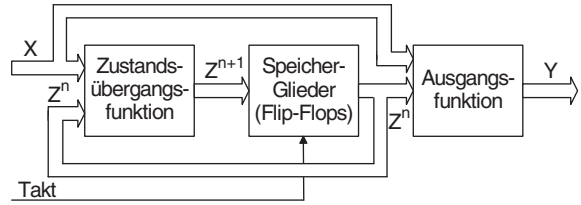


Abb. 5.23 Struktur des Mealy-Automaten

Die Ausgangsfunktion ist ebenfalls eine kombinatorische Schaltung und berechnet aus X und Z^n die Ausgangsvariablen Y.

Später in diesem Kapitel sind Beispiele für Automaten angegeben, um Struktur und Funktion des Mealy-Automaten zu verdeutlichen. Zunächst soll jedoch der andere bedeutende Automatentyp vorgestellt werden.

5.2.1.2 Moore-Automat

Der *Moore-Automat* ähnelt dem Mealy-Automat, hat jedoch einen wesentlichen Unterschied. Die Ausgangsfunktion hängt nur von den aktuellen Zustandsvariablen Z^n ab und nicht von den Eingangsvariablen X. Die Funktion für die Ausgangsvariablen Y lautet also: $Y=g(Z^n)$

Die Informationen der Eingangsvariablen beeinflussen also zunächst den Zustand und der Zustand bestimmt dann den Ausgang. Die Struktur ist in Abb. 5.24 zu sehen.

Verglichen mit dem Mealy-Automaten ist der Moore-Automat also etwas einfacher in der Struktur. Grundsätzlich können für praktische Problemstellungen stets beide Automaten verwendet werden. Für manche Problemstellungen ist ein Mealy-Automat besser geeignet, für andere ein Moore-Automat.

An den Beispielen, die später in diesem Kapitel folgen, werden die Unterschiede sowie Vor- und Nachteile deutlich.

5.2.1.3 Medwedew-Automat

Der *Medwedew-Automat* ist ein Spezialfall des Moore-Automaten. Bei ihm sind die Ausgangsvariablen Y gleich den Zustandsvariablen Z^n. Die Ausgangsfunktion ist also trivial und gibt die Zustandsvariablen direkt weiter. In der Funktionsschreibweise lautet dies: $Y=Z^n$

Auf den Medwedew-Automat wird später in Abschn. 5.2.7 kurz eingegangen.

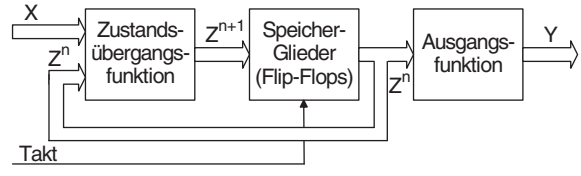


Abb. 5.24 Struktur des Moore-Automaten

5.2.2 Beispiel für einen Automaten

5.2.2.1 Schaltungsanalyse

Um die Funktionsweise eines Automaten zu verstehen, wird in diesem Abschnitt ein vorhandener Automat analysiert. Im darauffolgenden Abschnitt lernen Sie dann, wie Automaten entworfen werden.

Startpunkt der Analyse ist das Schaltbild des Automaten in Abb. 5.25. Vergleichen Sie ihn auch mit den Grundstrukturen von Mealy- und Moore-Automat in Abb. 5.23 und Abb. 5.24.

Im Schaltbild sind die drei Blöcke des Automaten hervorgehoben:

- Die Zustandsübergangsfunktion besteht aus fünf Logikgattern.
- Als Speicherglieder werden zwei D-Flip-Flops verwendet.
- Die Ausgangsfunktion besteht aus einem Logikgatter.

Die drei Variablengruppen des Automaten sind:

- Es gibt eine Eingangsvariable *X*
- Es gibt eine Ausgangsvariable *Y*
- Es gibt zwei Zustandsvariable *Z(0), Z(1)*

Außerdem ist das Taktsignal *CLK* vorhanden.

Eine Betrachtung der Struktur zeigt, dass es sich um einen Moore-Automaten handelt, denn der Ausgang *Y* hängt nur von den Zustandsvariablen und nicht auch noch von der Eingangsvariablen ab.

Zur weiteren Analyse werden die Funktionstabellen der beiden kombinatorischen Schaltungen für Zustandsübergangsfunktion und Ausgangsfunktion aufgestellt. Die Zustandsübergangsfunktion hat drei Eingänge, also müssen für $2^3 = 8$

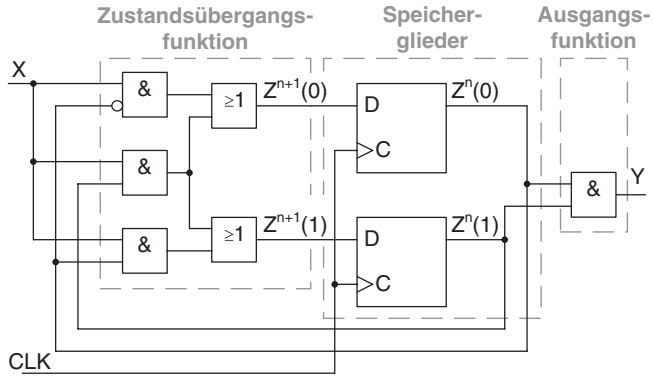


Abb. 5.25 Schaltbild eines Automaten

Eingangskombinationen die Funktionswerte ermittelt werden. Die Ausgangsfunktion hat zwei Eingänge, also $2^2 = 4$ Eingangskombinationen. Die Funktionstabellen werden direkt berechnet, indem alle Kombinationen in die Grafik oder die logische Funktion eingesetzt werden. Wenn Sie möchten, können Sie dies als Übung selbst berechnen, ansonsten finden Sie das Ergebnis in Abb. 5.26.

Beachten Sie die Unterscheidung für die Zustandsvariable *Z(0), Z(1)*. Die aktuellen Werte $Z^n(0)$, $Z^n(1)$ sind *Eingänge* für beide Funktionstabellen. Die Werte $Z^{n+1}(0)$, $Z^{n+1}(1)$ für den nächsten Zeitschritt sind die *Ausgabe* der Zustandsübergangsfunktion.

5.2.2.2 Zustände und Zustandsfolgetabelle

Da der Automat zwei Zustandsvariable hat, können vier verschiedene Zustände gespeichert werden. Zur besseren Anschaulichkeit werden diese Zustände durch Buchstaben *A, B, C, D* gekennzeichnet. Als allgemeine Bezeichnung für Zustände wird der Buchstabe *s* (engl. *State*) verwendet. Die Zuordnung zwischen Zustandsvariablen und Zuständen zeigt Abb. 5.27.

Jetzt können Zustandsübergangsfunktion und Ausgangsfunktion mit der Codierung der Zustände kombiniert werden. In Tabelle Abb. 5.26 werden also *Z(0)* und *Z(1)* durch die Zustandsnamen *A, B, C, D* aus Abb. 5.27 ersetzt. Das Ergebnis wird als *Zustandsfolgetabelle* (Abb. 5.28) bezeichnet. Die acht Zeilen der Zustandsübergangsfunktion (Abb. 5.26) sind umsortiert, so dass die Zustände in vier Zeilen und die Eingangsvariable in zwei Spalten angeordnet sind.

In der Zustandsfolgetabelle Abb. 5.28 steht links der aktuelle Zustand s^n. Auf der rechten Seite ist für die beiden Möglichkeiten der Eingangsvariablen der jeweilige Folgezustand s^{n+1} angegeben. Ganz rechts findet sich die Ausgangsvariable Y. Wie oben gesagt, ergibt sich Abb. 5.28 direkt aus den Funktionstabellen und der Zustandscodierung. Zum Nachvollziehen können Sie als Übung die Zustandsfolgetabelle selbst noch einmal erstellen.

Abb. 5.26 Funktionstabellen für Zustandsübergangsfunktion (links) und Ausgangsfunktion (rechts)

X	$Z^n(1)$	$Z^n(0)$	$Z^{n+1}(1)$	$Z^{n+1}(0)$
0	0	0	0	0
0	0	1	0	0
0	1	0	0	0
0	1	1	0	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	1
1	1	1	1	1

$Z^n(1)$	$Z^n(0)$	Y
0	0	0
0	1	0
1	0	0
1	1	1

Abb. 5.27 Codierung der Zustände

Codierung		Zustand
Z(1)	Z(0)	s
0	0	A
0	1	B
1	0	C
1	1	D

Abb. 5.28 Zustandsfolgetabelle

s^n	s^{n+1}		
	X=0	X=1	Y
A	A	B	0
B	A	C	0
C	A	D	0
D	A	D	1

Die Übergänge zwischen den Zuständen lassen sich auch grafisch darstellen. Hierzu dient das *Zustandsfolgediagramm* in Abb. 5.29. Die Zustände sind als Kreise angegeben und enthalten auch die Ausgabewerte der jeweiligen Zustände. Die Übergänge zwischen den Zuständen sind Pfeile. Bei jeder steigenden Taktflanke geht der Automat einen Übergang, also einen Pfeil weiter. Am Pfeil steht jeweils die Bedingung, bei der der Übergang erfolgt, also $X = 0$ oder $X = 1$.

Da es für X zwei Möglichkeiten gibt, gibt es für jeden Zustand zwei mögliche Folgezustände. Dabei ist es auch möglich, dass ein Zustand sein eigener Folgezustand ist. Jeder Zustand ist Startpunkt für genau zwei Pfeile. Für die Endpunkte der Pfeile gibt es keine Beschränkung. Manche Zustände können nur von einem Pfeil, also einem Übergang erreicht werden. Andere Zustände können das Ziel von mehreren Zustandsübergängen sein.

5.2.2.3 Funktion

Durch das Zustandsfolgediagramm oder vielleicht bereits durch die Zustandsfolgetabelle wird die Funktion des Automaten deutlich. Der Automat erkennt Folgen von 1 am Eingang X. Wenn der Eingang das dritte Mal 1 ist, wird auch der Ausgang 1 und bleibt 1 so lange weiter eine 1 am Eingang anliegt. Wenn eine 0 am Eingang anliegt, geht der Ausgang auf 0 und es müssen wieder drei Werte mit 1 anliegen, damit der Ausgang 1 wird. Wenn nach zweimal 1 bereits eine 0 am Eingang X anliegt, beginnt das Zählen wieder von neuem; es muss wieder dreimal eine 1 auftreten.

Dieses Verhalten wird durch die Zustände wie folgt umgesetzt. Vergleichen Sie zur Beschreibung die Zustandsfolgetabelle (Abb. 5.28) und das Zustandsfolgediagramm (Abb. 5.29).

- Bei einer 0 am Eingang geht der Automat in den Zustand *A*. Dieser Zustand hat also die Bedeutung: „Der letzte Eingangswert war 0."
- Bei der ersten 1 geht der Automat in den Zustand *B*. Dieser Zustand hat die Bedeutung: „Es gab bisher eine 1."

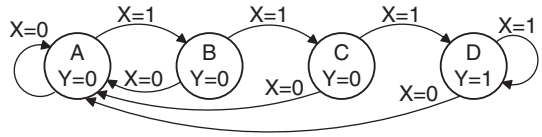


Abb. 5.29 Zustandsfolgediagramm

- Wenn im Zustand *B* eine 0 anliegt, muss wieder von vorne gestartet werden und der Automat geht nach *A*. Eine 1 im Zustand *B* wäre jedoch die zweite 1 und der Automat geht in den Zustand *C* mit der Bedeutung: „Es gab bisher zweimal eine 1."
- Eine weitere 1 wäre die dritte 1 und dies soll der Automat ja erkennen. Dann geht der Automat in den Zustand *D* und gibt am Ausgang eine 1 aus.
- Bei jeder weiteren 1 bleibt der Automat in *D* und gibt weiter 1 aus. Der Zustand *D* hat also die Bedeutung: „Drei oder mehr Eingangswerte nacheinander waren 1."

Wie Sie aus der Beschreibung erkennen, hat also jeder Zustand eine bestimmte Bedeutung.

> **Zustand:** Der Zustand speichert Informationen aus der Vergangenheit, die für die Funktion erforderlich sind.

Abb. 5.30 zeigt das Zeitverhalten des Automaten beispielhaft für einen Zeitverlauf am Eingang *X*. Das Eingangssignal wird jeweils bei der steigenden Taktflanke ausgewertet und daraus ergeben sich der Zustand und das Ausgabesignal *Y* für den jeweiligen Taktzyklus.

In praktischen Anwendungen arbeiten fast alle Schaltungen mit einem Taktsignal. Deshalb verwenden auch alle Automaten, die in diesem Buch beschrieben sind, einen Takt und die Informationen am Eingang eines Automaten werden immer nur bei der steigenden Taktflanke ausgewertet. Die Beschreibung „Der Eingang *X* war dreimal 1." meint daher eigentlich „Der Eingang *X* war bei drei steigenden Taktflanke auf 1."

5.2.3 Entwurf von Automaten

Normalerweise ist in der Praxis der Ablauf umgekehrt zu dem zuvor erläuterten Beispiel. Bei einer Entwicklung ist meist eine Aufgabe gegeben und hierzu soll eine Schaltung entworfen werden. Der Ablauf beim Entwurf umfasst die folgenden Schritte:

1. Spezifikation des Verhaltens
2. Aufstellen der Zustandsfolgetabelle
3. Minimierung der Zustände
4. Codierung der Zustände
5. Aufstellen der Ansteuerungstabelle
6. Logikminimierung

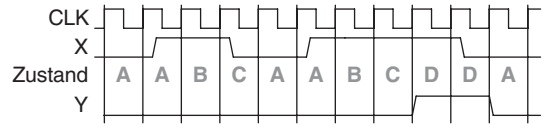


Abb. 5.30 Zeitdiagramm für den analysierten Automaten

5.2.3.1 Spezifikation des Verhaltens

Das gewünschte Verhalten eines Automaten ist meist in Textform gegeben. Ein einfacher Automat kann in einem Absatz beschrieben werden. Für eine komplexe Schaltung, z. B. einen Mikroprozessor, kann die *Spezifikation* aber auch mehrere 100 Seiten Umfang haben. Gerade bei größeren Spezifikationen können Unklarheiten auftreten, zum Beispiel weil nicht alle möglichen Fälle des Eingangsverhaltens spezifiziert sind. Diese Unklarheiten müssen dann während des Entwurfs durch Rückfragen bei den Verantwortlichen für die Spezifikation geklärt werden.

In diesem Unterkapitel soll eine Schaltung mit folgender Spezifikation entworfen werden:

> Zum Entprellen eines Tasters soll ein Automat entwickelt werden. Der Automat soll am Ausgang *Y* den entprellten Wert des Eingangs *X* angeben. Wenn am Eingang drei Takte lang der gleiche Wert 0 oder 1 anliegt, soll der Ausgang *Y* diesen Wert annehmen. Ansonsten soll der letzte Eingangswert, der mindestens drei Takte anlag ausgegeben werden.
> Beim Einschalten soll der Wert 0 ausgegeben werden.

Ein Zeitdiagramm kann die Spezifikation ergänzen. Zeitdiagramme sind dabei aber nur Beispiel und dienen der Illustration einer Spezifikation. Sie sind kein Ersatz für eine Spezifikation, denn die Angabe aller möglichen Abfolgen von Eingangskombinationen und Zuständen ist in einem Zeitdiagramm meist gar nicht möglich. Das Zeitdiagramm des Entprell-Automaten in Abb. 5.31 zeigt die Reaktion auf eine exemplarische Eingabe.

5.2.3.2 Aufstellen der Zustandsfolgetabelle

Das Aufstellen der Zustandsfolgetabelle ist der eigentliche kreative Schritt bei der Entwicklung eines Automaten. Am übersichtlichsten und einfachsten ist die grafische Darstellung als Zustandsfolgediagramm und spätere Abschrift als Tabelle.

Als Erstes muss entschieden werden, ob eine Implementierung als Mealy- oder Moore-Automat erfolgen soll. Bei Übungsaufgaben ist normalerweise der Typ vorgegeben. Hier soll ein Moore-Automat erstellt werden. Wenn Sie mehrere Automaten entworfen haben, können Sie selbst beurteilen, welcher Automatentyp günstiger ist.

Das Zustandsfolgediagramm wird schrittweise erstellt und dieser Entwurf soll hier auch in einzelnen Schritten erklärt werden, damit Sie die Vorgehensweise nachvollziehen können.

Schritt 1

Um einen Anfang für das Diagramm zu haben, wird mit einem ersten Zustand begonnen. In diesem Beispiel wird der Fall betrachtet, dass die Eingabe immer 0 ist. In diesem Fall ist auch die Ausgabe 0 und der Automat bleibt immer im gleichen Zustand.

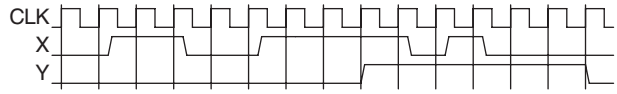


Abb. 5.31 Zeitdiagramm für Entprell-Automat

Abb. 5.32 zeigt den ersten Zustand. Um die Bedeutung anzudeuten, hat er den Namen „stabil 0“. Zunächst wird ja nur der Fall betrachtet, dass der Eingang stets 0 ist, so dass auch nur ein Übergangspfeil eingetragen wird. Er führt wieder auf den Zustand „stabil 0“. Die Ausgabe des Zustands ist 0.

Dieser Zustand ist auch der Startzustand, denn laut Spezifikation soll beim Einschalten der Wert 0 ausgegeben werden. Dies wird durch einen Pfeil mit „Reset“ gekennzeichnet.

Schritt 2

Der Automat wird jetzt schrittweise erweitert. Als nächster Schritt wird angenommen, dass der Eingang auf 1 wechselt und dann auf diesem Wert bleibt. Der Automat muss mitzählen, wie oft der Eingang 1 ist. Dieses Mitzählen erfolgt durch die unterschiedlichen Zustände, denn bei jedem Takt geht der Automat ja einen Übergang, also einen Pfeil weiter.

Die ersten beiden Male darf er laut Spezifikation noch nicht reagieren. Erst beim dritten Mal wird der Wechsel auf 1 akzeptiert und auch die Ausgabe geht auf 1.

Dieses Verhalten wird, wie in Abb. 5.33 zu sehen, durch drei neue Zustände erreicht:

- Bei der ersten 1 merkt sich ein Zustand, dass einmal eine 1 aufgetreten ist. Dieser Zustand wird als „1-mal 1“ bezeichnet. Er hat noch die Ausgabe Y=0, da erst nach drei Takten ein Wechsel akzeptiert werden soll.
- Mit der zweiten 1 wird der Zustand „2-mal 1“ erreicht.

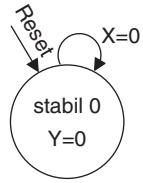


Abb. 5.32 Zustandsfolgediagramm des Entprell-Automaten – Schritt 1

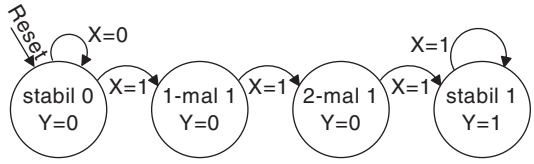


Abb. 5.33 Zustandsfolgediagramm des Entprell-Automaten – Schritt 2

- Mit der dritten 1 akzeptiert der Automat, dass der neue Wert lange genug aufgetreten ist und jetzt stabil anliegt. Der neue Zustand „stabil 1“ hat die Ausgabe 1.

  Wenn der Eingang danach weiterhin 1 ist, bleibt der Automat im Zustand „stabil 1“.

Schritt 3
Als weiterer Schritt kann der Weg von der Ausgabe 1 zurück zu 0 eingetragen werden. Es wird angenommen, dass der Eingang jetzt wieder auf 0 wechselt und dort bleibt. Das Verhalten des Automaten ist ähnlich wie in Schritt 2, so dass jetzt zwei neue Zustände „1-mal 0“ und „2-mal 0“ eingetragen werden (Abb. 5.34). Danach wechselt der Automat wieder in den zuerst eingetragenen Zustand „stabil 0“, ganz links.

Schritt 4
Als letzter Schritt wird überprüft, ob alle Übergänge für die Zustände eingetragen sind. Bei n Eingangsvariablen hat jeder Zustand 2^n Möglichkeiten für Folgezustände. Es müssen also prinzipiell 2^n Pfeile vorhanden sein, wobei auch mehrere Pfeile auf den gleichen Folgezustand führen können.

Der hier betrachtete Automat hat eine Eingangsvariable *X,* mit zwei möglichen Werten 0 und 1. Darum muss jeder Zustand zwei Übergänge, also zwei Pfeile haben. Hierzu müssen noch einige Pfeile eingetragen werden.

- Wenn bei „1-mal 1“ der Eingang *X* auf 0 ist, wird das Zählen der 1-Werte abgebrochen und der Automat geht wieder auf den Zustand „stabil 0“.
- Auch bei „2-mal 1” ist für *X* gleich 0 die erforderliche Anzahl von drei 1-Werten nicht erreicht. Der Automat geht auf “stabil 0”.
- Bei „1-mal 0“ fehlt der Übergang für *X* gleich 1. In diesem Fall geht der Automat auf „stabil 1“.
- Bei „2-mal 0“ ist für *X* gleich 1 der Folgezustand ebenfalls „stabil 1“.

Abb. 5.35 zeigt den kompletten Automaten. Alle Zustände haben zwei Folgezustände, so dass keine Übergänge fehlen.

Die Aufteilung in vier Schritte ergibt sich hier durch die Überlegungen zu den Teilfunktionen des Automaten. Bei anderen Aufgabenstellungen können mehr oder weniger Schritte sinnvoll sein.

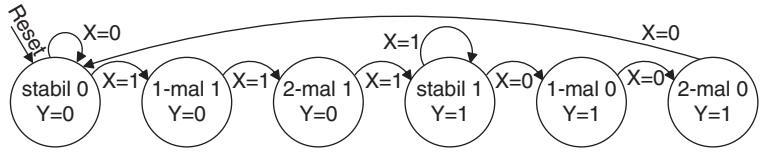


Abb. 5.34 Zustandsfolgediagramm des Entprell-Automaten – Schritt 3

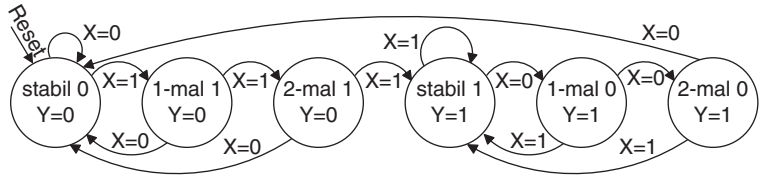


Abb. 5.35 Zustandsfolgediagramm des Entprell-Automaten – Schritt 4

Aufstellen der Zustandsfolgetabelle

Aus dem Zustandsfolgediagramm kann jetzt als textuelle Form die Zustandsfolgetabelle erstellt werden. Dazu wird für jeden Zustand eine Zeile und für jede mögliche Eingangskombination eine Spalte angelegt. In diese Felder wird für jede Kombination aus Eingangswerten und Zustand der Folgezustand eingetragen.

Außerdem erhalten die Ausgangswerte eine Spalte.

Die Zustandsfolgetabelle des Automaten in Abb. 5.36 benötigt also sechs Zeilen für die sechs Zustände. In zwei Spalten werden die Folgezustände für $X = 0$ und $X = 1$ eingetragen; eine dritte Spalte gibt den Wert des Ausgangs Y an. In die Felder werden die Informationen des Zustandsfolgediagramms (Abb. 5.35) eingetragen. Der Startzustand wird mit einem Stern gekennzeichnet. Das Aufstellen der Tabelle ist eher formell, die kreative Arbeit wurde bei der Erstellung des Diagramms geleistet. Natürlich sollte noch einmal die Plausibilität des Automaten überprüft werden, also ob für jeden möglichen Fall auch ein Folgezustand definiert wurde.

5.2.3.3 Minimierung der Zustände

In diesem Schritt wird geprüft, ob die Anzahl der Zustände reduziert werden kann, oder ob die Anzahl bereits minimal ist. Eine Vereinfachung ist möglich, wenn *äquivalente* (also gleichbedeutende) Zustände zusammengefasst werden können. Zwei Zustände sind äquivalent, wenn für alle Eingangskombinationen die Folgezustände gleich oder äquivalent sind und außerdem die Ausgangswerte gleich sind.

s^n	s^{n+1}		Y
	X = 0	X = 1	
stabil 0*	stabil 0	1-mal 1	0
1-mal 1	stabil 0	2-mal 1	0
2-mal 1	stabil 0	stabil 1	0
stabil 1	1-mal 0	stabil 1	1
1-mal 0	2-mal 0	stabil 1	1
2-mal 0	stabil 0	stabil 1	1

* = Reset

Abb. 5.36 Zustandsfolgetabelle des Entprell-Automaten

Der Entprell-Automat ist minimal, benötigt also mindestens sechs Zustände, denn:

- Die drei linken und die drei rechten Zustände in Abb. 5.35 haben unterschiedliche Ausgaben.
- Die Folgezustände sind nicht gleich. Für die drei linken Zustände führt $X = 0$ zwar immer nach „stabil 0“. Für $X = 1$ sind jedoch unterschiedliche Folgezustände vorhanden. Ähnliches gilt für die drei rechten Zustände.

Es gibt Algorithmen, mit denen äquivalente Zustände gefunden und der Automat minimiert werden können. In der Praxis werden diese Algorithmen aus zwei Gründen jedoch selten verwendet. Zum einen können durch Betrachten eines Automaten recht gut äquivalente Zustände identifiziert werden. Zum anderen wird akzeptiert, wenn ein oder zwei Zustände zu viel vorhanden sind, solange die Struktur des Automaten verständlich bleibt.

Beispiel für die Minimierung von Zuständen

Unnötige Zustände entstehen, wenn im Zustandsfolgediagramm ein neuer Zustand erstellt wurde, obwohl ein bereits vorhandener Zustand genutzt werden könnte. Schauen Sie sich dazu noch einmal Schritt 3 der Erstellung des Zustandsfolgediagramms in Abb. 5.34 an. Hier fehlt noch der Fall, dass bei „1-mal 1“, „2-mal 1“ eine 0 auftritt, ein Wechsel also nur einen oder zwei Takte lang ist. Ähnliches gilt für „1-mal 0“, „2-mal 0“.

Man könnte jetzt für diese fehlenden Übergänge zwei neue Zustände erstellen, und zwar „bleib 0“ und „bleib 1“. Dies wäre nicht nötig, denn die Übergänge könnten nach „stabil 0“ und „stabil 1“ gehen. Aber eventuell wird dies bei der Erstellung des Automaten nicht erkannt.

Von den Zuständen „bleib 0“ und „bleib 1“ gehen die Übergänge auf sich selbst sowie auf „1-mal 1“ beziehungsweise „1-mal 0“. Es entsteht das Diagramm in Abb. 5.37. Dieses Zustandsfolgediagramm ist ein korrekter Automat, entsprechend der Spezifikation, aber er ist nicht minimal, denn er verwendet acht statt der erforderlichen sechs Zustände.

Zur Minimierung des Automaten in Abb. 5.37 können „bleib 0“ und „stabil 0“ zusammengefasst werden. Sie sind äquivalent, denn:

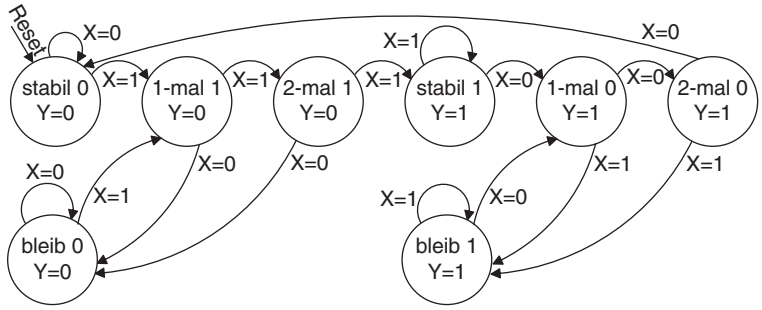


Abb. 5.37 Nicht minimales Zustandsfolgediagramm des Entprell-Automaten

- Beide Zustände haben die gleichen Folgezustände, nämlich sich selbst für $X = 0$ und „1-mal 1“ für $X = 1$.
- Beide Zustände geben $Y = 0$ aus.

Gleiches gilt für „bleib 1“ und „stabil 1“.

Damit ergibt sich wieder der minimale Automat aus Abb. 5.35. Beide Automaten, also Abb. 5.37 und 5.35 sind äquivalent, denn sie ergeben für gleiche Eingaben auch die gleiche Ausgabe. Von außen, also ohne Sichtbarkeit des aktuellen Zustands, sind die Automaten nicht zu unterscheiden.

5.2.3.4 Codierung der Zustände

Als nächster Entwurfsschritt wird für die Zustände des Automaten eine *Zustandscodierung* bestimmt. Es muss also festgelegt werden, welche 0-1-Kombinationen für die Zustände gelten. Die Codewortlänge n muss so gewählt werden, dass alle m Zustände dargestellt werden können. Mathematisch ausgedrückt muss also gelten:

$$2^n \geq m$$

Aufgelöst nach der Codewortlänge n ergibt sich folgende Formel, bei der *ld* den Zweierlogarithmus bezeichnet:

$$n \geq ld\, m$$

Das Beispiel dieses Kapitels hat $m = 6$ Zustände, also ist $n \geq ld\, 6 = 2{,}58$ als Codewortlänge nötig. Da nur ganzzahlige Werte möglich sind, muss n mindestens 3 sein. Mit der Zweierpotenz kann man ähnlich rechnen. Für $n = 3$ gilt $2^3 = 8 \geq 6$. Die Gegenprobe für $n = 2$ zeigt, dass die kleinere Codewortlänge von zwei nicht möglich ist: $2^2 = 4 < 6$. Da die Zweierpotenzen für kleine Zahlen recht einfach zu merken sind, ist diese Rechenweise meist einfacher als der Logarithmus.

Tipp zur Berechnung: Taschenrechner haben normalerweise keine Taste für den Zweierlogarithmus. Der Wert kann berechnet werden, als Zehnerlogarithmus einer Zahl geteilt durch Zehnerlogarithmus von zwei:

$$ld\, m = \log\, m\, /\, \log\, 2$$

Für $m = 6$ lautet die Rechnung:

$$ld\, 6 = \log\, 6 / \log\, 2 = 0,778 / 0,301 = 2,58$$

Ziel der Zustandscodierung ist ein möglichst geringer Aufwand, eine möglichst hohe Taktgeschwindigkeit oder eine Kombination aus diesen beiden Anforderungen. Für die Codierung gibt es prinzipiell sehr viele Möglichkeiten, sodass diese nicht alle ausprobiert werden können. Es gibt darum verschiedene Strategien, die im Abschn. 5.2.4 noch erläutert werden.

In der Praxis wird oft eine einfache Zuordnung gewählt und das soll auch für das hier betrachtete Beispiel so erfolgen. Als Codierung werden die Zustände entsprechend der

Abb. 5.38 Codierung des Entprell-Automaten mit minimaler Codewortlänge

s^n	Z(2:0)
stabil 0	000
1-mal 1	001
2-mal 1	010
stabil 1	011
1-mal 0	100
2-mal 0	101

s^n	Z^n	Z^{n+1}		Y
		X = 0	X = 1	
stabil 0*	000*	000	001	0
1-mal 1	001	000	010	0
2-mal 1	010	000	011	0
stabil 1	011	100	011	1
1-mal 0	100	101	011	1
2-mal 0	101	000	011	1
-	110	- - -	- - -	-
-	111	- - -	- - -	-

Abb. 5.39 Ansteuerungstabelle des Entprell-Automaten

Dualzahlen durchnummeriert. Der Automat hat 6 Zustände, die entsprechend der Tabelle in Abb. 5.38 mit dem Codewort *Z(2:0)* codiert werden. Da die Anzahl der Zustände keine Zweierpotenz ist, sind einige Codewörter unbenutzt, hier sind das die Codierungen 110 und 111.

5.2.3.5 Aufstellen der Ansteuerungstabelle

Mit der gewählten Codierung kann jetzt die Funktionstabelle für die kombinatorischen Schaltungen im Automat erstellt werden. In der Zustandsfolgetabelle werden also die Namen der Zustände durch die Codierung ersetzt. Diese neue Tabelle wird als Ansteuerungstabelle bezeichnet.

Abb. 5.39 zeigt die Ansteuerungstabelle für die Codierung aus Abb. 5.38. Eine Besonderheit sind die beiden unbenutzten Codierungen für die keine Folgezustände und Ausgabewerte definiert sind. Für sie werden Don't-Care-Werte eingetragen.

Für sicherheitskritische Schaltungen kann für die unbenutzten Codierungen auch ein bestimmter Folgezustand gewählt werden. Falls die Schaltung durch eine Störung, beispielsweise einen Spannungseinbruch, in einen undefinierten Zustand gerät, wird somit im Folgeschritt wieder ein gültiger Zustand erreicht.

5.2.3.6 Logikminimierung

Aus der Ansteuerungstabelle können jetzt die Logikfunktionen durch Minimierung, also mit Karnaugh-Diagramm ermittelt werden. Dies sind insgesamt vier Karnaugh-Diagramme für Ausgangswert *Y* und die drei neuen Zustandsvariable $Z^{n+1}(2:0)$. Die

Diagramme haben vier Eingangswerte, nämlich Eingangsvariable X und drei Zustandsvariable $Z^n(2:0)$. Da bei dem Moore-Automaten die Ausgabe unabhängig vom Eingang ist, hat das Karnaugh-Diagramm für Y nur drei Eingangswerte $Z^n(2:0)$.

Auf die Darstellung der Karnaugh-Diagramme wird hier verzichtet. Die minimierten Funktionen sind:

$$Z^{n+1}(2) = \bar{X} \,\&\, Z^n(1) \,\&\, Z^n(0) \vee \bar{X} \,\&\, Z^n(2) \,\&\, \overline{Z^n(0)}$$

$$Z^{n+1}(1) = X \,\&\, Z^n(2) \vee X \,\&\, Z^n(1) \vee X \,\&\, Z^n(0)$$

$$Z^{n+1}(0) = X \,\&\, Z^n(2) \vee X \,\&\, Z^n(1) \vee X \,\&\, \overline{Z^n(0)} \vee Z^n(2) \,\&\, \overline{Z^n(0)}$$

$$Y = Z^n(2) \vee Z^n(1) \,\&\, Z^n(0)$$

Mit diesen Funktionen ergibt sich für den Automaten das Schaltbild aus Abb. 5.40. Es enthält drei Flip-Flops für die Zustandsvariablen sowie ein Dutzend Logik-Gatter für Zustandsübergangsfunktion und Ausgangsfunktion. Der Startzustand „stabil 0“ hat die Codierung 000. Darum wird der Reset so geschaltet, dass alle Flip-Flops auf 0 gesetzt werden.

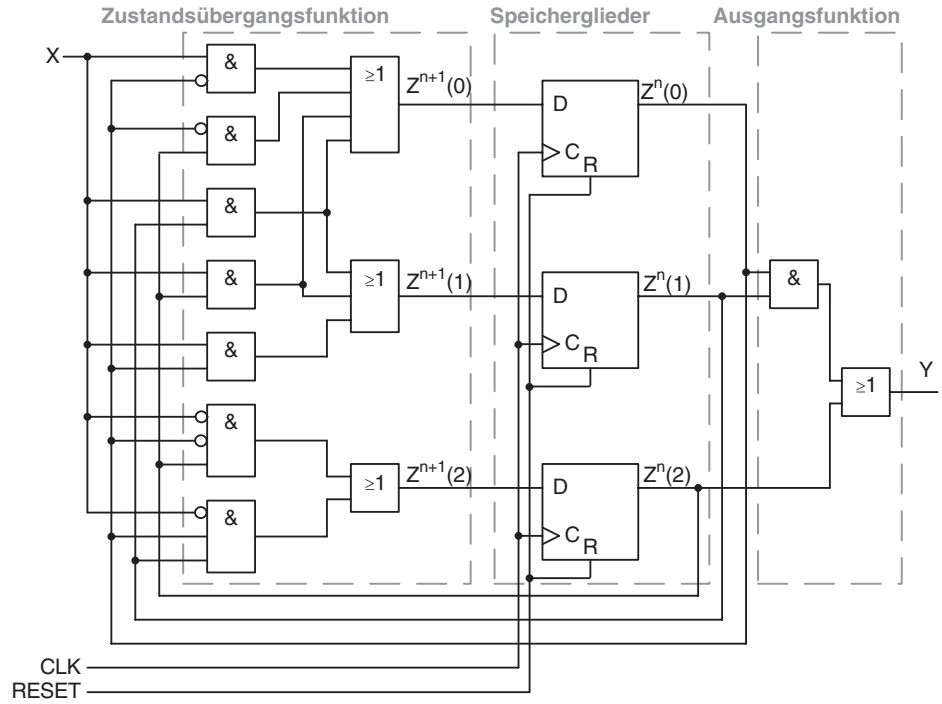


Abb. 5.40 Schaltbild des Entprell-Automaten

Damit ist der Automat komplett entworfen. In der Praxis würde nun die Dokumentation folgen, die ein Nachvollziehen des Schaltungsentwurfs ermöglicht. Außerdem werden durch eine Dokumentation spätere Modifikationen vereinfacht, die sich eventuell durch eine geänderte Spezifikation ergeben.

5.2.4 Codierung von Zuständen

Für die *Codierung der Zustände* gibt es verschiedene Strategien. Wichtiges Unterscheidungsmerkmal ist die *Codewortlänge*.

5.2.4.1 Codierung mit minimaler Codewortlänge

Die *Codierung mit minimaler Codewortlänge* wurde im vorstehenden Beispiel bereits verwendet. Bei der Zuordnung von Zuständen und Codewörtern gibt es mehrere Möglichkeiten. Theoretisch könnte man hier verschiedene Codierungen ausprobieren, um zu versuchen, möglichst einfache kombinatorische Schaltungen zu erhalten.

In der Praxis wird meist eine einfache Zuordnung gewählt, beispielsweise das oben verwendete Durchnummerieren der Zustände entsprechend der Dualzahlen. Der Aufwand zum kompletten Ausprobieren verschiedener Möglichkeiten ist meist zu hoch.

5.2.4.2 Codierung mit redundanter Codewortlänge

Eine andere Strategie zur Codierung benutzt mehr Stellen des Codewortes als eigentlich erforderlich wären. Die Codewortlänge ist also *redundant* und erfordert mehr Flip-Flops als bei minimaler Codewortlänge. Dies erscheint zunächst nicht sinnvoll, allerdings werden oft die kombinatorischen Schaltungen für Zustandsübergangsfunktion und Ausgangsfunktion einfacher und schneller.

Häufig verwendete Codes sind die *One-Hot-Codierung* sowie die *Zero-One-Hot-Codierung*. Die One-Hot-Codierung ist ein 1-aus-n-Code, das heißt von den n Stellen des Codeworts ist genau eine Stelle 1 (also „Hot“), die anderen sind 0. Die Anzahl der möglichen Codewörter ist genauso groß wie die Codewortlänge.

Die Zero-One-Hot-Codierung ist eine Variante, bei der zusätzlich das Codewort mit nur 0-Stellen erlaubt ist. Bei n Stellen sind also $n+1$ Codewörter möglich.

Der Entprell-Automaten aus Abschn. 5.2.3 hat 6 Zustände, so dass eine One-Hot-Codierung die Codewortlänge 6 hat. Die Zero-One-Hot-Codierung ergibt die Codewortlänge 5. Eine Zuordnung von Codierung und Zuständen ist in Abb. 5.41 angegeben.

Zum Vergleich der Codierungen soll der Entprell-Automat auch mit der One-Hot-Codierung implementiert werden. Genau wie im vorherigen Abschnitt wird in die Zustandsfolgetabelle die Codierung eingesetzt, so dass sich die Ansteuerungstabelle in Abb. 5.42 ergibt. Durch die Codewortlänge 6 sind 2^6=64 Codierungen möglich, von denen 6 benutzt sind. Die 58 unbenutzten Codierungen haben Don’t-Care als Folgezustand und Ausgabe und können somit zur Optimierung benutzt werden.

Abb. 5.41 Codierung des Entprell-Automaten mit redundanter Codewortlänge

	„One-Hot“	„Zero-One-Hot“
s^n	Z (5:0)	Z (4:0)
stabil 0	000001	00000
1-mal 1	000010	00001
2-mal 1	000100	00010
stabil 1	001000	00100
1-mal 0	010000	01000
2-mal 0	100000	10000

s^n	$Z^n(5:0)$	$Z^{n+1}(5:0)$		Y
		X=0	X=1	
stabil 0*	000001*	000001	000010	0
1-mal 1	000010	000001	000100	0
2-mal 1	000100	000001	001000	0
stabil 1	001000	010000	001000	1
1-mal 0	010000	100000	001000	1
2-mal 0	100000	000001	001000	1
	sonst	------	------	-

* = Reset

Abb. 5.42 Ansteuerungstabelle des Entprell-Automaten für One-Hot-Codierung

Aus der Ansteuerungstabelle werden wiederum die Logikfunktionen durch Minimierung erstellt. Für den Folgezustand sind sieben Eingangswerte zu beachten, nämlich sechs aktuelle Zustandsvariable sowie der Eingangswert X. Für den Ausgang sind es die sechs aktuellen Zustandsvariablen. Dies ist für ein Karnaugh-Diagramm zu unübersichtlich, sodass eine rechnergestützte Minimierung durchgeführt wird. Das Ergebnis lautet:

$$Z^{n+1}(5) = \bar{X} \,\&\, Z^n(4)$$

$$Z^{n+1}(4) = \bar{X} \,\&\, Z^n(3)$$

$$Z^{n+1}(3) = X \,\&\, \overline{Z^n(1)} \,\&\, \overline{Z^n(0)}$$

$$Z^{n+1}(2) = X \,\&\, Z^n(1)$$

$$Z^{n+1}(1) = X \,\&\, Z^n(0)$$

$$Z^{n+1}(0) = \bar{X} \,\&\, \overline{Z^n(4)} \,\&\, \overline{Z^n(3)}$$

$$Y = \overline{Z^n(2)} \,\&\, \overline{Z^n(1)} \,\&\, \overline{Z^n(0)}$$

Zwei Dinge fallen bei den Gleichungen auf:

- Die Zustandsvariable $Z^n(5)$ wird nicht verwendet. Es sind also nur fünf Stellen des Codeworts und damit auch nur fünf Flip-Flops nötig. Damit wird die Codierung zu einer Zero-One-Hot-Codierung, allerdings mit anderer Zuordnung als in Abb. 5.41.

- Die Logik-Funktionen sind deutlich einfacher als bei der Variante mit minimaler Codewortlänge. Es wird jeweils nur ein UND-Gatter mit zwei oder drei Eingängen benötigt. Die Informationen müssen nur durch eine Stufe an Logikgattern, wodurch die Schaltung prinzipiell schneller ist.

Die Schaltung des Automaten mit One-Hot-Codierung ist in Abb. 5.43 dargestellt. Für den Startzustand (vgl. Abb. 5.42) muss $Z^n(0)$ auf 1, die anderen Zustandsvariablen auf 0 gesetzt werden.

Auch im optischen Vergleich zu Abb. 5.40 wird sichtbar, dass die One-Hot-Codierung einen Nachteil durch zusätzliche Flip-Flops und Vorteile durch weniger Logikgatter und nur eine Logikstufe hat.

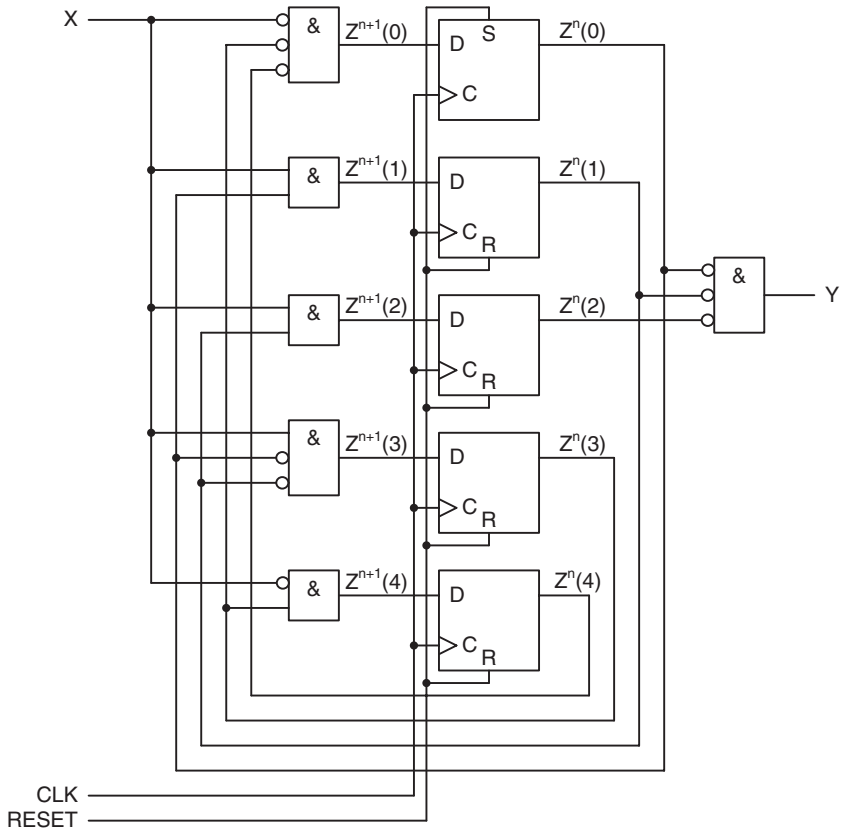


Abb. 5.43 Schaltbild des Entprell-Automaten mit One-Hot-Codierung

5.2.4.3 Optimierte Codierung

Um verschiedene Codierungen zu vergleichen, wird eine weitere Variante vorgestellt. Es handelt sich um eine Codierung bei der die Code-Zuordnung optimiert wird. Dazu wird die Zustandsfolgetabelle (vgl. Abb. 5.36) genauer betrachtet. Wie in Abb. 5.44 verdeutlicht, fallen zwei Dinge auf:

1. Drei der Zustände können nur bei X=0 als Folgezustand auftreten, die drei anderen Zustände nur bei X=1.
2. Für drei Zustände ist die Ausgabe Y= 0, für die drei anderen Zustände ist Y= 1.

Diese beiden Eigenschaften können ausgenutzt werden, um die Codierung möglichst einfach zu wählen.

1. Eine Zustandsvariable *Z(0)* wird entsprechend des Folgezustands gewählt. Das heißt, die Zustände, die Folgezustand bei X=0 sind, werden auch mit *Z(0)*=0 codiert. Die anderen Zustände, die Folgezustand bei X=1 sind, werden mit *Z(0)*=1 codiert.
2. Eine Zustandsvariable Z*(1)* wird entsprechend des Ausgabewertes gewählt. Das heißt, die Zustände mit Ausgangswert Y=0, werden mit *Z(1)*=0 codiert, die Zustände mit Y=1, haben *Z(1)*=1 als Code.

Weitere Zustandsvariable werden ohne besondere Zuordnung gewählt. Dabei muss beachtet werden, dass alle Zustände unterschiedliche Codierungen bekommen. Für die 6 Zustände des Entprell-Automaten ist eine dritte Zustandsvariable *Z(2)* erforderlich. Die Codierung hat hier minimale Codewortlänge; dies ist jedoch keine zwingende Bedingung für eine optimierte Codierung.

Der gewählte Code ist in Abb. 5.45 dargestellt. Die Codierungen 100 und 011 werden nicht verwendet.

Die Ansteuerungstabelle und die Logikfunktionen werden hier nicht gezeigt, sondern direkt das Schaltbild des Automaten mit optimierter Codierung in Abb. 5.46. Die beiden Optimierungen sind direkt im Schaltbild zu erkennen. Da *Z(0)* entsprechend des Folgezustands gewählt ist, wird direkt der Eingang X ohne weitere Verarbeitung gespeichert. Und da *Y(1)* entsprechend des Ausgangs ist, kann diese Zustandsvariable direkt als Ausgang Y verwendet werden.

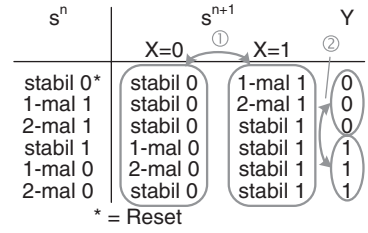


Abb. 5.44 Analyse der Zustandsfolgetabelle zur Optimierung

Abb. 5.45 Codierung des Entprell-Automaten mit optimierter Codierung

s^n	Z(2:0)	
stabil 0	0 0 0	① Zustände, die bei X=‚1' folgen
1-mal 1	0 0 1	
2-mal 1	1 0 1	
stabil 1	1 1 1	
1-mal 0	0 1 0	② Zustände mit Ausgabe Y=‚1'
2-mal 0	1 1 0	

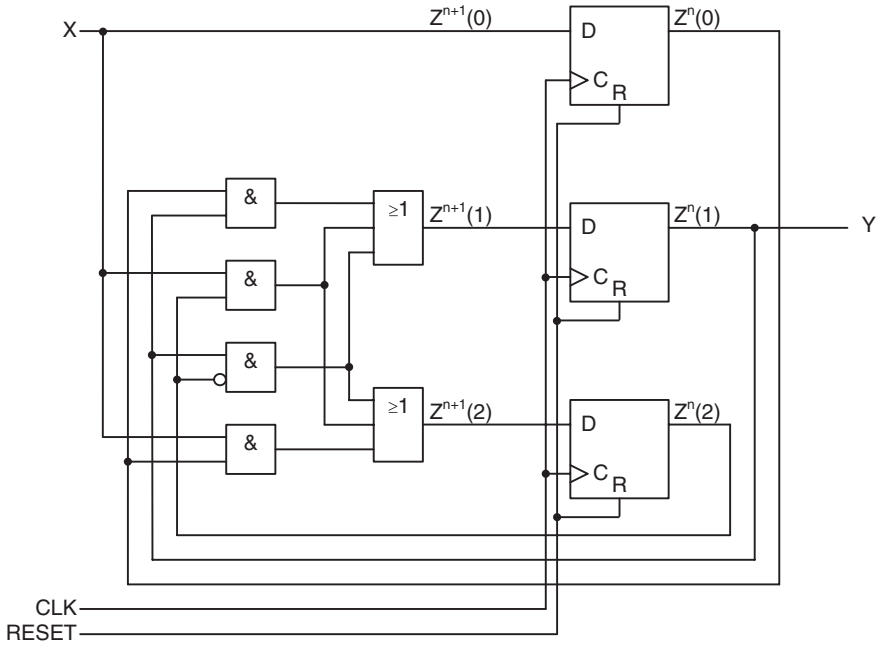


Abb. 5.46 Schaltbild des Entprell-Automaten mit optimierter Codierung

Auch für andere Automaten können oft optimierte Codierungen entsprechend der Ausgabe oder der Folgezustände gefunden werden.

5.2.4.4 Vergleich der Codierungen

Um die Codierung der Zustände und die Struktur des Automaten besser zu verstehen, sollen hier noch einmal die drei Varianten verglichen werden:

- Codierung mit minimaler Codewortlänge und einfacher Durchnummerierung der Zustände, Abb. 5.40
- Codierung mit redundanter Codewortlänge und One-Hot-Codierung, Abb. 5.43
- Codierung mit optimierter Zustandscodierung durch Analyse der Zustandsfolgetabelle, Abb. 5.46

Zunächst ist wichtig zu sagen, dass alle Automaten *äquivalent* sind. Das heißt, sie ergeben bei gleicher Eingabe auch die gleiche Ausgabe. Damit sind sie in ihrem logischen Verhalten von außen nicht zu unterscheiden.

Ob allgemein eine Codierung mit minimaler oder redundanter Codewortlänge die geeignete Schaltung ergibt, hängt von der Struktur des Automaten, den Anforderungen und der Technologie der Schaltungsimplementierung ab. Es handelt sich um Strategien, die bei der Schaltungsoptimierung probiert werden können.

In der Praxis muss der Aufwand für eine Optimierung und der erzielte Nutzen beachtet werden. Die Arbeitszeit, die für eine optimale Zustandscodierung erforderlich ist, lohnt sich meist nicht, denn in einer sehr großen Schaltung werden nur einige Logikgatter gespart.

Der Schaltungsentwurf erfolgt heutzutage mit Computer-Unterstützung. In Abschn. 5.3 wird erläutert, wie die Zustandsfolgetabelle in VHDL umgesetzt werden kann. Die Codierung der Zustände und Berechnung der Logikfunktionen erfolgt durch den Computer, der eine Codierung mit minimaler oder redundanter Wortlänge wählt oder beide Möglichkeiten ausprobiert. Die Optimierung der Zustandscodierung erfolgt also durch den Rechner. Sie sollten die Rückmeldungen des Computers verstehen (z. B. „Choosing One-Hot-Coding“).

5.2.5 Entwurf von Mealy-Automaten

Der Entwurf eines *Mealy-Automaten* gleicht in weiten Teilen dem eines Moore-Automaten.

5.2.5.1 Unterschied zum Moore-Automaten

Der wesentliche Unterschied beim Mealy-Automaten ist, dass die Ausgabe nicht von den Zuständen sondern den Zustandsübergängen abhängt. Das bedeutet, im Zustandsfolgediagramm wird die Ausgabe nicht in die Zustandskreise, sondern an den Pfeilen der Zustandsübergänge eingetragen (Abb. 5.47).

Dieser Unterschied kommt daher, dass beim Mealy-Automat die Ausgabe ja auch von den aktuellen Eingangswerten und nicht nur vom Zustand abhängt. Auch in der

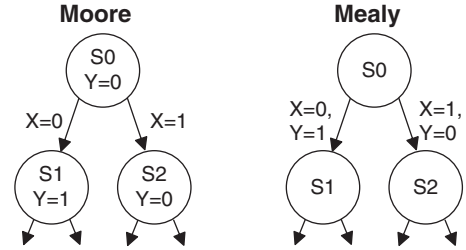


Abb. 5.47 Vergleich der Zustandsfolgediagramme für Moore- und Mealy-Automat

Abb. 5.48 Vergleich der Zustandsfolgetabellen für Moore- und Mealy-Automat

Moore

s^n	s^{n+1}		Y
	X=0	X=1	
S0	S1	S2	0
S1	S3	S4	1
...	...	...	...

Mealy

s^n	s^{n+1},Y	
	X=0	X=1
S0	S1,1	S2,0
S1	S3,0	S4,1
...	...	...

Zustandsfolgetabelle ist dann die Ausgabe abhängig von Zustand und Eingang und wird nicht einmal pro Zustand, sondern für jede Eingangsspalte angegeben (Abb. 5.48).

Diese Unterschiede erscheinen zunächst etwas formell. Sie eröffnen jedoch weitere Möglichkeiten für den Entwurf eines Automaten. Um dies zu verdeutlichen, wird im folgenden Beispiel ein Mealy-Automat entworfen.

5.2.5.2 Beispiel für einen Mealy-Automaten

Am Anfang des Entwurfs steht wieder eine Spezifikation des Verhaltens. Als Beispiel soll ein Mealy-Automat mit folgender Spezifikation entworfen werden:

> Ein Automat soll die Anzahl von Takten mit dem Wert 1 halbieren. Wenn am Eingang *X* der Wert 1 anliegt, soll für jeden zweiten Wert eine 1, ansonsten eine 0 am Ausgang *Y* ausgegeben werden. Die Zählung soll durch Eingangswerte 0 nicht beeinflusst werden. Bei einer 0 am Eingang soll 0 ausgegeben werden.
> Beim Einschalten soll für die erste 1 der Wert 0 ausgegeben werden.

Auch hier wird die Spezifikation durch ein Zeitdiagramm ergänzt (Abb. 5.49). Der Wert von *X* wird immer bei der steigenden Taktflanke ausgewertet. Der erste Impuls mit 1 wird unterdrückt, der zweite Impuls führt zur Ausgabe 1. Wenn *X* dauerhaft auf 1 ist, führt dies zu einer 0-1-Folge an *Y*.

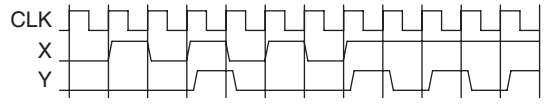
5.2.5.3 Aufstellen der Zustandsfolgetabelle

Um den Automat zu entwerfen, wird zunächst überlegt, welche Informationen sich der Automat merken muss. Der Automat gibt nur jede zweite 1 am Eingang weiter und unterdrückt die jeweils andere 1. Er muss sich also merken, ob die nächste 1 weitergegeben oder unterdrückt wird. Mit dieser Grundidee an Zuständen wird der Automat wieder grafisch, durch das Aufstellen des Zustandsfolgediagramms entworfen.

Schritt 1

Es wird mit zwei Zuständen entsprechend obiger Überlegung gestartet (Abb. 5.50). Sie erhalten den Namen „next-0“ und „next-1“ mit der Bedeutung:

Abb. 5.49 Zeitdiagramm für Halbieren der 1-Werte



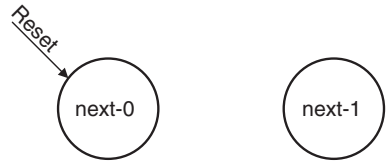


Abb. 5.50 Zustandsfolgediagramm zum Halbieren der 1-Werte – Schritt 1

- next-0: Die nächste 1 am Eingang wird unterdrückt. Dies ist laut Spezifikation der Startzustand.
- next-1: Die vorherige 1 wurde unterdrückt, also wird die nächste 1 des Eingangs an den Ausgang weitergegeben.

Wie in Abb. 5.50 zu sehen, ist für die Zustände keine Ausgabe definiert, da ein Mealy-Automat entworfen wird.

Schritt 2
Für die beiden Zustände wird nun überlegt, was laut Spezifikation im Falle der Eingaben X=0 und X=1 passieren muss.

- Für X=0 wird eine 0 ausgegeben. Das Zählen der 1-Werte wird nicht beeinflusst, darum ändert der Automat seinen Zustand nicht.
- Für X=1 sind zwei Fälle möglich:
  - Im Zustand next-0 wird die 1 unterdrückt, also eine 0 ausgegeben. Der Automat merkt sich, dass die nächste 1 weitergegeben wird, wechselt also nach next-1.
  - Im Zustand next-1 wird die 1 weitergegeben, also eine 1 ausgegeben. Der Automat merkt sich, dass die nächste 1 wieder unterdrückt wird, wechselt also nach next-0.

Damit sind für alle Zustände beide mögliche Folgezustände definiert und das Zustandsfolgediagramm in Abb. 5.51 ist komplett. Es werden zwei Zustände benötigt, die sich nicht zusammenfassen lassen.

Der Unterschied zum Moore-Automaten zeigt sich in der Definition der Ausgangswerte. Beim Mealy-Automat in Abb. 5.51 sind die Ausgänge für die Zustandsübergänge,

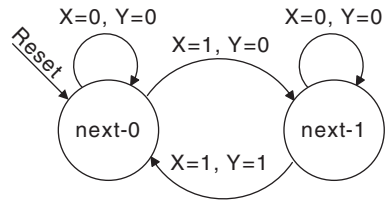


Abb. 5.51 Zustandsfolgediagramm zum Halbieren der 1-Werte – Schritt 2

also für die Pfeile definiert. Beim Moore-Automat in Abb. 5.35 sind die Ausgänge für die Zustände, also die Kreise definiert.

Zustandsfolgetabelle

Die Zustandsfolgetabelle (Abb. 5.52) kann direkt aus dem Diagramm erstellt werden. Wie erläutert ist die Ausgabe abhängig von Zustand und Eingang. Darum wird sie zusammen mit dem Folgezustand für jede Eingangsspalte in der Form s^{n+1},Y angegeben.

5.2.5.4 Implementierung des Mealy-Automaten

Nächster Schritt zur Implementierung ist die Codierung der Zustände. Bei nur zwei Zuständen ist ein Codewort mit nur einer Stelle erforderlich. Die Wahl der Codierung lässt nicht viele Optionen zu und wird so gewählt, dass next-0 mit Z=0 und next-1 mit Z=1 codiert wird.

Nach Aufstellen der Ansteuerungstabelle kann der Automat mit einem Flip-Flop für den Zustandsspeicher, einem EXOR- sowie einem UND-Gatter implementiert werden (Abb. 5.53).

5.2.5.5 Vereinfachte Darstellung des Zustandsfolgediagramms

Die Darstellung des Zustandsfolgediagramms muss natürlich nicht exakt den Beispielen in Abb. 5.35 oder 5.51 entsprechen. Wenn mehrere Eingabe- oder Ausgabewerte vorhanden sind oder die Zustandsbezeichnungen zu lang werden, kann ein Diagramm auch unübersichtlich werden. Ziel sollte eine kompakte grafische Darstellung sein.

s^n	s^{n+1},Y	
	X=0	X=1
next-0 *	next-0, 0	next-1, 0
next-1	next-1, 0	next-0, 1

* = Reset

Abb. 5.52 Zustandsfolgetabelle zum Halbieren der 1-Werte

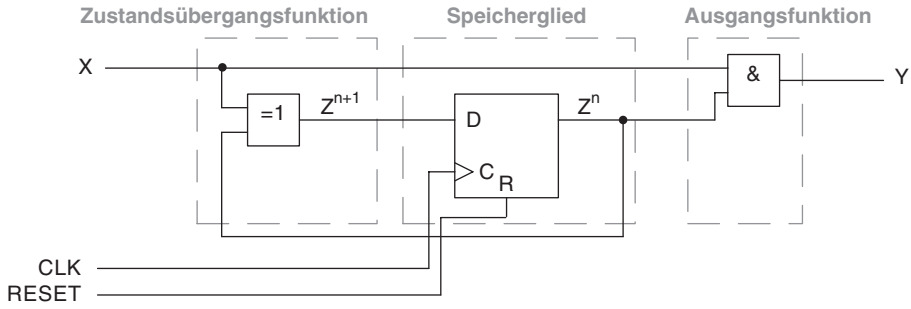


Abb. 5.53 Schaltbild des Mealy-Automaten zum Halbieren der 1-Werte

Einige Möglichkeiten zur vereinfachten Darstellung sind in Abb. 5.54 dargestellt:

1. Ein- und Ausgänge müssen nicht mit *X, Y* bezeichnet werden, sondern können natürlich Abkürzungen entsprechend der Spezifikation haben, beispielsweise im Bild *A1, A2, P, T.*
2. Eingangs- und Ausgangswerte müssen nicht stets neu benannt werden, sondern können in einer festen Reihenfolge angegeben werden. Eine empfohlene Reihenfolge ist: Eingangswerte, Schrägstrich, Ausgangswerte
3. Wenn für mehrere Eingangskombinationen derselbe Folgezustand eingenommen werden soll, kann dies an einen gemeinsamen Übergangspfeil angetragen werden
4. Zustände können einfach durchnummeriert werden (*S0, S1,* …) und die Bedeutung wird als Liste dokumentiert.

Eine andere Vereinfachung ist für die Zustandsübergänge möglich. Es kommt vor, dass für einen Zustandsübergang nur ein Teil der Eingangsvariablen beachtet werden muss. Dies kann man darstellen, indem man die erforderliche Eingabe benennt (Abb. 5.55, links) oder die nicht erforderliche Eingabe mit ‚X', für „Eingang beliebig" bezeichnet (Abb. 5.55, rechts).

Wichtig ist, dass sämtliche 2^n Eingangskombinationen bei n Eingangswerten berücksichtigt sind. Außerdem darf ein Diagramm auch nicht kryptisch kurz werden. In der Praxis muss man nach zwei Wochen, zwei Monaten oder zwei Jahren das Diagramm immer noch lesen und verstehen können.

5.2.6 Vergleich von Mealy- und Moore-Automat

Anhand der vorgestellten Beispiele können die Charakteristika von Mealy- und Moore-Automat jetzt verglichen werden. Der Mealy-Automat hat mehr Möglichkeiten, denn eine Ausgabe ist für jeden Übergangspfeil und nicht nur für die Zustandskreise definiert.

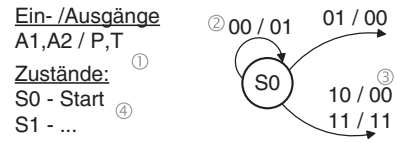


Abb. 5.54 Vereinfachte Darstellung eines Zustandsfolgediagramms

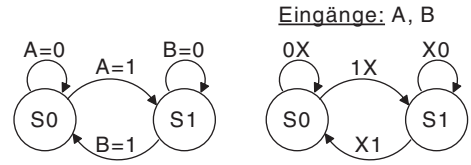


Abb. 5.55 Zwei Varianten zur Zusammenfassung von Zustandsübergängen

Dies macht ihn jedoch im Entwurf auch etwas komplexer. Der Moore-Automat hat hingegen den Vorteil, dass weniger Fälle für die Ausgabe definiert werden müssen, was ihn übersichtlicher macht.

Moore-Automat
Wegen der besseren Übersichtlichkeit wird in der Praxis meist der Moore-Automat verwendet. Die Zustände des Moore-Automaten entsprechen oft einer bestimmten Ausgabesituation, sodass die Funktion des Automaten einfacher nachvollzogen werden kann.

Die Übersichtlichkeit eines Schaltungsentwurfs erhöht seine *Wartbarkeit*. Damit ist nicht die Reparatur einer defekten Schaltung gemeint, sondern die Möglichkeit, einen Entwurf später einmal zu ändern und anzupassen. Je übersichtlicher ein Schaltungsentwurf ist, umso höher ist die Wartbarkeit.

Mealy-Automat
Ein wesentlicher Vorteil des Mealy-Automaten ist dessen Geschwindigkeit. Der Moore-Automat geht für eine Änderung der Ausgabe in einen neuen Zustand, was stets einen Taktzyklus dauert. Für viele Anwendungen stellt diese Verzögerung kein Problem dar.

Manchmal muss eine Schaltung jedoch sehr schnell reagieren, ohne auf ein Taktsignal zu warten. Dies kann zum Beispiel bei Bussystemen wie dem PCI-Bus im PC der Fall sein. Für solche Fälle kann der Mealy-Automat noch im gleichen Taktzyklus eine Antwort geben. Dies ist auch im Zeitablauf von Abb. 5.49 ersichtlich. Die 1-Impulse werden im gleichen Taktzyklus weitergegeben. Es tritt nur eine kleine Verzögerung durch das UND-Gatter der Ausgangsfunktion auf.

Verwendung beim Automatenentwurf
Es wird empfohlen, im Normalfall einen Automaten als Moore-Automaten zu entwerfen. Nur wenn der Automat noch im gleichen Taktzyklus eine Antwort ausgeben muss, empfiehlt sich der Einsatz eines Mealy-Automaten.

5.2.7 Registerausgabe

5.2.7.1 Taktkonzept

Mit der bisher gezeigten Struktur erfolgt für die Automaten die Ausgabe der Signalwerte Y aus einer kombinatorischen Verknüpfung. In der Praxis ist es vorteilhaft, wenn Teilschaltungen klare Schnittstellen zu den folgenden Teilschaltungen haben. Deshalb wird oft ein Taktkonzept verwendet, bei dem die Ausgänge von Teilschaltungen immer aus einem Flip-Flop stammen müssen. Man spricht auch von einer *Registerausgabe*.

Für den Mealy-Automaten ist eine Registerausgabe normalerweise nicht erwünscht, denn der Vorteil bei diesem Automaten ist ja gerade die Reaktion der Schaltung ohne Warten auf das nächste Taktsignal.

5.2.7.2 Moore-Automat mit Registerausgabe

Für den Moore-Automaten kann eine Registerausgabe durch eine Veränderung des Blockschaltbilds erreicht werden. Dies ist in Abb. 5.56 dargestellt ist. Die Änderung funktioniert so, dass die Ausgangsfunktion nicht mit der gespeicherten aktuellen Zustandsvariable Z^n rechnet, sondern mit der neuen Zustandsvariable Z^{n+1}. Dadurch liegt das Ergebnis Y^* der Ausgangsfunktion bereits früher vor. Damit das gleiche Zeitverhalten wie im ursprünglichen Blockschaltbild entsteht, werden die Variablen Y^* in einer Registerstufe gespeichert und ergeben den Ausgang Y. Die Ausgangsfunktion wird also vor die Flip-Flops geschoben und die Ausgabe zum Ausgleich durch Flip-Flops gespeichert.

Beide Strukturen des Moore-Automaten sind äquivalent, haben also die gleiche logische Funktion. Allerdings ist das Zeitverhalten anders. Durch das Verschieben der Ausgabefunktion gibt der Automat die Werte für Y direkt aus Flip-Flops aus, was für das Taktkonzept gewünscht ist. Die nachfolgende Schaltung hat die komplette Zeit des Taktzyklus für ihre Berechnungen.

Eine ausführliche Erläuterung von Taktkonzept und Laufzeiten befindet sich in Kapitel 6.

5.2.7.3 Beispiel für Moore-Automat mit Registerausgabe

Der im Abschn. 5.2.3 entworfene Moore-Automat zum Entprellen eines Signals wurde auf Registerausgabe umgestellt. Als Ausgangsbasis wurde das Schaltbild in Abb. 5.40 verwendet. Für die Registerausgabe wird die Ausgangsfunktion vor die Speicherglieder

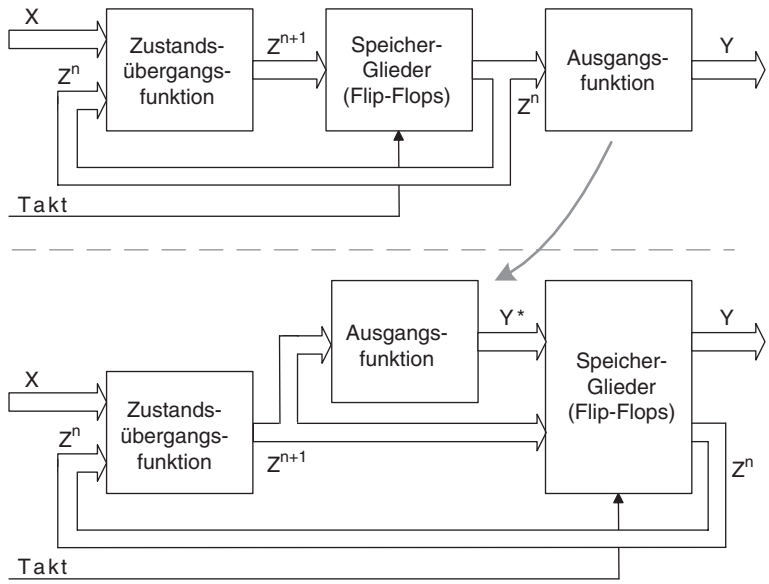


Abb. 5.56 Struktur des Moore-Automaten mit Registerausgabe

gezogen und der Wert Y^* in einem Speicherglied gespeichert. Die veränderte Schaltung ist in Abb. 5.57 dargestellt. Die Größe der Schaltung ändert sich nicht. Lediglich für den Ausgangswert Y wird ein weiteres Flip-Flop benötigt, aber genau dieses Flip-Flop ist ja erwünscht.

5.2.7.4 Medwedew-Automat

Der Medwedew-Automat ist ein Spezialfall des Moore-Automaten, bei dem die Ausgangsvariablen Y gleich den Zustandsvariablen Z^n sind. Darum sind für den Medwedew-Automat keine weiteren Ausgangs-Flip-Flops erforderlich, weil die Ausgangsvariablen ja bereits aus einem Flip-Flop kommen. Diese Struktur zeigt Abb. 5.58.

Für bestimmte Anwendungen lässt sich beim Entwurf eines Moore-Automaten einplanen, dass die Zustandsvariablen auch als Ausgangsvariablen verwendet werden. Ein Beispiel hierfür ist die optimierte Codierung des Entprell-Automaten (Abschn. 5.2.4.3), bei dem eine Zustandsvariable gleich dem Ausgang gewählt wurde. Auch ein Zähler ist ein Medwedew-Automat. Er gibt nacheinander Zahlenwerte aus, wie 0, 1, 2, 3, … Diese Zahl wird als Zustand gespeichert und ist die Ausgabe.

In der Praxis wird in vielen Fällen der Aufwand für zusätzliche Ausgangs-Flip-Flops akzeptiert. Der Arbeitsaufwand für eine spezielle Codierung wird hingegen vermieden.

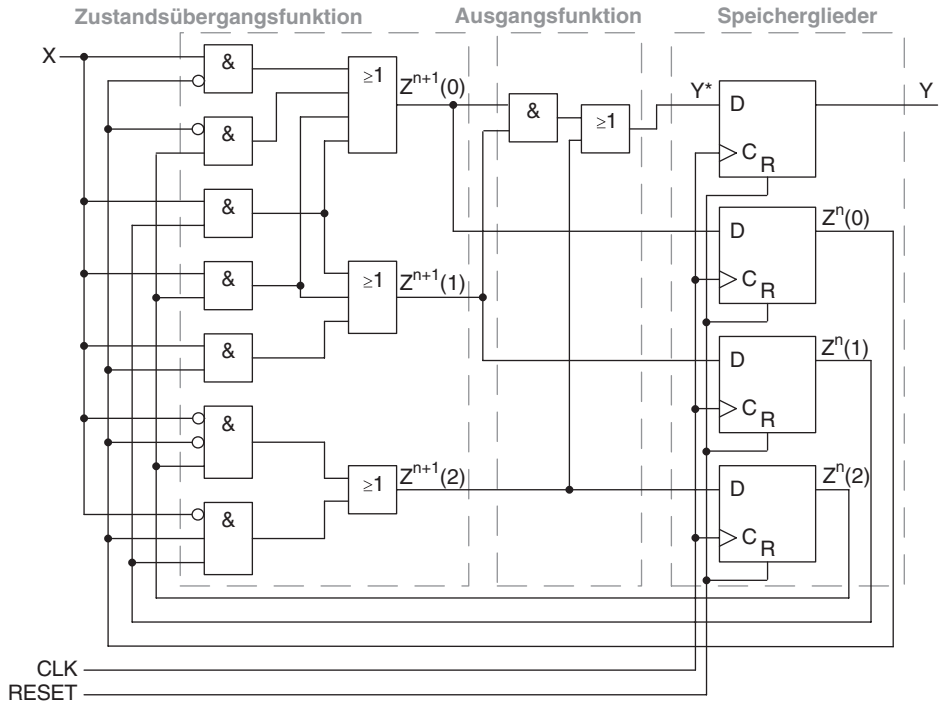


Abb. 5.57 Entprell-Automat mit Registerausgabe

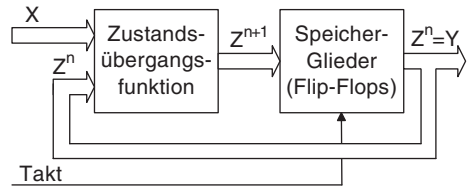


Abb. 5.58 Struktur des Medwedew-Automaten

5.2.8 Asynchrone Automaten

Eine weitere Form von Automaten sind *asynchrone Automaten*. Sie werden in der Praxis sehr selten entworfen und daher wird hier nur kurz ihre prinzipielle Struktur erläutert.

5.2.8.1 Struktur

Bei asynchronen Automaten sind keine Flip-Flops zur Datenspeicherung vorhanden. Die Zustandsinformation wird stattdessen direkt vom Ausgang der Zustandsübergangsfunktion zurück nach dessen Eingang gegeben. Die Speicherung der Information findet in der Verzögerung der Logikgatter und der Verbindungsleitungen statt.

Abb. 5.59 zeigt diese Struktur. Die kombinatorische Schaltung besteht aus den Logikgattern für Zustandsübergangsfunktion und Ausgabefunktion. Der als Verzögerung angegebene Block ist kein reales Bauelement, sondern symbolisiert das Zeitverhalten der Logikgatter.

Asynchrone Automaten haben in der Theorie einige Vorteile gegenüber synchronen Automaten, also Automaten mit Flip-Flops:

- Höhere Geschwindigkeit, denn der Takt muss nicht auf die langsamste Verzögerung der kombinatorischen Schaltung warten.
- Niedrigerer und gleichmäßigerer Stromverbrauch, denn bei synchronen Schaltungen sind bei den Taktflanken Hunderttausende von FFs gleichzeitig aktiv.
- Geringere Störausstrahlung, denn es gibt keinen Takt.

In der Praxis gibt es jedoch auch schwerwiegende Nachteile, die gleich in Abschn. 5.2.8.3 folgen.

Abb. 5.59 Struktur eines asynchronen Automaten

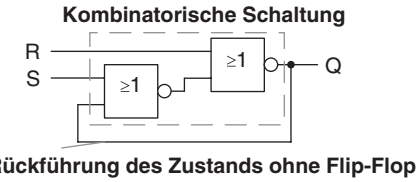


Abb. 5.60 Strukturelemente des asynchronen Automaten beim RS-Flip-Flop

5.2.8.2 Beispiel eines asynchronen Automaten

Das in Abschn. 5.1.1 beschriebene RS-Flip-Flop ist ein Beispiel für einen asynchronen Automaten. Abb. 5.60 zeigt erneut den Aufbau des RS-FFs (wie in Abb. 5.3) mit den Strukturelementen des asynchronen Automaten. Die Rückführung des Zustands Q erfolgt ohne Verzögerung oder Flip-Flop.

5.2.8.3 Einsatz

Der praktische Einsatz von asynchronen Automaten ist nicht einfach, denn beim Entwurf sind wesentlich mehr Bedingungen zu beachten als bei synchronen Automaten.

- Ein asynchroner Automat ist nur stabil, wenn die Änderung einer Zustandsvariablen nicht erneut zu immer weiteren Änderungen von Zustandsvariablen führt. Ansonsten kann der Automat zwischen verschiedenen Zuständen schwingen.
- Die kombinatorische Schaltung darf keine kurzzeitigen Zwischenwerte ausgeben. Ansonsten kann der Automat in einen falschen Zustand übergehen.

Aufgrund dieser Bedingungen sind asynchrone Automaten wesentlich schwieriger zu entwerfen, denn Fehler beim Einhalten der Bedingungen lassen sich nur schwer entdecken. Das Risiko beim Entwurf eines asynchronen Automaten ist relativ hoch.

In der Praxis werden darum asynchrone Automaten so gut wie nicht entworfen. In der Regel werden lediglich bewährte und besonders geprüfte Grundschaltungen eingesetzt, wie zum Beispiel das RS-Flip-Flop.

5.3 Entwurf sequenzieller Schaltungen mit VHDL

5.3.1 Grundform des getakteten Prozesses

Der Entwurf sequenzieller Schaltungen erfolgt in VHDL mit einer besonderen Form des bereits beschriebenen Prozesses. Der Prozess benötigt keine Sensitivity-Liste und beginnt mit einem Wait-Statement für die steigende Taktflanke. Dieses Wait-Statement hat die Schreibweise *wait until rising_edge(clk);* und sagt aus, dass die nachfolgenden Anweisungen nur bei einer steigenden Taktflanke ausgeführt werden sollen. Nach dem Wait-Statement steht der VHDL-Code, der bei der steigenden Taktflanke ausgeführt werden soll.

```
signal a, b : std_logic;
…
process
begin
   wait until rising_edge(clk);
   b <= a;
end process;
```

Im VHDL-Code sind nur die Definition von *a* und *b* sowie der Prozess gezeigt. Entity und Architecture-Definition werden zur besseren Übersicht zunächst weggelassen. Ein vollständiges Beispiel folgt später. Das Taktsignal *clk* ist ein normales Signal in VHDL; oft ist es direkt ein Eingangssignal der Schaltung.

Das Beispiel beschreibt ein einfaches D-Flip-Flop. Mit der steigenden Taktflanke wird der Wert des Signals *a* im Signal *b* gespeichert. Diese Beschreibung entspricht einem D-Flip-Flop entsprechend Abb. 5.61.

Die Schreibweise *rising_edge()* ist für die Beschreibung sequenzieller Schaltungen sehr wichtig. Syntheseprogramme erkennen diese Funktion und generieren eine Schaltung mit D-Flip-Flops. Es gibt außerdem die Variante *falling_edge()*. Hiermit wird eine Funktion beschrieben, die bei einer fallenden Taktflanke aktiv ist. Entsprechend werden D-Flip-Flops generiert, die mit der fallenden Taktflanke aktiv sind.

5.3.2 Erweiterte Funktion des getakteten Prozesses

Die Grundform des Prozesses erscheint zunächst relativ aufwendig, denn für ein einzelnes Flip-Flop werden vier Zeilen VHDL-Code benötigt. Die Stärke von VHDL liegt darin, dass nach dem Wait-Statement weitere Funktionen beschrieben werden können. Es sind If-Abfragen, Case-Bedingungen und logische Verknüpfungen, auch ineinander geschachtelt, möglich. Die Optimierung der Schaltung wird von einem Synthese-Programm übernommen.

Als immer noch kleines Beispiel wird eine Überlauferkennung betrachtet. *count* ist eine Zahl mit dem Wertebereich von 0 bis 15 und in VHDL als unsigned-Signal mit 4 bit Wortbreite definiert. Eine Schaltung soll überprüfen, ob der Zahlenwert größer als zehn ist und das Ergebnis in einem Flip-Flop speichern. Dies könnte zum Beispiel anzeigen, dass ein Speicher überläuft.

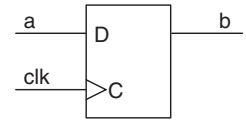


Abb. 5.61 Schaltung des in VHDL beschriebenen D-Flip-Flops

```
signal count : unsigned(3 downto 0);
signal overflow : std_logic;
…
process
begin
   wait until rising_edge(clk);
   if count > 10 then
        overflow <= '1';
   else
        overflow <= '0';
   end if;
end process;
```

Nach dem Wait-Statement wird eine If-Abfrage mit der Konstanten zehn geschrieben. Ein Syntheseprogramm würde hieraus die Schaltung in Abb. 5.62 synthetisieren. Der Vorteil von VHDL ist, dass man sich über die Logikfunktion keine Gedanken machen muss. Auch Änderungen sind einfach. Wenn der Überlauf nicht bei Werten größer zehn, sondern bei elf oder zwölf erfolgen soll, wird einfach die Zahl im VHDL-Code geändert und das Synthese-Programm berechnet die neue Schaltung.

5.3.3 Steuerleitungen für Flip-Flops

Durch VHDL-Beschreibungen können auch die am Anfang dieses Kapitels in Abschn. 5.1.4 beschriebenen Erweiterungen des D-Flip-Flops realisiert werden, also Reset, Set und Enable. Die Reset- und Set-Eingänge können entweder als synchrone oder als asynchrone Eingänge implementiert werden. Für die VHDL-Beschreibung wird ein synchrones Rücksetzen der Schaltung empfohlen. Zum einen wird dies in der Praxis meist verwendet, zum andern ist die VHDL-Beschreibung etwas einfacher.

5.3.3.1 Synchroner Reset und Set

Der synchrone Reset und Set wird durch eine If-Abfrage des Steuersignals beschrieben. Diese If-Abfrage folgt direkt nach der Wait-Anweisung und beschreibt erst das Verhalten bei der Initialisierung und dann in der Else-Verzweigung die reguläre Verarbeitung.

Der folgende VHDL-Code erzeugt zwei D-Flip-Flops, *f* mit synchronem Reset und *g* mit synchronem Set. Beim Steuersignal wird üblicherweise der Name *reset* verwendet,

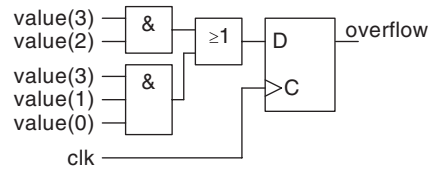


Abb. 5.62 Schaltung der in VHDL beschriebenen Überlauferkennung

egal auf welche Polarität initialisiert wird. Für die Else-Verzweigung werden als Beispiel einfache kombinatorische Verknüpfungen aufgerufen.

```
process
begin
   wait until rising_edge(clk);
   if reset = '1' then
      f <= '0';
      g <= '1';
   else
      f <= a or b;
      g <= b and c and d;
   end if;
end process;
```

Im Else-Zweig können, wie im Beispiel gezeigt, Berechnungen und Verknüpfungen programmiert werden. Für den Reset-Fall sind jedoch nur feste Werte, also 0 oder 1 möglich. Der Grund hierfür ist, dass die VHDL-Beschreibung in eine digitale Schaltung umgewandelt werden soll. Dabei wird der Reset-Wert für die Auswahl des Flip-Flops verwendet. Deswegen muss ein fester Wert vorhanden sein, anhand dessen entweder ein Flip-Flop mit Reset oder Set verwendet wird.

- Steht im Reset-Zweig die Anweisung *f <= '0';* wird ein Flip-Flop mit Reset erzeugt.
- Steht im Reset-Zweig die Anweisung *f <= '1';* wird ein Flip-Flop mit Set erzeugt.
- Steht im Reset-Zweig die Anweisung *f <= a;* oder *f <= b* ***or*** *c;* kann nicht entschieden werden, ob ein Flip-Flop mit Set oder Reset erzeugt wird. Stattdessen wird ein Flip-Flop ohne Rücksetzfunktion erzeugt und die Funktion wird durch Logikgatter umgesetzt. Dies ist normalerweise nicht erwünscht, wenn der VHDL-Code eine Initialisierung beschreibt.

5.3.3.2 Asynchroner Reset und Set

Sequenzielle Schaltungen mit asynchronem Reset und Set werden durch einen VHDL-Programmierstil ohne Wait-Statement beschrieben. Stattdessen wird der Prozess mit einer Sensitivity-Liste für Takt und Steuersignal aufgerufen. Die Beschreibung der sequenziellen Schaltung erfolgt durch eine If-Elsif-Abfrage. Das Reset-Verhalten wird im If-Zweig, der Takt im Elsif-Zweig beschrieben. Die Syntax für Reset und Set ist gleich; die Unterscheidung erfolgt durch Zuweisung einer 0 oder 1.

Die Reihenfolge von If- und Elsif-Zweig entspricht der Priorität, denn das asynchrone Rücksetzen erfolgt ja unabhängig vom Takt. Die Takt-Abfrage folgt mit *elsif,* denn sie wird nur ausgeführt, wenn kein Reset anliegt.

```
process(clk, reset)
begin
   if reset = '1' then
      f <= '0';
      g <= '1';
   elsif rising_edge(clk) then
      f <= a or b;
      g <= b and c and d;
   end if;
end process;
```

Die Syntax von *if reset = '1' then* und *elsif rising_edge(clk) then* muss unbedingt eingehalten werden. Nur so kann das Syntheseprogramm erkennen, dass es ein D-Flip-Flop mit Reset oder Set einbauen soll. Zwischen *end if;* und *end process;* darf kein anderer VHDL-Code eingeschoben werden. Natürlich kann auch ein Flip-Flop mit fallender Taktflanke erzeugt werden, indem die Abfrage auf *falling_edge(clk)* erfolgt.

Wie zuvor sind für den Reset-Fall nur feste Werte, also 0 oder 1 möglich. Bei der Anweisung *f <= '0';* wird ein Flip-Flop mit Reset erzeugt, bei *f <= '1';* ein Flip-Flop mit Set. Falls keine Konstante für den Reset-Fall angegeben ist, würde das Syntheseprogramm einen Fehler ausgeben. Der Grund hierfür ist, dass es für diese Beschreibung kein passendes Schaltungselement gibt. Angenommen im Reset-Fall stände die Anweisung *f <= a;*. Bei *a*=0 soll ein asynchroner Reset erfolgen, bei *a*=1 ein asynchroner Set. Das Syntheseprogramm muss aber entweder ein Flip-Flop mit Set oder mit Reset einbauen. Da dies nicht möglich ist, erfolgt die Fehlermeldung.

5.3.3.3 Enable

Auch eine Enable-Funktionalität wird durch eine If-Abfrage beschrieben. Die Enable-Abfrage enthält keine Else-Beschreibung. Wenn *enable* aktiviert ist, wird der neue Wert übernommen, ansonsten findet keine Änderung statt. Der folgende VHDL-Code erzeugt zwei D-Flip-Flops mit Enable.

```
process
begin
   wait until rising_edge(clk);
   if enable = '1' then
      f <= a or b;
      g <= b and c and d;
   end if;
end process;
```

Enable und Reset/Set können auch miteinander kombiniert werden. Dabei wird zuerst die If-Anweisung für die Rücksetzfunktion geschrieben, denn die Initialisierung hat normalerweise eine höhere Priorität als der Enable-Eingang.

5.3.4 Entwurf von Automaten

Mit einem Prozess kann auch ein kompletter Automat beschrieben werden. Zuvor müssen das Zustandsfolgediagramm und die Zustandsfolgetabelle erstellt werden (vgl. Abschn. 5.2.3). Die weiteren Schritte, also Codierung der Zustände und Generierung der Logik wird dann durch VHDL-Beschreibung und Logiksynthese übernommen.

5.3.4.1 Elemente der VHDL-Beschreibung

Im VHDL-Code wird die Funktion des Automaten nach der Wait-Anweisung beschrieben (vgl. Abschn. 5.3.2). Außerdem erfolgt ein Reset des Automaten.

Eine Besonderheit ist die Beschreibung des Zustands. Es ist empfehlenswert, für die Speicherung des Zustands einen neuen, individuellen Datentyp zu definieren. Dies hat zwei Vorteile:

- Der VHDL-Code wird lesbarer.
- Das Syntheseprogramm weiß durch diese Beschreibung, dass ein Automat synthetisiert werden soll und kann die Schaltung optimieren.

Die Definition des Datentyps erfolgt in der Architecture mit dem Befehl:

```
type   <type_name> is (value_0, value_1, …);
```

Dieser Befehl definiert nur, dass es einen neuen Datentyp gibt. Zusätzlich muss noch ein Signal mit diesem Datentyp erzeugt werden. Dies erfolgt mit dem Befehl:

```
signal   <signal_name> : <type_name>;
```

In diesem Abschnitt soll der Entprell-Automat aus Abschn. 5.2.3 als Beispiel verwendet werden. Der Automat hat sechs Zustände, die erst als Datentyp definiert und dann als Signal verwendet werden. Die Zustandsnamen des Beispiels müssen leicht angepasst werden, da VHDL-Signale nicht mit Ziffern beginnen und keine Leerzeichen enthalten dürfen. Das Zustandsfolgediagramm Abb. 5.35 ist mit den angepassten Zustandsnamen in Abb. 5.63 noch einmal dargestellt.

Damit lautet die Signaldefinition in VHDL:

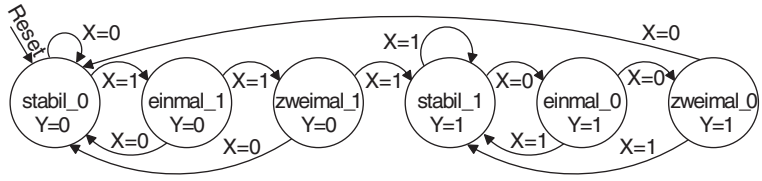


Abb. 5.63 Zustandsfolgediagramm des Entprell-Automaten für die VHDL-Beschreibung

```
type   state_type is (stabil_0, einmal_1, zweimal_1,
                      stabil_1, einmal_0, zweimal_0);
signal state        :  state_type;
```

Die Funktion des Automaten wird dann in einem synchronen Prozess umgesetzt. Zunächst wird mit einem Reset der Startzustand programmiert und auch die Ausgabe *Y* des Automaten auf einen Startwert gesetzt. Laut Zustandsfolgediagramm Abb. 5.63 ist *stabil_0* mit *Y*=0 der Startzustand. Dies schreibt sich in VHDL:

```
process
begin
   wait until rising_edge(clk);
   if reset = '1' then
      state <= stabil_0;
      y     <= '0';
   else
      …
```

Als nächstes folgt die Beschreibung der einzelnen Zustände. Hierfür wird ein Case-Statement mit dem Zustandssignal als Operator verwendet. Die Zustände sind die When-Bedingungen. Innerhalb der When-Bedingungen wird dann Folgezustand und Ausgabe für den Folgezustand beschrieben. Die Abhängigkeit von den Eingangswerten wird durch ein If-Statement oder ein Case-Statement beschrieben.

Der folgende VHDL-Code gilt wieder für den Entprell-Automaten. Er beschreibt das Case-Statement abhängig von Zustandssignal *state* und den ersten Fall für den Zustand *stabil_0*. Der Folgezustand ist abhängig vom Eingang *x,* und wird als If-Statement beschrieben. Die Beschreibung für die weiteren Zustände erfolgt in der gleichen Weise und wird hier zunächst übersprungen. Der komplette VHDL-Code steht im folgenden Unterabschnitt.

Beachten Sie, dass mit den Folgezuständen jeweils die neuen Ausgabewerte beschrieben werden. Im folgenden VHDL-Code gibt es den Folgezustand *stabil_0,* der die Ausgabe *Y* gleich 0 hat, sowie den Folgezustand *einmal_1,* der ebenfalls die Ausgabe *Y* gleich 0 hat. Durch diese Schreibweise wird ein Moore-Automat mit Registerausgabe erzeugt, wie in Abschn. 5.2.7 beschrieben.

```
case state is
   when stabil_0 =>
      if x='0' then
         state <= stabil_0;
         y <= '0';
      else
         state <= einmal_1;
         y <= '0';
```

```
      end if;
   when einmal_1 =>
      …
end case;
```

Die Beschreibung der einzelnen Zustände kann direkt aus dem Zustandsfolgediagramm (Abb. 5.63) und der Zustandsfolgetabelle übertragen werden.

Damit ist die komplette Funktion des Automaten beschrieben. Der komplette VHDL-Code mit Aufruf der IEEE-Bibliothek, Entity und Architecture ist im folgenden Unterkapitel angegeben. Die If-Statements für die Folgezustände werden in drei Zeilen formatiert, um den Automaten kompakter und damit übersichtlicher zu beschreiben. Die Formatierung hat keine Auswirkung auf die Bedeutung des VHDL-Codes und sollte gut lesbar gestaltet werden.

5.3.4.2 Kompletter VHDL-Code des Automaten

```
library ieee;
use ieee.std_logic_1164.all;

entity entprell is
   port (clk   :in  std_logic;
         reset :in  std_logic;
         x     :in  std_logic;
         y     :out std_logic);
end;

architecture behave of entprell is

   type   state_type is (stabil_0, einmal_1, zweimal_1,
                         stabil_1, einmal_0, zweimal_0);
   signal state       :  state_type;

begin

process
begin
wait until rising_edge(clk);
   if reset = '1' then
      state <= stabil_0;
      y     <= '0';
   else
      case state is
         when stabil_0 =>
            if x='0' then   state <= stabil_0;     y <= '0';
```

```
            else            state <= einmal_1;    y <= '0';
            end if;
         when einmal_1 =>
            if x='0' then    state <= stabil_0;    y <= '0';
            else            state <= zweimal_1;   y <= '0';
            end if;
         when zweimal_1 =>
            if x='0' then    state <= stabil_0;    y <= '0';
            else            state <= stabil_1;    y <= '1';
            end if;
         when stabil_1 =>
            if x='1' then    state <= stabil_1;    y <= '1';
            else            state <= einmal_0;    y <= '1';
            end if;
         when einmal_0 =>
            if x='1' then    state <= stabil_1;    y <= '1';
            else            state <= zweimal_0;   y <= '1';
            end if;
         when zweimal_0 =>
            if x='1' then    state <= stabil_1;    y <= '1';
            else            state <= stabil_0;    y <= '0';
            end if;
      end case; -- state
   end if; -- reset
end process;
end;
```

5.3.5 Programmierstile für VHDL-Code

Wie in jeder Programmiersprache sind auch für VHDL verschiedene Programmierstile möglich. Wir haben einen Programmierstil gewählt, der gut lesbar und wenig fehleranfällig ist. In der Praxis werden Sie sicherlich auch VHDL-Code in anderer Schreibweise begegnen. Für den Einstieg in VHDL empfehlen wir, zunächst bei einer Schreibweise zu bleiben.

Der VHDL-Code wird durch ein Syntheseprogramm in eine Schaltung umgewandelt. Heutige Syntheseprogramme sind so intelligent, dass sie für die meisten Programmierstile eine kompakte und schnelle Schaltung erzeugen.

In der Praxis gibt es innerhalb größerer Entwicklerteams oft eigene Richtlinien für Programmierstile, damit von verschiedenen Personen geschriebener Code nicht zu inhomogen wird.

5.4 Übungsaufgaben

Haben Sie den Inhalt des Kapitels verstanden? Prüfen Sie sich mit den Aufgaben und Fragen am Kapitelende. Die Lösungen und Antworten finden Sie am Ende des Buches. Bei den Auswahlfragen ist immer genau eine Antwort korrekt.

Bitte versuchen Sie unbedingt, die Aufgaben zu den Automaten zuerst selber zu lösen. Nur durch Übung lernen Sie den Entwurf von Automaten. Die Lösungen sind bewusst sehr knapp gehalten und werden am besten verstanden, wenn Sie vorher selbst eine Lösung ermittelt haben.

Aufgabe 5.1
Was gilt für ein RS-Flip-Flop (RS-FF)?

a) Daten werden unabhängig von einem Takt gespeichert
b) Daten werden bei Takt gleich 1 gespeichert
c) Daten werden bei Takt gleich 0 gespeichert
d) Daten werden bei einer Taktflanke gespeichert

Aufgabe 5.2
Welche Ansteuerung für Flip-Flops ist heutzutage üblich?

a) Taktflanke
b) Unabhängig vom Takt
c) Taktpegel

Aufgabe 5.3
Wie reagiert ein D-Flip-Flop (D-FF) auf einen asynchronen Reset?

a) Ausgang geht bei der nächsten Taktflanke auf 0
b) Ausgang geht sofort auf 1
c) Ausgang geht bei der nächsten Taktflanke auf 1
d) Ausgang geht sofort auf 0

Aufgabe 5.4
Wie reagiert ein D-Flip-Flop (D-FF) auf einen synchronen Set?

a) Ausgang geht bei der nächsten Taktflanke auf 0
b) Ausgang geht sofort auf 1
c) Ausgang geht bei der nächsten Taktflanke auf 1
d) Ausgang geht sofort auf 0

Aufgabe 5.5
Ein Automat, bei dem der Ausgang nur vom Zustand und NICHT von den momentanen Eingangswerten abhängt, bezeichnet man als, …

a) Endlicher Automat
b) Mealy-Automat
c) Moore-Automat
d) Turing-Automat
e) Medwedew-Automat

Aufgabe 5.6
Ein Automat, bei dem der Ausgang vom Zustand UND von den momentanen Eingangswerten abhängt, bezeichnet man als, …

a) Endlicher Automat
b) Moore-Automat
c) Medwedew-Automat
d) Turing-Automat
e) Mealy-Automat

Aufgabe 5.7
Betrachten Sie die Taktsignale in Abb. 5.64. Wie groß ist für die Diagramme a) bis c) jeweils:

- Periodendauer
- Taktfrequenz
- Duty Cycle der 1-Phase

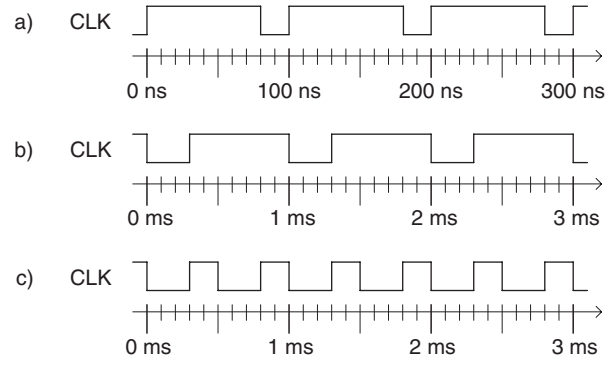


Abb. 5.64 Taktsignale

Aufgabe 5.8
Ein Automat mit 11 Zuständen soll mit minimaler Codewortlänge codiert werden. Wie viele Stellen muss das Codewort haben?

Aufgabe 5.9
Ein Automat mit 9 Zuständen soll mit einer One-Hot-Codierung codiert werden. Wie viele Stellen muss das Codewort haben?

Aufgabe 5.10
Wie viele Zustände können bei minimaler Codewortlänge mit 5 Stellen codiert werden?

Aufgabe 5.11
Wie viele Zustände können mit einer One-Hot-Codierung mit 8 Stellen codiert werden?

Aufgabe 5.12
Die Jalousie an einem Fenster soll durch einen einzelnen Taster angesteuert werden. Um nur einen Taster zu verwenden, ändert sich die Bewegungsrichtung der Jalousie bei jeder neuen Betätigung des Tasters. Solange wie der Taster gedrückt gehalten wird, bewegt sich die Jalousie nach oben oder nach unten. Beim Loslassen stoppt die Jalousie, kann also auch halb oder zweidrittel geschlossen werden.

Beispiel:

- Der Taster wird gedrückt und festgehalten: Die Jalousie bewegt sich nach unten.
- Der Taster wird losgelassen: Die Jalousie stoppt.
- Der Taster wird gedrückt und festgehalten: Die Jalousie bewegt sich nach oben.
- Der Taster wird losgelassen: Die Jalousie stoppt.
- Der Taster wird gedrückt und festgehalten: Die Jalousie bewegt sich nach unten.

Nach dem Start soll sich die Jalousie bei Tastendruck zuerst nach unten bewegen. Das Ende der Bewegung, also ganz offen oder ganz geschlossen, wird nicht überprüft, da der Motor dann selbstständig stoppt.

Die Jalousie soll durch einen Moore-Automaten angesteuert werden. Der Taster liegt am Eingang T und ist 1, wenn er gedrückt wird. Der Motor wird durch zwei Ausgänge *M(1:0)* angesteuert. Die Bedeutung ist:

- M=00 – Motor ausgeschaltet
- M=01 – Motor fährt herunter
- M=10 – Motor fährt herauf
- M=11 – nicht zulässig

a) Stellen Sie das Zustandsfolgediagramm auf. Verwenden Sie so wenige Zustände wie möglich.
b) Erstellen Sie die Zustandsfolgetabelle.

Aufgabe 5.13
Mit einem Automaten sollen Parkmünzen zum Preis von 50 Cent verkauft werden. Ein elektromechanisches System erkennt Münzen im Wert von 10, 20 und 50 Cent und meldet eine eingeworfene Münze auf zwei Leitungen *M(1:0)*. Wird keine Münze eingeworfen, ist M=00. Erkannte Münzen werden mit einem Signal der Länge eines Taktes angezeigt. Die Münzen werden wie folgt codiert:

- M=01 – 10 Cent
- M=10 – 20 Cent
- M=11 – 50 Cent

Werden insgesamt mehr als 50 Cent eingeworfen, wird das übrige Geld einbehalten. Beispiele für erlaubte Kombinationen sind also:

- 20 Cent, 20 Cent, 10 Cent
- 50 Cent
- 20 Cent, 20 Cent, 20 Cent (10 Cent verfallen)
- 20 Cent, 50 Cent (ungeschickte Reihenfolge, 20 Cent verfallen)

Entwerfen Sie einen Moore-Automaten, der die Leitungen *M(1:0)* auswertet und nach Einwurf eines Betrags von mindestens 50 Cent einen Ausgang P für einen Takt auf 1 schaltet, um eine Parkmünze auszugeben. Danach kann erneut Geld für die nächste Parkmünze eingeworfen werden. Aufgrund der mechanischen Auswertung vergehen zwischen zwei Münzeinwürfen mehrere Takte.

a) Stellen Sie das Zustandsfolgediagramm auf. Verwenden Sie so wenige Zustände wie möglich.
b) Erstellen Sie die Zustandsfolgetabelle.

Aufgabe 5.14
Der Automat zum Halbieren der Takte mit dem Wert 1 aus Abschn. 5.2.5 soll als Moore-Automat entworfen werden.

Wenn am Eingang X der Wert 1 anliegt, soll für jeden zweiten Wert eine 1, ansonsten eine 0 am Ausgang Y ausgegeben werden. Die Zählung soll durch Eingangswerte 0 nicht beeinflusst werden. Bei einer 0 am Eingang soll 0 ausgegeben werden. Beim Einschalten soll für die erste 1 der Wert 0 ausgegeben werden. Der Zeitablauf entspricht Abb. 5.49, allerdings ist die Ausgabe bis zur nächsten Taktflanke verzögert (da Moore-Automat).

a) Stellen Sie das Zustandsfolgediagramm auf. Verwenden Sie so wenige Zustände wie möglich.
b) Erstellen Sie die Zustandsfolgetabelle.

Aufgabe 5.15
Auf einer Datenleitung *D* werden Datenworte der Länge 4 übertragen. Die Datenworte bestehen aus drei Stellen Nutzinformation und einer vierten Stelle zur Fehlererkennung, der Parity-Stelle. Diese vierte Parity-Stelle ist so gewählt, dass die Anzahl der 1-Stellen im Datenwort immer ungerade ist. Ein Fehler bei der Übertragung kann erkannt werden, wenn beim Empfänger die Anzahl der 1-Stellen, also die Parität, gerade ist.

Entwerfen Sie einen **Mealy-Automaten,** der die Datenleitung *D* überwacht und ein falsches Datenwort erkennt. Wenn ein falsches Datenwort mit gerader Anzahl der 1-Stellen auftritt, soll der Ausgang *E* (Error) für einen Takt auf 1 sein. Innerhalb eines Datenwortes und wenn kein Fehler auftritt, ist *E* auf 0.

Abb. 5.65 ist ein Beispiel für einen Zeitablauf. Die Klammern kennzeichnen die Datenworte.

- Das erste Datenwort hat zwei 1-Stellen, also fehlerhaft, da Parität gerade.
- Das zweite Datenwort hat drei 1-Stellen, also korrekt, da Parität ungerade.
- Das dritte Datenwort hat vier 1-Stellen, also fehlerhaft, da Parität gerade.

a) Stellen Sie das Zustandsfolgediagramm auf. Verwenden Sie so wenige Zustände wie möglich.
b) Erstellen Sie die Zustandsfolgetabelle.

Hinweise:

- Der Automat muss mitzählen, wie viele Stellen und welche Werte empfangen wurden.
- Es müssen nicht alle verschiedenen Kombinationen unterschieden werden. Es sind weniger als zehn Zustände nötig.
- Achten Sie darauf, dass nach der vierten Stelle sofort das neue Datenwort beginnt. Der Automat darf keine Pause einlegen.

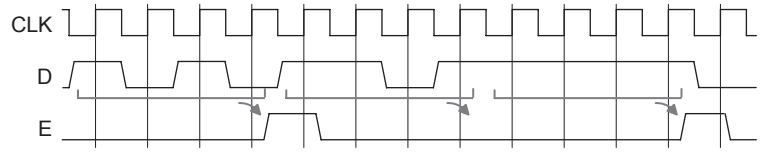


Abb. 5.65 Zeitdiagramm für die Fehlererkennung mit Parity-Stelle

Schaltungsstrukturen

6

In den Kapiteln 4 und 5 wurde gezeigt, wie aus einer Aufgabenstellung eine kombinatorische oder sequenzielle Schaltung entwickelt werden kann. Dieser allgemeine Entwurfsweg ist prinzipiell für jede Spezifikation möglich. Für bestimmte Aufgabenstellungen kann es aber auch einfacher gehen. Es gibt einige Grundstrukturen, die häufig in digitalen Schaltungen vorkommen und solche Strukturen werden in diesem Kapitel vorgestellt. Die Strukturen können durch eine Beschreibung in VHDL erzeugt werden.

6.1 Grundstrukturen digitaler Schaltungen

Wenn Sie die hier gezeigten Grundstrukturen kennen, können Sie oft eine digitale Schaltung direkt aus diesen Strukturen zusammenstellen. Sie sparen sich damit möglicherweise den allgemeinen Entwurfsweg über Funktionstabelle oder Zustandsfolgediagramm. In Abschn. 6.5.2 wird hierzu ein ausführliches Beispiel gezeigt.

6.1.1 Top-down Entwurf

Größere Digitalschaltungen werden *Top-down* entworfen, also „von oben nach unten“. Damit ist gemeint, dass eine Schaltung schrittweise in immer kleinere Teile aufgeteilt wird. Das Gesamtsystem besteht also aus Teilschaltungen, die auch als *Untermodul* bezeichnet werden. Die Untermodule können wiederum aus weiteren Untermodulen zusammengesetzt sein. Auf dem untersten Schritt der Aufteilung befinden sich Grundelemente. Dies können Schaltungsstrukturen dieses Kapitels sein, aber auch Automaten (Kapitel 4) oder einzelne Gatter und Flip-Flops.

W. Gehrke et al., *Digitaltechnik,* Springer-Lehrbuch,
DOI 10.1007/978-3-662-49731-9_6

Beispielsweise lässt sich ein Mikrocontroller (siehe Kap. 13 und 14) in die folgenden Teilschaltungen aufteilen:

- Der Mikrocontroller besteht aus den Untermodulen CPU, Speicher, Eingabe, Ausgabe, Bussystem, …
- Die CPU besteht aus den Untermodulen Rechenwerk, Steuerwerk, Register, Speicherinterface, …
- Das Steuerwerk besteht aus Programmzähler, Befehlsdecoder, …
- Der Programmzähler wird durch die Grundstruktur Zähler implementiert.

Der Vorteil dieser Vorgehensweise ist, dass die Untermodule einzeln entworfen werden können. Dies ist übersichtlich und erlaubt auch eine Aufteilung auf mehrere Personen oder, bei größeren Projekten, sogar auf mehrere Standorte eines Unternehmens.

Die einzelnen Teilschaltungen werden dann Schritt für Schritt zur Gesamtschaltung zusammengesetzt. Dieses Zusammenfügen wird als *Bottom-up* bezeichnet.

6.1.2 Darstellung von Schaltungsstrukturen

In den bisherigen Kapiteln wurden bereits grafische Symbole *(Schaltzeichen)* für die Darstellung von Schaltungselementen verwendet. Gemeint sind die rechteckigen Kästen, bei denen sich die Eingänge auf der linken Seite und die Ausgänge auf der rechten Seite befinden. An der oberen und unteren Kante können sich Steuersignale befinden.

Auch für Schaltungsstrukturen werden solche Schaltzeichen verwendet. Es gibt eine standardisierte Darstellung, die in Abb. 6.1 zu sehen ist. Die Eingänge sind links, die Ausgänge rechts angeordnet. Der obere Block erhält Steuersignale, welche die Datensignale beeinflussen. Der untere Block umfasst die Datensignale. Dabei trennt ein horizontaler Strich unabhängige Datensignale voneinander. Abkürzungen und Symbole bei „*“ und „**“ geben die Funktion an.

Die Verwendung dieser Darstellung ist in der Praxis allerdings sehr uneinheitlich. Englischsprachige Quellen verwenden die Symbole kaum und darum findet sich auch in Deutschland oft eine einfachere Darstellung. Meist wird ein einfaches Rechteck verwendet und die Funktion durch eine Beschriftung verdeutlicht.

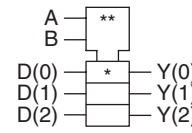


Abb. 6.1 Standardisierte Darstellung eines Schaltungselements

6.2 Kombinatorische Grundstrukturen

6.2.1 Multiplexer

Eine wichtige Grundstruktur für kombinatorische Schaltungen ist der *Multiplexer,* kurz „Mux“. Abhängig von Steuersignalen wird einer von mehreren Eingängen ausgewählt und auf den Ausgang gegeben. Je nach Anzahl der Auswahlmöglichkeiten sind ein oder mehrere Steuerleitungen erforderlich.

- 1-aus-2-Multiplexer: Für zwei Dateneingänge ist eine Steuerleitung erforderlich
- 1-aus-4-Multiplexer: Für vier Dateneingänge sind zwei Steuerleitungen nötig, denn die zwei Steuerleitungen können vier Möglichkeiten anzeigen
- 1-aus-8-Multiplexer: Für acht Dateneingänge sind drei Steuerleitungen nötig

Auch Multiplexer für mehr Eingänge sind möglich. Mit n Steuerleitungen kann aus 2^n Eingängen ausgewählt werden.

Das Schaltsymbol für einen 1-aus-4-Multiplexer ist in Abb. 6.2 dargestellt. Links befindet sich das Symbol in der standardisierten Darstellung mit dem Steuerblock und den zwei Steuerleitungen *A(1:0).* Entsprechend der Werte an *A* wird einer der vier Dateneingänge *D(3:0)* ausgewählt und an den Ausgang *Y* gegeben. In der Mitte ist ein vereinfachtes Symbol dargestellt, welches in der Praxis häufig verwendet wird.

Ebenfalls in Abb. 6.2 gezeigt ist die Funktionstabelle für den 1-aus-4-Multiplexer. Die Leitung *A* gibt als Binärzahl an, welcher Eingangswert auf *Y* geschaltet wird. Es wird als ein Datenwert ausgewählt und die Schaltung wird darum auch als Datenselektor bezeichnet.

VDHL-Beschreibung
Ein Multiplexer kann durch das bereits bekannte Case-Statement erzeugt werden. Als Bedingung wird das Steuersignal *a* verwendet. Im Code sind die Signale definiert als:

- a : std_logic_vector(1 downto 0);
- d : std_logic_vector(3 downto 0);
- y : std_logic;

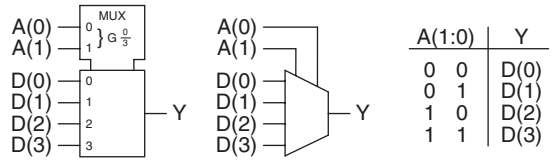


Abb. 6.2 Symbole und Funktionstabelle für 1-aus-4-Multiplexer

```
case a is
   when "00"   => y <= d(0);
   when "01"   => y <= d(1);
   when "10"   => y <= d(2);
   when others => y <= d(3);
end case;
```

Die Case-Anweisung kann nur innerhalb eines Prozesses aufgerufen werden. Die VHDL-Befehle für den Prozess werden hier und in den folgenden Beispielen zur besseren Übersichtlichkeit weggelassen. Der Others-Fall ist erforderlich, um alle möglichen Werte des Datentyps std_logic zu erfassen, also zum Beispiel auch ‚X' oder ‚U'.

6.2.2 Demultiplexer

Die entgegengesetzte Schaltung ist der *Demultiplexer*, kurz „Demux". Abhängig von Steuersignalen *A* wird ein Eingangssignal *D* auf einen von mehreren möglichen Ausgängen *Y* gelegt. Die anderen Ausgänge sind 0. Genau wie beim Multiplexer gibt es Varianten mit verschiedener Anzahl an Wahlmöglichkeiten, also 1-auf-2, 1-auf-4, 1-auf-8-Demultiplexer oder auch größere Schaltungen. Abb. 6.3 zeigt das Symbol und die Funktionstabelle für einen 1-auf-4-Demultiplexer.

Die Begriffe Multiplexer und Demultiplexer stammen von einer möglichen Anwendung, bei der sich mehrere Signalwege eine gemeinsame Leitung teilen. Dies ist in Abb. 6.4 dargestellt. Die Schaltungsstrukturen werden jedoch auch für andere Anwendungen eingesetzt.

Eine andere Bezeichnung für den Demultiplexer ist *Adressdecoder*. Dabei wählt eine Binärzahl mit n Stellen eine von 2^n Ausgangsleitungen und eine weitere Steuerleitung G aktiviert den Ausgang. Die Funktionstabelle für 8 Ausgangsleitungen ist in Abb. 6.5 dargestellt. Eine beispielhafte Anwendung ist, dass eine Adresse einen von 8 Speicherbausteinen

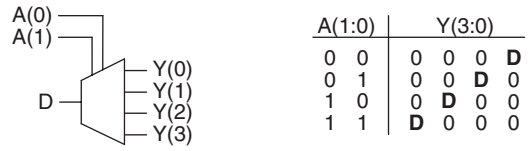


Abb. 6.3 Symbol und Funktionstabelle für 1-auf-4-Demultiplexer

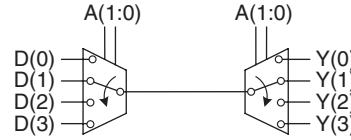


Abb. 6.4 Multiplexer und Demultiplexer

Abb. 6.5 Funktionstabelle eines Adressdecoders für 8 Leitungen

G	A(2:0)	Y(7)	Y(6)	Y(5)	Y(4)	Y(3)	Y(2)	Y(1)	Y(0)
0	X X X	0	0	0	0	0	0	0	0
1	0 0 0	0	0	0	0	0	0	0	**1**
1	0 0 1	0	0	0	0	0	0	**1**	0
1	0 1 0	0	0	0	0	0	**1**	0	0
1	0 1 1	0	0	0	0	**1**	0	0	0
1	1 0 0	0	0	0	**1**	0	0	0	0
1	1 0 1	0	0	**1**	0	0	0	0	0
1	1 1 0	0	**1**	0	0	0	0	0	0
1	1 1 1	**1**	0	0	0	0	0	0	0

auswählt. Die Schaltung entspricht exakt einem 1-auf-8-Demultiplexer des Datensignals *G*. Je nach Anwendungsgebiet ist die Bezeichnung als Adressdecoder jedoch verständlicher.

VHDL-Beschreibung

Auch der Demultiplexer kann durch ein Case-Statement erzeugt werden. Zunächst werden alle Ausgangssignale *y* auf 0 gesetzt. Der Eingang *d* wird dann einer der vier Ausgangsleitungen zugewiesen und damit für diesen Ausgang die Zuweisung der Null wieder überschrieben. Da die zwei Zuweisungen nacheinander innerhalb eines Prozesses ausgeführt werden, hat die erste Zuweisung für die Hardware-Synthese keine Wirkung.

Am ausgewählten Ausgang wird also *nicht* kurzzeitig eine 0 (erste Zuweisung) und danach auf der Wert von *d* (zweite Zuweisung) zu beobachten sein.

Die Signale sind definiert als:

- a : std_logic_vector(1 downto 0);
- d : std_logic;
- y : std_logic_vector(3 downto 0);

```
y <= "0000";
case a is
   when "00"   => y(0) <= d;
   when "01"   => y(1) <= d;
   when "10"   => y(2) <= d;
   when others => y(3) <= d;
end case;
```

6.2.3 Addierer

Arithmetische Berechnungen sind eine wichtige Grundfunktion von digitalen Schaltungen. Eine Grundschaltung für die Addition zweier Zahlen wird als *Addierer* bezeichnet. In diesem Abschnitt werden Addierer für Binärzahlen beschrieben. Die Addition von Zweierkomplementzahlen erfolgt mit der gleichen Struktur; lediglich das Vorzeichen muss berücksichtigt werden.

Zwei Zahlen der Wortbreite n ergeben eine Summe der Wortbreite $n + 1$, denn der Wertebereich der Summe kann ja größer als die Summanden sein. Für die Beispiele in diesem Abschnitt wird $n = 8$ gewählt, wenn nichts anderes angegeben ist.

Für diesen Fall der Wortbreite $n = 8$ haben die Summanden einen Wertebereich von [0,255]. Die Summe kann den Wertebereich von [0,510] haben und benötigt eine Wortbreite von $n = 9$.

Ein Addierer hat somit $2{\cdot}n$ Eingangsleitungen für die beiden Summanden A und B, sowie $n + 1$ Ausgangsleitungen für die Summe S. Ein Entwurf der Schaltung mit dem Karnaugh-Diagramm ist nicht möglich, da das Diagramm bei $2{\cdot}n$ Eingangsleitungen $2^{2{\cdot}n}$ Wertekombinationen hätte. Bei $n = 8$ wären diese $2^{16} = 65.536$ Einträge, also viel zu viel für eine grafische Optimierung

Ripple-Carry-Addierer

Für den Entwurf eines Addierers analysiert man die arithmetische Rechenoperation und setzt diese in eine Schaltung um. Zur Veranschaulichung ist in Abb. 6.6 die Addition zweier Zahlen dargestellt. Die Berechnung findet nacheinander für die einzelnen Binärstellen der Summanden A und B statt. Die beiden Werte werden mit dem *Übertrag* aus der vorherigen Stelle addiert und ergeben eine Summenstelle sowie einen Übertrag in die nächste Stelle. Der Übertrag zur ersten Stelle ist 0; der Übertrag der letzten Stelle ergibt die zusätzliche Summenstelle.

Diese Berechnung kann direkt in eine Schaltung umgesetzt werden. Die Berechnung für jede Stelle wird in einem Untermodul mit der Bezeichnung *Volladdierer* (VA) durchgeführt. Dieses Untermodul wird gleich noch beschrieben.

Für eine Addition von n Stellen werden n Volladdierer eingesetzt. Jeder Volladdierer erhält die beiden Stellen der Summanden sowie den Übertrag aus der vorherigen Stelle. Als Ausgabe des Volladdierers gibt es die Summe der aktuellen Stelle sowie den Übertrag für die nächste Stelle. Der erste Volladdierer hat am Eingang des Übertrags den Wert 0, denn die erste Stelle hat noch keinen Übertrag. Der Ausgang des Übertrags vom letzten Volladdierer ergibt die zusätzliche Summenstelle. Diese Struktur ist für $n = 8$ in Abb. 6.7 dargestellt. Der Übertrag (engl. *Carry*) läuft durch alle Stellen und darum wird diese Schaltungsstruktur als *Ripple-Carry-Addierer* (engl. *Ripple-Carry-Adder*) bezeichnet.

Es gibt noch weitere Addiererstrukturen, die für große Wortbreiten schneller arbeiten. Der Ripple-Carry-Addierer ist jedoch die wichtigste und am häufigsten vorkommende Addiererstruktur.

Volladdierer

Der Volladdierer hat drei Eingangssignale und zwei Ausgangssignale. Eingänge sind *A, B* und *CI,* also die beiden Binärstellen der Summanden sowie der Übertrag aus der

A		1	0	1	1	1	0	0	1
+ B		1	0	0	1	1	1	0	0
	1	0	1	1	1	0	0	0	Übertrag
S	1	0	1	0	1	0	1	0	1

Abb. 6.6 Addition zweier Binärzahlen der Wortbreite 8 bit

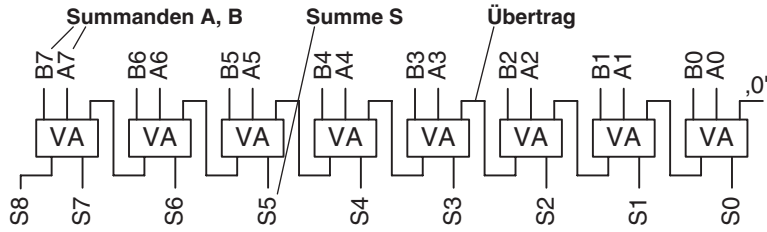


Abb. 6.7 Ripple-Carry-Addierer für Binärzahlen mit 8 Stellen

vorherigen Binärstelle mit der Bezeichnung *CI* für Carry-In. Ausgänge sind *S* und *CO,* also die Binärstelle der Summe sowie der Übertrag für die nächste Binärstelle mit der Bezeichnung *CO* für Carry-Out.

Die Schaltung muss die drei Eingangswerte *A, B, CI* summieren, was einen Wert von 0 bis 3 ergeben kann. Wenn diese Summe 2 oder 3 ist, erfolgt ein Übertrag in die nächste Stelle. Der Ausgang *S* wird 1, falls die Summe 1 oder 3 ist. Diese Funktion und das Symbol für einen Volladdierer ist in Abb. 6.8 gezeigt. Die Schaltung besteht aus wenigen Gattern.

Eine vereinfachte Form des Volladdierers ist der *Halbaddierer* (HA). Dieses Modul hat keinen Carry-Eingang und kann für die unterste Stelle des Ripple-Carry-Addierers verwendet werden.

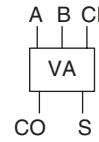
VHDL-Beschreibung

Der Addierer kann direkt durch das Plus-Zeichen erzeugt werden. Die Signale können als Vektor vom Typ signed/unsigned oder als integer definiert werden. Wie in Abschn. 3.5. erläutert, muss für signed und unsigned die Erweiterung der Wortbreite beachtet werden. Für Operanden *A* und *B* mit 8 bit Wortbreite lautet die Beschreibung:

```
s <=  '0' & a +  '0' & b;  -- für Datentyp unsigned
s <= a(7) & a + b(7) & b;  -- für Datentyp signed
```

Die Grundstruktur des Addierers wird auch für die Subtraktion eingesetzt. Prinzipiell wird statt des Volladdierers ein ähnlich definierter *Vollsubtrahierer* verwendet. In der Praxis wird jedoch oft einfach der Subtrahend invertiert und eine Addition durchgeführt. Damit ist kein weiteres Grundelement erforderlich. Es wird also $S = A + (-B)$

Abb. 6.8 Symbol und Funktionstabelle für einen Volladdierer



A	B	CI	CO	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

gerechnet. Damit die Invertierung dem Zweierkomplement entspricht, muss die 0 für den ersten Übertrag ebenfalls invertiert werden.

In VHDL wird die Subtraktion einfach durch das Minus-Zeichen aufgerufen. Für den Datentyp signed lautet die Beschreibung:

```
s <= a(7) & a - b(7) & b;
```

6.3 Sequenzielle Grundstrukturen

6.3.1 Zähler

Zähler sind wichtige Grundschaltungen, um Abläufe in digitalen Schaltungen zu steuern. Dabei wird als Grundoperation eine Binärzahl mit jedem Takt um eins erhöht. Auch ein Rückwärtszählen ist möglich, aber nicht so anschaulich. Gezählt wird stets ab dem Wert Null, also nicht ab Eins.

Die meisten Zähler beginnen nach Erreichen des letzten Ausgabewertes automatisch wieder beim ersten Ausgabewert, also der Null. Man bezeichnet dies als *Modulo-m Zähler,* wobei *m* die Anzahl der Zustände ist.

Beispielsweise zählt ein Modulo-5 Aufwärtszähler 0, 1, 2, 3, 4, 0, 1, 2, 3, 4, 0, 1, …

Besonders einfach sind *Modulo-2^n Zähler.* Sie durchlaufen alle *n*-stelligen Binärzahlen. Um alle 10-stelligen Binärzahlen zu durchlaufen wird ein Modulo-2^{10} Zähler eingesetzt. Er läuft von 0 bis 1023 und startet danach wieder bei 0.

Diese Grundfunktion kann verändert und durch verschiedene Steuersignale erweitert werden.

- **Enable:** Der Zähler geht nur zum nachfolgenden Wert, wenn ein Steuereingang *enable* $= 1$ ist.
- **Clear:** Der Zähler springt wieder auf den Startzustand. Dies entspricht einem Reset.
- **Load:** Der Zähler lädt auf einen Wert von einem Eingang.
- **Up/Down:** Die Zählrichtung kann gewählt werden, das heißt der Zähler zählt auf Wunsch in die negative Richtung.
- Kein automatischer Neustart, das heißt der Zähler hält beim Erreichen des Maximalwerts an und startet erst nach Clear erneut.

Ein ausführliches Beispiel für die Verwendung von Zählern folgt in Abschn. 6.5.2.

Implementierung

Ein Zähler wird mit der gleichen Struktur wie ein Moore-Automat implementiert. Der aktuelle Zählerstand ist in Flip-Flops gespeichert und aus diesem Wert sowie den Steuersignalen berechnet eine kombinatorische Schaltung den nächsten Zählerstand (Abb. 6.9).

Im einfachsten Fall besteht die kombinatorische Schaltung aus einem Addierer, der zum aktuellen Zählerstand den Wert Eins addiert. Je nach benötigten Steuersignalen sind weitere Gatter erforderlich.

VHDL-Beschreibung

Ein Zähler wird durch die Addition einer Zählvariablen in einem getakteten Prozess erzeugt. Besonders einfach ist der Modulo-2^n Zähler, wenn die Zählvariable als unsigned definiert ist. Bei Erreichen des Maximalwerts ist die Addition so definiert, dass sie danach wieder den Wert Null ergibt.

Für einen Modulo-2^{10} Zähler wird die Zählvariable count definiert als:

- count : unsigned(9 downto 0);

```
process
begin
wait until rising_edge(clk);
   count <= count + 1;
end process;
```

Zähler mit Steuersignalen lassen sich durch Erweiterung des Codes mit If-Bedingungen umsetzen. Der folgende Code beschreibt einen Modulo-100 Zähler, der nur bei *enable* = 1 zählt. Der Steuereingang *clear* setzt den Zähler auf 0 und zwar unabhängig von *enable* (siehe auch Abb. 6.9). Er entspricht einem synchronen Rücksetzen.

Die Steuereingänge sind als std_logic und die Zählvariable als unsigned definiert:

- clear: std_logic;
- enable: std_logic
- count : unsigned(6 downto 0);

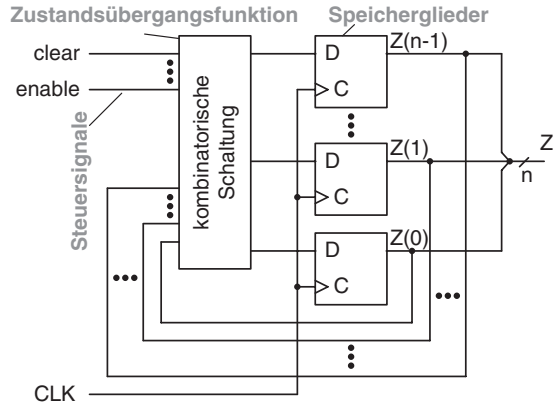


Abb. 6.9 Implementierung eines Zählers

```
process
begin
wait until rising_edge(clk);
   if clear = '1' then
      count <= (others => '0');
   elsif enable = '1' then
        if count = 99 then
           count <= (others => '0');
        else
            count <= count + 1;
        end if;
   end if;
end process;
```

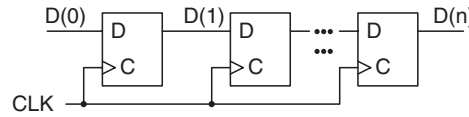
6.3.2 Schieberegister

Mehrere hintereinander geschaltete D-Flip-Flops werden als *Schieberegister* bezeichnet. In einem Schieberegister werden die gespeicherten Werte mit jedem Takt einen Wert weiter geschoben (Abb. 6.10).

Durch Steuersignale, beispielsweise ein Enable, kann die Grundstruktur erweitert werden. Ein Schieberegister wird verwendet, wenn Daten vor oder innerhalb einer Verarbeitung um wenige Takte verzögert werden sollen. Bei größeren Verzögerungen (ab ca. 16 Takte) sind jedoch Speicher meist effizienter.

Eine wichtige Anwendung von Schieberegistern ist die Verarbeitung serieller Daten und die Umwandlung zwischen seriellen und parallelen Daten. In Abb. 6.11 ist ein Schieberegister zur Wandlung paralleler Daten zur seriellen Datenübertragung

Abb. 6.10 Schieberegister



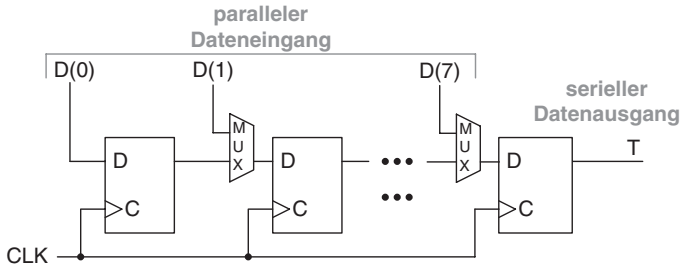


Abb. 6.11 Schieberegister zur Parallel-Seriell-Wandlung

dargestellt. Der Dateneingang D hat 8 Leitungen, die zu Beginn der Übertragung mit Multiplexern in ein Schieberegister geladen werden. Dann wird 8 Takte lang das Datenwort auf die serielle Leitung T ausgegeben. Nach diesen 8 Takten kann das nächste Datenwort übertragen werden.

VDHL-Beschreibung

Für die VHDL-Beschreibung eines Schieberegisters wird die Zusammenfassung von Vektoren mit dem Concatenation-Operator & verwendet. Achtung: Verwechseln Sie diesen Operator nicht mit der UND-Verknüpfung.

Für ein einfaches Schieberegister ähnlich wie in Abb. 6.10 wird ein std_logic_vector definiert. Hier soll als Wortbreite 8 bit verwendet werden und ein Steuereingang *enable* beachtet werden. Der oberste Wert des Schieberegisters, das MSB (Most Significant Bit), wird nicht mehr gespeichert. Die übrigen Werte rücken eine Stelle auf und werden mit dem neuen Eingangswert *data* ergänzt. Dies wird programmiert, indem der Wert des Schieberegisters ohne MSB (also *d(6:0)*) mittels Concatenation um das Signal *data* ergänzt wird.

Die Signale sind definiert als:

- d : std_logic_vector(7 downto 0);
- data: std_logic;
- enable : std_logic;

```
process
begin
wait until rising_edge(clk);
   if enable = '1' then
      d <= d(6 downto 0) & data;
  end if;
end process;
```

6.3.3 Rückgekoppeltes Schieberegister

Bei einem *rückgekoppelten Schieberegister* werden einige Stellen XOR-verknüpft und wieder in das Schieberegister gegeben. Die englische Bezeichnung hierfür ist *Linear Feedback Shift Register* oder *LFSR*. Abb. 6.12 zeigt ein LFSR mit 4 Stellen. Die Daten an Position 3 und 4 werden XOR-verknüpft und wieder an Position 0 in das Schieberegister gegeben.

Bei geeigneter Wahl der Rückkopplung werden bei einem n-Bit-Schieberegister 2^n-1 verschiedene Zustände durchlaufen. Von den 2^n möglichen Kombinationen treten also sämtliche Werte auf, ausgenommen alle Stellen auf 0. Die Werte treten dabei nicht in der

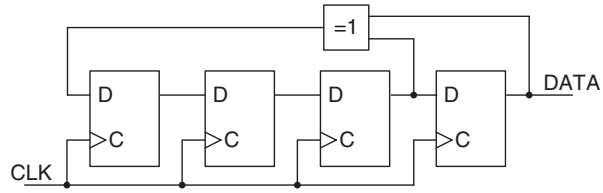


Abb. 6.12 Rückgekoppeltes Schieberegister mit 4 Stellen

arithmetischen Reihenfolge auf und können darum auch als Pseudo-Zufallszahlen genutzt werden. Die Initialisierung muss vermeiden, dass alle Werte auf 0 sind (nicht in Abb. 6.12 dargestellt).

Die Abgriffe der Rückkopplung sind für verschiedene Längen des Schieberegisters in Tabellen angegeben. Eingesetzt werden LFSR beispielsweise als Zahlengeneratoren in der Kommunikationstechnik.

6.4 Zeitverhalten

6.4.1 Verzögerungszeit realer Schaltungen

Logikgatter benötigen eine kurze Laufzeit bis der Ausgang auf eine Änderung der Eingangsvariablen reagiert. Diese Laufzeit ist abhängig von der Technologie. Für ein einzelnes Gatter in einem Gehäuse kann die Laufzeit über 10 ns betragen. Als Teil eines modernen hochintegrierten ASICs sind Laufzeiten unter 0,1 ns möglich.

Realistische Werte für die Verzögerungszeit eines Gatters innerhalb integrierter Schaltungen betragen etwa 0,1 bis 1,0 ns, während die Verzögerungszeit diskreter Gatter bei etwa 1 ns bis 10 ns liegt (vgl. Kap. 7). Dabei ist die Verzögerungszeit auch abhängig von der Funktion des Logikgatters. Ein Inverter ist meist schneller als ein ODER-Gatter mit 8 Eingängen. Auch gleichartige Gatter können eine unterschiedliche Laufzeit haben, abhängig beispielsweise davon, ob ihr Ausgang 1 oder 10 weitere Gatter ansteuert.

6.4.2 Transiente Signalzustände

Beim Wechsel einer oder mehrerer Eingangsvariablen treten aufgrund der Verzögerungszeiten oft kurze Zwischenzustände auf. Diese werden als *Spike, Glitch* oder *Hazard* bezeichnet.

Zum besseren Verständnis wird die Schaltung in Abb. 6.13 betrachtet. Bei ihr wechselt der mittlere Eingang von 1 auf 0. Für beide Eingangswerte ist der Ausgang Y auf 1. Durch Verzögerungszeiten der Gatter kann jedoch ein kurzzeitiger Spike am Ausgang Y auftreten. Dieser entsteht durch den folgenden Ablauf:

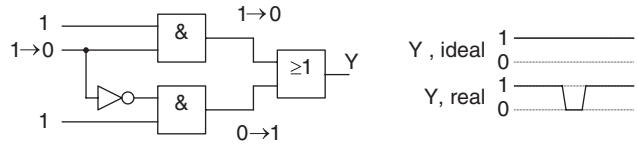


Abb. 6.13 Spike beim Wechsel eines Eingangssignals

- Der mittlere Eingang wechselt von 1 auf 0.
- Das obere UND-Gatter wechselt dadurch von 1 auf 0.
- An beiden Eingängen des ODER-Gatters liegt 0 an und der Ausgang ist kurzzeitig 0.
- Das untere UND-Gatter ist durch den vorgeschalteten Inverter etwas langsamer als das obere UND-Gatter und wechselt später von 0 auf 1.
- Der untere Eingang des ODER-Gatters wird 1 und der Ausgang wird wieder 1.

6.4.3 Signalübergänge in komplexen Schaltungen

Bei komplexen Schaltungen können auch mehrere Übergänge auftreten, bis der endgültige Ausgangswert erreicht ist. Dies lässt sich beim Ripple-Carry-Addierer aus Abb. 6.7 gut nachvollziehen. Eine Summenstelle hängt von den Eingangswerten der aktuellen Stelle sowie von allen tieferen Stellen ab. Summenstelle *S(6)* beispielsweise hängt von *A(6)* und *B(6)* sowie dem Übertrag aus allen vorherigen Stellen also *A(5)* bis *A(0)* und *B(5)* bis *B(0)* ab. Wenn zwei neue Zahlen für die Berechnung am Addierer anliegen, liegt an Stelle 6 die Information von *A(6)* und *B(6)* sofort an. Die Informationen aus den vorherigen Stellen müssen jedoch erst durch mehrere Volladdierer weitergegeben werden und treffen später an der Stelle 6 ein.

Als ein Beispiel werden die Signalwechsel des Ripple-Carry-Addierers simuliert. An den Eingängen *A* und *B* liegen zunächst die Werte 85_{10} und 170_{10}, also binär 01010101_2 und 10101010_2 an. Dann wechseln die Werte auf 171_{10} und 85_{10}, binär 10101011_2 und 01010101_2. Als Verzögerungszeit für einen Volladdierer wird 0,3 ns angenommen. Außerdem wird angenommen, dass der Eingang B eine etwas längere Anschlussleitung und dadurch eine zusätzliche Laufzeit von 0,1 ns hat.

Das Ergebnis der Simulation ist in Abb. 6.14 zu sehen. Die Summe wechselt von 255_{10} auf 256_{10}, binär von 011111111_2 auf 100000000_2. Durch die schrittweise Verarbeitung des Übertrags wechseln die höheren Summenausgänge *S* mehrfach den Wert.

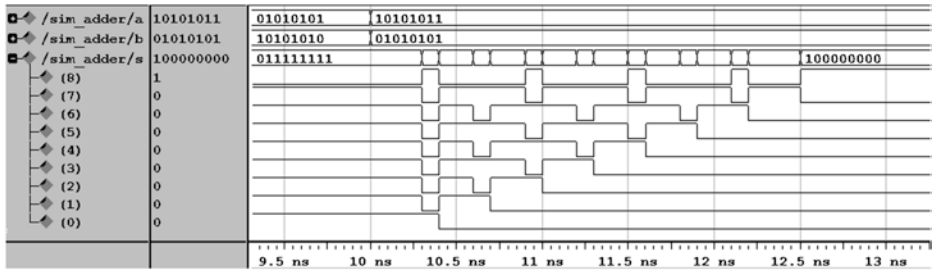


Abb. 6.14 Simulation eines Ripple-Carry-Addierers (VHDL-Simulator Modelsim)

6.5 Taktkonzept in realen Schaltungen

6.5.1 Register-Transfer-Level (RTL)

Der mehrfache Wechsel von Signalzuständen lässt sich in Digitalschaltungen kaum vermeiden. Er stellt aber auch kein Problem dar, denn fast alle Schaltungen werden durch einen Takt gesteuert. Das allgemeine *Taktkonzept* ist in Abb. 6.15 dargestellt. Die Eingangssignale einer Schaltung werden zunächst in Flip-Flops gespeichert. Die Flip-Flop-Ausgänge werden dann in einer kombinatorischen Schaltung verarbeitet. Dabei können mehrfache Signalwechsel auftreten. Wenn alle Wechsel der kombinatorischen Schaltung erfolgt sind, werden die Informationen in einer zweiten Flip-Flop-Stufe gespeichert. Von dort werden die Daten in der nachfolgenden Taktperiode an die nächste kombinatorische Schaltung gegeben, nach der sich erneut eine Flip-Flop-Stufe befindet.

Die Flip-Flop-Stufen werden auch als *Register* bezeichnet und als kompakte Darstellung das in Abb. 6.15 gezeigte Schaltsymbol verwendet. Das Taktkonzept bezeichnet man als *Register-Transfer* und diese Schaltungsstruktur ermöglicht ein sicheres Arbeiten

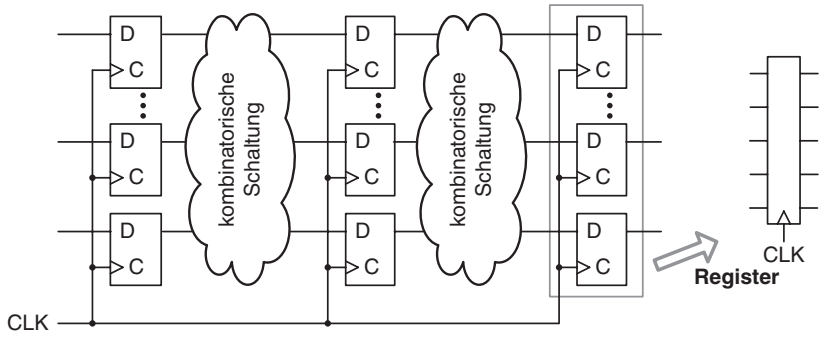


Abb. 6.15 Taktkonzept Register-Transfer

der Schaltung, da die Register jeweils abwarten, bis alle Signalübergänge in der kombinatorischen Schaltung erfolgt sind.

Ein wesentlicher Vorteil dieses Schaltungskonzepts ist auch die Übersichtlichkeit beim Schaltungsentwurf. Beim Entwurf kann man sich gut vorstellen, welche Informationen jeweils in einer Registerstufe vorhanden sind. Daraus kann man dann beschreiben, was im nächsten Schritt mit diesen Informationen passieren soll. Die Entwurfsmethodik ist weit verbreitet und wird als *Register-Transfer-Level* (RTL) bezeichnet.

6.5.2 Beispiel für Entwurf mit Register-Transfer-Level: Ampelsteuerung

Der Entwurf im Register-Transfer-Level, kurz RTL-Design, wird mit einem ausführlichen Beispiel verdeutlicht. Dabei werden auch die Grundstrukturen aus Abschn. 6.3 verwendet.

Aufgabenstellung
Eine Fußgängerampel soll durch eine Digitalschaltung angesteuert werden. Die Straße hat eine Ampel mit drei Lichtzeichen Rot, Gelb, Grün und der Fußgängerüberweg eine Ampel mit zwei Lichtzeichen Rot, Grün (Abb. 6.16). Um die Schaltung einfach zu halten, sollen keine Tasten ausgewertet werden, sondern stets folgender Ablauf stattfinden:

- 10 s grün für die Straße
- 1 s gelb für die Straße
- 1 s rot für die Straße
- 5 s grün für die Fußgänger
- 2 s rot für die Fußgänger
- 1 s rot und gelb für die Straße
- Zyklus beginnt erneut

Für eine echte Fußgängerampel wäre diese Steuerung sicher zu einfach, deswegen nehmen wir an, die Schaltung sei für eine Modelleisenbahn.

Die Digitalschaltung verwendet einen Takt mit der Frequenz 50 MHz.

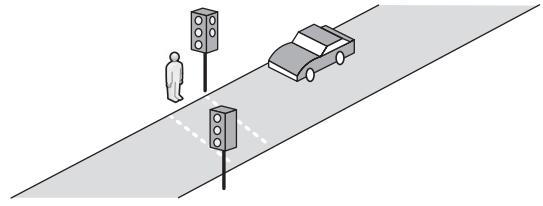


Abb. 6.16 Einfache Fußgängerampel

Struktur

Der Entwurfsablauf beim RTL-Design ist so, dass die Aufgabe zunächst in einzelne Teilschritte unterteilt wird. Diese Teilschritte werden dann zwischen den Registern berechnet. Für die Ampelsteuerung sind drei Teilschritte sinnvoll.

1. Aus dem Takt (50 MHz) wird ein Sekundensignal erzeugt.
2. Mit dem Sekundensignal werden die 20 Schritte des Ampelablaufs gezählt.
3. Mit der Information, welcher Schritt des Ampelablaufs vorliegt, werden die Lichtsignale ausgegeben.

VHDL-Beschreibung

Die Schaltung könnte prinzipiell mit einem Moore-Automaten umgesetzt werden. Es ist jedoch deutlich einfacher und übersichtlicher, wenn die Grundstrukturen Zähler und Multiplexer verwendet werden. Im Folgenden ist der komplette VHDL-Code inklusive Bibliotheksaufruf und Entity angegeben.

Das Eingangssignal *clock_50* ist der Takt von 50 MHz. Die Ausgangssignale sind *strasse* mit drei Werten für rotes, gelbes, grünes Licht (gezählt von MSB nach LSB) sowie *fussweg* mit zwei Werten für rotes und grünes Licht (MSB und LSB). Beim Wert „001“ für *strasse* ist also der drittgenannte Wert, das grüne Licht aktiv. Beim Wert „10“ für *fussweg* ist der erstgenannte Wert aktiv, also das rote Licht.

Die drei Schritte des Register-Transfer-Levels sind durch die Kommentarzeilen gekennzeichnet.

1. Der erste RTL-Schritt ist ein Zähler, der mit *count_a* 50 Mio. Werte zählt und dann *enable* für einen Takt auf 1 setzt. Die benötigte Wortbreite des Zählers berechnet sich aus dem Zweierlogarithmus von 50.000.000 und ergibt aufgerundet 26 bit.

$$ld 50\,000\,000 = \log 50\,000\,000 / \log 2 = 7,699 / 0,301 = 25,58$$

2. Der zweite RTL-Schritt ist ebenfalls ein Zähler, der durch *enable* einmal pro Sekunde aktiviert wird. Er zählt mit *count_b* die 20 Schritte des Ampelzyklus. Die benötigte Wortbreite beträgt 5 bit.
3. Der dritte RTL-Schritt ist eine Fallunterscheidung, codiert als If-Anweisung, die aus dem Wert von *count_b* die Ansteuerung der Ampellichter ermittelt.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity ampel is
   port (clock_50 : in  std_logic;
         strasse  : out std_logic_vector(2 downto 0); -- rot, gelb, grün
         fussweg  : out std_logic_vector(1 downto 0));-- rot, grün
end;
```

```
architecture behave of ampel is

signal  enable  : std_logic;
signal  count_a : unsigned(25 downto 0);
signal  count_b : unsigned(4 downto 0);

begin

process
begin
wait until rising_edge(clock_50);

   -- Zähler für 1 Impuls je Sekunde
   if count_a >= 49999999 then
      count_a <= (others => '0');
      enable  <= '1';
   else
      count_a <= count_a + 1;
      enable  <= '0';
   end if;

   -- Zähler für 20 Schritte der Ampel
   if enable = '1' then
        if count_b >= 19 then
           count_b <= (others => '0');
        else
            count_b <= count_b + 1;
        end if;
   end if;

   -- Abfrage für Lichter der Ampel
   if      count_b < 10 then
        -- 10 Sekunden grün für Straße, rot für Fussweg
        strasse <= "001"; fussweg <= "10";
   elsif  count_b < 11 then
        -- 1 Sekunde gelb für Straße
        strasse <= "010"; fussweg <= "10";
   elsif  count_b < 12 then
        -- 1 Sekunde rot für Straße
        strasse <= "100"; fussweg <= "10";
   elsif  count_b < 17 then
        -- 5 Sekunden grün für Fußweg
        strasse <= "100"; fussweg <= "01";
   elsif  count_b < 19 then
        -- 2 Sekunden rot für Fußweg
```

```
        strasse <= "100"; fussweg <= "10";
    else
        -- 1 Sekunde rot/gelb für Straße
        strasse <= "110"; fussweg <= "10";
    end if;
end process;
end;
```

6.5.3 Kritischer Pfad

Die Speicherung in einer Flip-Flop-Stufe darf erst erfolgen, wenn alle Wechsel der kombinatorischen Schaltung abgelaufen sind. Hierfür muss die maximale Verzögerungszeit der kombinatorischen Schaltung berechnet werden. Der langsamste Weg durch die Schaltung wird als *kritischer Pfad* bezeichnet. Ein Pfad beginnt bei einem Flip-Flop-Ausgang und endet bei einem Flip-Flop-Eingang.

Als Beispiel ist in Abb. 6.17 der kritische Pfad eines Ripple-Carry-Addierers dargestellt (vergleiche Abb. 6.7). Die Summanden *A* und *B* sowie die Summe *S* werden entsprechend der RTL-Methodik in Flip-Flop-Stufen gespeichert. Der kritische Pfad beginnt bei der untersten Stelle eines Summanden und endet bei der höchsten Stelle des Ergebnisses. Dazwischen müssen die acht Volladdierer des Ripple-Carry-Addierers durchlaufen werden.

Für die Verzögerungszeit des kritischen Pfads werden die Verzögerungszeiten aller Gatter sowie die Signallaufzeiten der Leitungen addiert. Außerdem hat auch der Ausgang des Flip-Flops am Start des Pfads eine Verzögerungszeit. Beim Flip-Flop am Ende des Pfads muss die Setup-Zeit eingehalten werden, also die Zeit vor der steigenden Taktflanke, in der das Eingangssignal stabil sein muss (siehe Kapitel 5).

Als ein Beispiel wird der kritische Pfad des Ripple-Carry-Addierers berechnet. Dazu werden folgende Verzögerungszeiten angenommen.

- Verzögerungszeit eines Volladdierers: 0,3 ns
- Setup-Zeit eines Flip-Flops: 0,2 ns

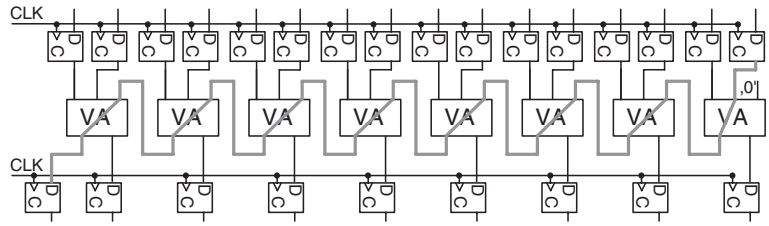


Abb. 6.17 Kritischer Pfad eines Ripple-Carry-Addierers

- Verzögerungszeit von Takt nach Flip-Flop-Ausgang: 0,2 ns
- Laufzeit einer Leitung: 0,1 ns

Für einen 8-Bit-Addierer besteht der kritische Pfad dann aus:

- Flip-Flop-Ausgang: 0,2 ns
- 8 Volladdierer: 8·0,3 ns = 2,4 ns
- 9 Verbindungsleitungen: 9·0,1 ns = 0,9 ns
- Flip-Flop Setup-Zeit: 0,2 ns

Dies ergibt in Summe 3,7 ns.

Für einen 32-Bit-Addierer müssen 32 Volladdierer und 33 Verbindungsleitungen berücksichtigt werden. Der kritische Pfad beträgt dann 13,3 ns.

In der Praxis wird der kritische Pfad durch Entwurfsprogramme ermittelt, indem sämtliche Pfade der Schaltung berechnet werden. In Abb. 6.17 könnte auch der andere Eingang des ersten Volladdierers sowie der andere Ausgang des letzten Volladdierers Anfang und Ende des kritischen Pfads sein. Dies hängt von den Verbindungsleitungen und dem inneren Aufbau der Volladdierer ab.

6.5.4 Pipelining

Mögliche Taktfrequenz

Aus dem kritischen Pfad kann als Kehrwert die mögliche Taktfrequenz berechnet werden.

- Der 8-Bit-Ripple-Carry-Addierer hat im kritischen Pfad eine Verzögerungszeit von 3,7 ns. Die maximal mögliche Taktfrequenz beträgt darum 1/(3,7 ns) = 270 MHz.
- Für den 32-Bit-Ripple-Carry-Addierer mit der Verzögerungszeit von 13,3 ns beträgt die maximal mögliche Taktfrequenz 75 MHz.

Oft ist jedoch die Vorgehensweise anders herum, das heißt für eine Problemstellung ist die erforderliche Taktgeschwindigkeit vorgegeben. Sie ergibt sich entweder direkt aus der Aufgabe oder aus der Leistungsfähigkeit von Konkurrenzprodukten. Der kritische Pfad muss dann diese Vorgabe erfüllen.

Dies wird durch die beiden folgenden Zahlenbeispiele verdeutlicht:

- Eine digitale Schaltung soll Radarsignale analysieren, die mit 100 Mio. Werten pro Sekunde auftreten. Die Schaltung muss daher eine Taktfrequenz von 100 MHz erreichen. Der kritische Pfad darf 10 ns betragen.
- Ein Mikrocontroller soll entworfen werden. Die vorhandenen Produkte arbeiten mit bis zu 200 MHz. Für das neue Produkt wird daher eine Taktfrequenz von 250 MHz spezifiziert. Der kritische Pfad darf 4 ns betragen.

Taktfrequenz und kritischer Pfad

Die Analyse des kritischen Pfads und der Vergleich mit der geforderten Taktfrequenz zeigen, ob die Geschwindigkeitsanforderungen an die Schaltung eingehalten werden. Wenn die Geschwindigkeit ausreicht, ist normalerweise keine weitere Optimierung erforderlich. Falls der kritische Pfad jedoch länger als die verfügbare Zeit ist, muss die Schaltung optimiert werden.

Zur Verkürzung des kritischen Pfads kann überlegt werden, ob die Verarbeitung einfacher aufgebaut oder in mehr Teilschritte aufgeteilt werden kann.

Beispielsweise wird in der Ampelsteuerung aus Abschn. 6.5.2 ein Zähler bis 50 Mio. eingesetzt. Falls dieser Zähler nicht mit der geforderten Taktfrequenz arbeitet, könnte er in zwei nacheinander geschaltete Zähler bis 10.000 und 5.000 aufgeteilt werden.

Einfügen von Flip-Flop-Stufen

Wenn eine Schaltung nicht umstrukturiert werden kann oder soll, lässt sich durch das Einfügen von Flip-Flop-Stufen die Verarbeitungsgeschwindigkeit erhöhen. Dies wird als *Pipelining* bezeichnet und in digitalen Schaltungen sehr häufig eingesetzt.

Abb. 6.18 zeigt das Einfügen einer Pipeline-Stufe. Die kombinatorische Logik wird durch einen Schnitt aufgeteilt und in sämtliche Verbindungsleitungen werden Flip-Flops eingefügt. Wichtig ist, dass tatsächlich alle Signale gleich verzögert werden, da die Informationen sonst zeitlich gegeneinander verschoben wären. In Abb. 6.18 wird die

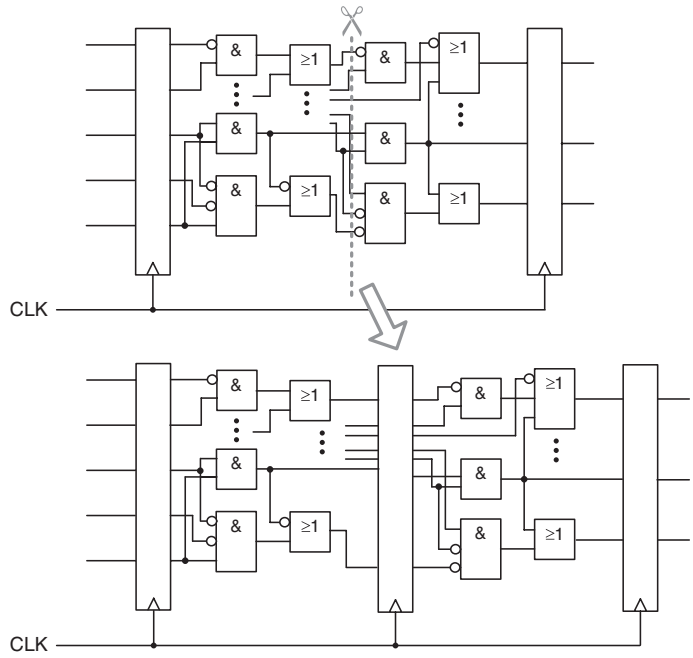


Abb. 6.18 Einfügen einer Pipeline-Stufe

Pipeline-Stufe bereits nach zwei Gattern eingefügt. Je nach Geschwindigkeitsanforderungen kann aber auch erst nach 10 oder 20 Gattern eine Pipeline-Stufe erforderlich sein. Pipelining verkürzt nicht die Gesamtlaufzeit durch die Kombinatorik, sondern die Laufzeit zwischen Flip-Flop-Stufen. Ein kritischer Pfad von beispielsweise 10 ns wird durch Pipelining in zwei Teile zu 5 ns aufgeteilt und die Schaltung kann dadurch mit 200 MHz statt mit 100 MHz betrieben werden. Allerdings dauert die Berechnung dann zwei Taktzyklen. Die gesamte Verzögerung einer Berechnung wird als *Latenzzeit* bezeichnet.

Der Vorteil des Pipelinings ist, dass während einer Berechnung in der zweiten Pipeline-Stufe, bereits die nächsten Werte in die erste Pipeline-Stufe gegeben werden können. Die Anzahl an Rechenzyklen wird als Durchsatz bezeichnet. Die Schaltung mit 100 MHz Takt hat einen *Durchsatz* von 100 Mio. Datenwerten, während bei 200 MHz der Durchsatz 200 Mio. Datenwerte beträgt. Pipelining bewirkt also eine Steigerung der Verarbeitungsleistung.

6.5.5 Taktübergänge

Taktbereiche

Bisher wurde überall in einer Schaltung der gleiche Takt verwendet. Dies ist auch möglichst anzustreben. Allerdings lässt sich nicht immer vermeiden, dass mehrere Takte verwendet werden. Ein Beispiel dafür ist ein PC:

- Die CPU arbeitet mit einem Takt zwischen 3 und 4 GHz.
- Der DRAM-Speicher arbeitet mit einem Takt im Bereich von 1 GHz.
- Die Grafikkarte arbeitet mit einem Takt zwischen 500 und 1000 MHz.
- Peripheriebausteine für LAN und USB nutzen eigene Taktsignale.

Die *Taktbereiche* werden auch als *Clock-Domain* bezeichnet. Beim Übergang zwischen Clock-Domains kann eine fehlerhafte Datenübernahme auftreten, die durch spezielle Schaltungsstrukturen verhindert werden muss.

Fehler bei Taktübergängen

Ein Fehler bei Taktübergängen tritt auf, wenn eine Information an mehreren Stellen einen Taktübergang hat. Zur Verdeutlichung des Problems ist in Abb. 6.19 eine Schaltung zur *Flankenerkennung* dargestellt. Das Signal A kommt aus einem anderen Taktbereich und die Schaltung soll erkennen, wenn es einen Übergang von 0 nach 1. Dies soll angezeigt werden, indem der Ausgang Q für einen Takt auf 1 gesetzt wird.

Die Funktionsweise der Flankenerkennung ist:

- Der Eingang A wird in einem Flip-Flop gespeichert. B ist somit der Wert des Eingangs A aus dem vorherigen Takt.

- Es wird überprüft, ob *A* im letzten Takt 0 war und jetzt 1 ist. Dies erfolgt durch ein UND-Gatter, welches nur 1 ist, wenn *A* auf 1 und *B* auf 0 ist (invertierter Eingang des Gatters).
- Das Ergebnis des UND-Gatters, Signal *C* wird in einem Flip-Flop gespeichert.
- Der Ausgang des Flip-Flops ist die gewünschte Flankenerkennung.

Die Schaltung ist relativ übersichtlich und das Zeitdiagramm zeigt, wie der Ablauf zu dem geplanten Verhalten führt. Ein Fehler tritt jedoch auf, wenn das Signal *A* sich nicht zu dem angenommenen Zeitpunkt ändert. Dies ist möglich, da *A* ja aus einer anderen Clock-Domain stammt.

Das fehlerhafte Verhalten ist in Abb. 6.20 dargestellt.

- Der Eingang *A* ändert sich kurz vor der Taktflanke.
- Das Flip-Flop für den Wert *B* übernimmt den neuen Wert noch.
- Das UND-Gatter hat eine kurze Verzögerung, sodass der Wert *C* nicht mehr vom Flip-Flop übernommen wird.
- Nach der Taktflanke hat das Flip-Flop für *B* schon den neuen Wert übernommen. Darum liegt an beiden Eingängen des UND-Gatters der Wert 1 an und es wird keine Flanke erkannt.

Auslöser des Fehlers ist die unbekannte Zeitbeziehung zwischen *A* und den Takt *CLK*. Da das Signal *A* aus einem anderen Taktbereich kommt, wechselt es manchmal in ausreichendem Abstand und manchmal fast gleichzeitig zum Taktsignal *CLK*. Schwierig für

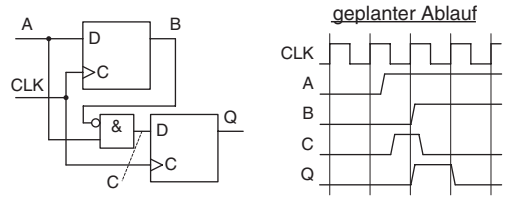


Abb. 6.19 Schaltung zur Flankenerkennung

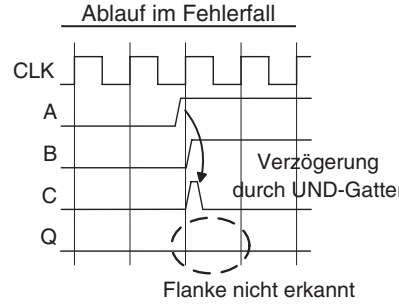


Abb. 6.20 Fehlerhafter Ablauf bei Flankenerkennung

die Fehlersuche ist, dass der Fehler nicht immer auftritt. Es ist gut möglich, dass 95 % der Taktflanken erkannt werden und nur für 5 % der Übergänge ein Fehler auftritt.

Korrekte Taktübernahme
Die Vermeidung des Fehlers erfolgt dadurch, dass ein Taktübergang nur an einer Stelle in der Schaltung erfolgen darf. Es muss also verhindert werden, dass das Signal *A* aus einem fremden Taktbereich die Eingangswerte für beide (!) Flip-Flops beeinflusst. Dies kann man in einer Schaltung erreichen, indem der Eingang *A* zunächst mit einem Flip-Flop in die Clock-Domain übernommen wird. In Abb. 6.21 wird *A* zunächst als *A_SYNC* in den Taktbereich von *CLK* übernommen. Damit ist die Zeitbeziehung von *A* zum Takt sichergestellt und es kann kein fehlerhafter Ablauf auftreten. Signal *B* wurde zur besseren Lesbarkeit umbenannt.

VHDL-Beschreibung
In der Praxis wird die Schaltung aus Abb. 6.21 natürlich in VHDL entworfen. Das UND-Gatter ergibt sich aus der If-Anweisung.

```
process
begin
wait until rising_edge(clk);
   a_sync      <= a;
   a_sync_old  <= a_sync;
   if (a_sync_old='0') and (a_sync='1') then
        q <= '1';
   else
        q <= '0';
   end if;
end process;
```

6.5.6 Metastabilität von Flip-Flops

Ein weiteres Problem bei der Taktübernahme ist die Einhaltung der Setup- und Hold-Zeiten (siehe Kapitel 5). Damit ein Flip-Flop Daten korrekt übernimmt, muss der Eingang kurz vor (Setup) bis kurz nach (Hold) der Taktflanke unverändert sein. Wenn sich Daten unabhängig vom Takt ändern, wird diese Bedingung nicht immer eingehalten.

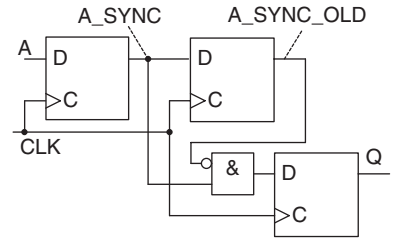


Abb. 6.21 Flankenerkennung mit sicherer Datenübernahme beim Taktübergang

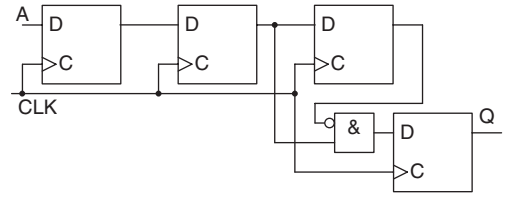


Abb. 6.22 Flankenerkennung mit Synchronisation gegen Metastabilität

Zunächst kann nicht vorhergesagt werden, ob noch der alte oder schon der neue Signalwert vom Flip-Flop nach *A_SYNC* (Abb. 6.21) übernommen wird. Diese Unsicherheit wäre kein Problem, da die Empfangsschaltung ohnehin nicht weiß, wann die Eingangsdaten übergeben werden und einen Zeitversatz berücksichtigen muss. Allerdings kann der Fall eintreten, dass ein Flip-Flop in der Mitte zwischen 0 und 1 „hängt“. Dieser Zwischenzustand wird als *Metastabilität* bezeichnet. Er tritt sehr selten auf, kann jedoch einen Fehler in der Verarbeitung verursachen.

Als Schutz gegen Metastabilität wird empfohlen, ein Signal beim Übergang in einen anderen Taktbereich mit zwei hintereinandergeschalteten Flip-Flops zu übernehmen (Abb. 6.22). Erst danach darf das Signal im Taktbereich verwendet werden. Ein metastabiles Signal des ersten Flip-Flops würde vom zweiten Flip-Flop nicht übernommen werden.

Allerdings erhöht sich durch das zweite Flip-Flop die Latenzzeit, also die Reaktionszeit auf den Eingang. Eine Synchronisation gegen Metastabilität wird darum nicht in allen Anwendungen eingesetzt.

6.5.7 Taktübergang mehrerer Signale

Schwieriger ist der Fall, wenn mehrere Signale gleichzeitig übernommen werden müssen. Wenn sich Daten unabhängig vom Empfangstakt ändern, ist nicht sicher, ob alle zusammengehörigen Informationen mit der gleichen Taktflanke gespeichert werden. Bei einem 8-Bit-Wert könnte es beispielsweise passieren, dass Bit 0 noch rechtzeitig von einer Taktflanke übernommen wird, Bit 1 jedoch erst von der nächsten Taktflanke. Dies ist ein Problem, da die Informationen eines Datenworts so auseinandergezogen werden.

Zur Vermeidung dieses Fehlers gibt es mehrere Möglichkeiten:

- Warten auf langsamste Information: Die empfangende Schaltung kann ein oder zwei Takte warten, bis alle Stellen einer Information anliegen und erst dann die Daten auswerten. Dies ist relativ einfach, aber nur möglich, wenn sich die Daten deutlich langsamer als der Takt ändern.

- Vermeidung mehrerer Signalwechsel: Wenn Daten eine feste Reihenfolge haben, beispielsweise bei einem Zähler, kann die Codierung so erfolgen, dass sich immer nur ein Wert im Datenwort ändert. Ein möglicher Code hierfür ist der Gray-Code (siehe Kapitel 2)
- Dual-Port-Speicher: Eine universelle Lösung ist ein Dual-Port-Speicher. In ihm kann mit einem Takt geschrieben und mit einem anderen gelesen werden. Die interne Steuerung sorgt für eine sichere Trennung der Taktbereiche. Für die Verwaltung des Speichers (z. B. Füllstand) werden dann häufig Zähler auf Basis des Gray-Codes eingesetzt.

6.6 Spezielle Ein-/Ausgangsstrukturen

Für Ein- und Ausgänge von digitalen Schaltungen gibt es spezielle Schaltungsstrukturen.

6.6.1 Schmitt-Trigger-Eingang

Digitale Signale werden ja durch Spannungspegel dargestellt. Dabei gibt es einen Bereich für den Low-Pegel und einen Bereich für den High-Pegel. Dazwischen ist ein Übergangsbereich, in dem das Signal undefiniert ist (vgl. Kapitel 1).

Der Übergangsbereich wird innerhalb digitaler Schaltungen normalerweise schnell durchlaufen. Am Eingang einer Schaltung kann es jedoch vorkommen, dass der Übergangsbereich langsamer durchlaufen wird und durch Rauschen gestört ist. Eine Digitalschaltung könnte dadurch mehrfach einen Pegelwechsel erkennen, was normalerweise nicht gewünscht ist.

Dieses Problem wird durch einen *Schmitt-Trigger* behoben. Ein Schmitt-Trigger hat eine Hysterese und behält einen Ausgangswert so lange, bis sich der Eingangswert deutlich ändert. Bei einem Eingangssignal im Übergangsbereich wird der vorhandene Ausgangswert beibehalten.

Das Symbol eines Schmitt-Triggers enthält zur Kennzeichnung eine Hysteresekurve (Abb. 6.23). In Abb. 6.23 ist das Zeitverhalten eines Schmitt-Triggers dargestellt.

- Eingang *A* hat zunächst Low-Pegel (L) und der Ausgang *Y* ist somit logisch 0.

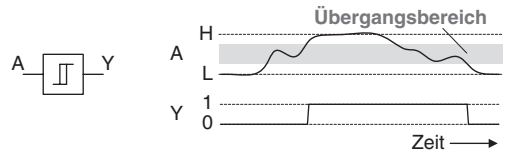


Abb. 6.23 Symbol und Zeitverhalten eines Schmitt-Triggers

- Der Eingang *A* geht dann in den Übergangsbereich, in dem eine normale Digitalschaltung ein undefiniertes Verhalten zeigen würde. Der Ausgang *Y* des Schmitt-Triggers bleibt jedoch auf logisch 0.
- Wenn *A* sich im Spannungsbereich des High-Pegels (H) befindet, wechselt auch *Y* auf logisch 1.
- *A* geht wieder in den Übergangsbereich, doch *Y* bleibt noch logisch 1.
- Erst wenn *A* wieder im Low-Pegel ist, wechselt auch *Y* auf logisch 0.

6.6.2 Tri-State-Ausgang

Digitale Ausgänge dürfen im allgemeinen Fall nicht miteinander verbunden werden. Wenn eine Leitung 0 und eine andere 1 ausgibt, fließt ein Kurzschlussstrom und der Logik-Pegel ist nicht eindeutig. Für den Einsatz in Bus-Systemen gibt es jedoch eine besondere Ausgangsstufe, die man parallel schalten kann.

Der *Tri-State-Ausgang* (auch *3-State* oder *Three-State*) hat drei Möglichkeiten für den Ausgabewert. Neben 0 und 1 kann der Ausgang abgeschaltet werden; er ist dann passiv und gibt keinen Wert aus. Dies wird als hochohmig mit der Abkürzung ‚Z' bezeichnet. In Schaltsymbolen wird ein Tri-State-Ausgang durch ein auf der Spitze stehendes Dreieck dargestellt (Abb. 6.24).

Der Ausgangstreiber hat dazu einen Steuereingang *EN* (Enable), der mit 1 die Datenausgabe aktiviert. Bei *EN* auf 0 ist der Ausgang hochohmig (Abb. 6.25).

Ein typisches Einsatzgebiet von Tri-State-Leitungen sind Bus-Systeme, zum Beispiel der PCI-Bus im PC. Hier sind CPU, Grafikkarte und weitere Peripheriekarten eingesteckt. Nur einer dieser Busteilnehmer gibt Daten aus, die anderen Anschlüsse sind hochohmig. Durch die Steuerung muss sichergestellt werden, dass stets nur ein Ausgang aktiv ist.

Auch die Verbindung zwischen CPU und DRAM nutzt Tri-State-Leitungen. Wenn die CPU Daten schreibt, ist der DRAM-Anschluss hochohmig. Wenn die CPU Daten liest, ist der CPU-Anschluss hochohmig.

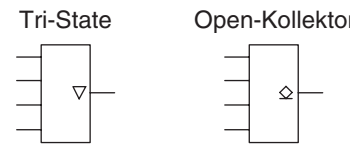
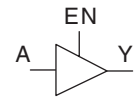


Abb. 6.24 Symbol für Tri-State- und Open-Kollektor-Ausgang



EN	A	Y
0	-	Z
1	0	0
1	1	1

Abb. 6.25 Tri-State-Treiber und Funktionstabelle

6.6.3 Open-Kollektor-Ausgang

Eine andere Methode zur Zusammenschaltung mehrerer Digitalausgänge ist der *Open-Kollektor-Ausgang*. Normalerweise hat ein Digitalausgang zwei Transistoren. Entweder zieht der eine Transistor den Ausgang zum Low-Pegel oder der andere Transistor zieht den Ausgang zum High-Pegel. Beim Open-Kollektor-Ausgang ist nur der Transistor zum Low-Pegel vorhanden. Der Kollektor dieser Schaltung bildet den Ausgang und liegt offen, daher der Name. Da statt Bipolar-Transistoren heute meist Feldeffekt-Transistoren verwendet werden, wird auch der Name *Open-Drain-Ausgang* verwendet.

Der Open-Kollektor-Ausgang wird an einen externen Lastwiderstand R_L angeschlossen, der die Ausgangsleitung nach Versorgungsspannung U_S, und damit nach High-Pegel zieht. Wenn der Ausgang aktiv ist, schaltet er den Transistor leitend und zieht die Ausgangsleitung Y nach Low-Pegel. Der Vorteil dieses Schaltungsprinzips besteht darin, dass mehrere Open-Kollektor-Ausgänge parallel geschaltet werden können und jeder den Ausgang auf Low-Pegel ziehen kann (Abb. 6.26).

Wenn ein oder mehrere Bauelemente die Ausgangsleitung auf Low-Pegel ziehen, ergibt sich eine logische 0. Nur wenn alle Bauelemente den Ausgang auf High-Pegel lassen, ergibt sich eine logische 1. Dies entspricht einer UND-Funktion. Die Zusammenschaltung wird auch als Wired-AND bezeichnet, also als „UND-Gatter durch Verdrahtung“.

In Schaltsymbolen wird ein Open-Kollektor-Ausgang durch eine Raute mit Balken dargestellt (Abb. 6.24).

Der Open-Kollektor-Ausgang wird eingesetzt, wenn mehrere Signale miteinander logisch verknüpft werden sollen. Es ist, anders als bei Tri-State-Ausgängen, keine zentrale Steuerung erforderlich. Allerdings ist auch nicht ohne weiteres ersichtlich, welcher Baustein das Signal auf 0 gezogen hat.

Als Beispiel nehmen wir an, dass mehrere Peripheriebausteine an eine CPU angeschlossen sind und einen Interrupt auslösen können. Durch ein Wired-AND können die Bausteine ihre Interrupt-Leitungen kombinieren und gemeinsam an die CPU geben, so dass nur ein einziger Interrupt-Eingang erforderlich ist. Wenn ein Interrupt auftritt, fragt die CPU ab, welcher Peripheriebaustein Auslöser des Interrupts war und bearbeitet die Anfrage.

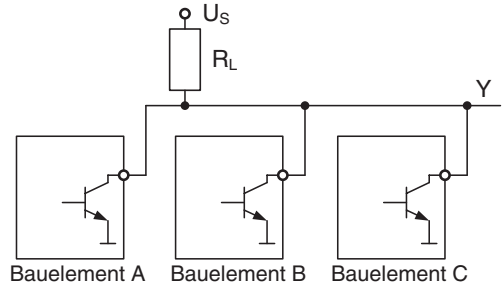


Abb. 6.26 Verdrahtung mehrerer Open-Kollektor-Ausgänge

6.7 Übungsaufgaben

Haben Sie den Inhalt des Kapitels verstanden? Prüfen Sie sich mit den Aufgaben und Fragen am Kapitelende. Die Lösungen und Antworten finden Sie am Ende des Buches.

Bei den Auswahlfragen ist immer genau eine Antwort korrekt.

Aufgabe 6.1
Wie bezeichnet man eine Digitalschaltung, bei der Steuereingänge einen von mehreren Dateneingängen auswählen?

a) Multiplexer
b) Demultiplexer
c) Addierer
d) Schieberegister
e) Datenregister

Aufgabe 6.2
Die Grundstruktur einer Additionsschaltung mit der Kaskadierung von Volladdierern nennt man, …

a) Halbaddierer
b) Carry-Overflow-Addierer
c) Carry-Pulse-Addierer
d) Carry-Overtake-Addierer
e) Ripple-Carry-Addierer

Aufgabe 6.3
Welche Aussage trifft für Tri-State-Ausgänge zu?

a) Mehrere Ausgänge werden UND-verknüpft
b) Rauschen am Eingang wird durch eine Hysterese entstört
c) Der High-Pegel kann konfiguriert werden
d) Eine Signalleitung kann als Eingang oder Ausgang geschaltet werden
e) Ein High-Pegel wechselt nach kurzer Zeit automatisch zum Low-Pegel

Aufgabe 6.4
Was wird als Spike (auch Glitch oder Hazard) bezeichnet?

a) Fehler durch Weltraumstrahlung
b) Invertierung eines Taktsignals
c) Kurze Zwischenzustände an Schaltungsausgängen

d) Höchstfrequenz eines Taktsignals
e) Verzögerungszeit bei Flip-Flops

Aufgabe 6.5
Wie bezeichnet man den langsamsten Weg durch eine kombinatorische Schaltung?

a) Periodendauer
b) Hold-Zeit
c) Zyklus
d) Setup-Zeit
e) Kritischer Pfad

Aufgabe 6.6
Wie viele Signalleitungen (Ein-/Ausgänge, keine Versorgungsspannung/Masse) hat ein 1-aus-4 Multiplexer/Datenselektor?

a) 9
b) 5
c) 7
d) 4
e) 6

Aufgabe 6.7
Wie viele Signalleitungen (Ein-/Ausgänge, keine Versorgungsspannung/Masse) hat ein 1-auf-8 Demultiplexer?

a) 8
b) 10
c) 11
d) 12
e) 9

Aufgabe 6.8
Ein Modulo-2^10 Zähler hat einen Takt von 50 MHz. Wie viele Zählzyklen schafft der Zähler pro Sekunde (gerundet)?

a) 5.000.000
b) 50.000
c) 50.000.000
d) 2.000
e) 100.000.000

Aufgabe 6.9
Ein Modulo-2^8 Zähler hat einen Takt von 500 kHz. Wie viele Zählzyklen schafft der Zähler pro Sekunde (gerundet)?

a) 100.000
b) 2.000
c) 5.000
d) 1.000
e) 500.000

Aufgabe 6.10
Die mögliche Taktfrequenz für den Addierer in Abb. 6.17 soll erhöht werden. Fügen Sie eine Pipeline-Stufe ein. Beachten Sie, dass alle Signale gleich verzögert werden, damit Informationen der Datenworte weiterhin zueinander passen.

Berechnen Sie kritischen Pfad und mögliche Taktfrequenz mit den Annahmen aus Abschn. 6.5.3.

Realisierung digitaler Systeme

7

Bei der Realisierung eines Systems müssen neben der digitalen Funktion weitere Aspekte berücksichtigt werden, die sowohl technischen als auch nicht-technischen Charakter besitzen. Einige Beispiele für diese Aspekte sind:

- Rechenleistung
- Verlustleistung
- Formfaktor, maximaler Platzbedarf
- Benötigte Logikpegel für Ein- und Ausgabe
- Entwurfskosten, Produktionskosten
- Entwicklungszeit, Time-to-Market
- Vorkenntnisse und Erfahrungen

Für die Realisierung eines digitalen Systems gibt es unterschiedliche Alternativen, die sich im Hinblick auf die genannten Eigenschaften unterscheiden. Es ist beispielsweise denkbar, ausschließlich Standard-Bausteine einzusetzen, deren Funktion vom Hersteller fest vorgegeben ist. Ebenso ist es möglich, selbst als Halbleiter-Hersteller zu agieren und eigene Chips produzieren zu lassen. Es können auch Programmierbare Logikbausteine eingesetzt werden, deren Hardware-Funktion flexibel festgelegt werden kann. Statt eine Funktion in Form von Gattern zu realisieren, ist auch ein softwareorientierter Ansatz möglich, bei dem beispielsweise Mikrocontroller eingesetzt werden. Diese Bausteine sind deutlich kompakter als ein PC und sind teilweise für weniger als 1 EUR erhältlich. In diesem Kapitel werden die verschiedenen Varianten näher beleuchtet.

W. Gehrke et al., *Digitaltechnik,* Springer-Lehrbuch,
DOI 10.1007/978-3-662-49731-9_7

7.1 Standardisierte Logikbausteine

Unter *Standardlogik-Bausteinen* werden Komponenten verstanden, die käuflich zu erwerben sind und eine einfache digitale Hardware-Funktion zur Verfügung stellen, welche durch den Anwender nicht modifiziert werden kann.

Standardlogik-Bausteine sind in Bausteinfamilien bzw. -serien zusammengefasst. Die wichtigsten Familien sind die sogenannte 4000er-Serie sowie die 7400er-Serie (bzw. kurz 74er-Serie). Die Bezeichnung dieser Familien geht auf die Kennzeichnung der zugehörigen Schaltkreise zurück. So beginnt die Bezeichnung eines Bausteins der 74er-Serie immer mit der Zahl 74. Diese wird meist von mehreren Buchstaben gefolgt, die die Implementierungstechnologie und damit auch einige Eigenschaften (zum Beispiel Logikpegel) des Bausteins beschreiben. Eine abschließende Zahl kennzeichnet die logische Funktion. So besitzen ein 74HC374 und ein 74LVC374 zwar die gleiche logische Funktion (acht D-Flip-Flops), aber ein unterschiedliches Zeitverhalten und unterschiedliche elektrische Eigenschaften.

Als ein Vertreter der 74er-Familie ist in Abb. 7.1 der Baustein 74HC04 abgebildet, welcher sechs Inverter enthält. Die Buchstaben *SN* stehen für den Hersteller und das *N* am Ende der Bausteinkennzeichnung gibt die Gehäuseform an.

In den 1970er Jahren wurden die Standardlogik-Bausteine noch zur Realisierung von Computern eingesetzt. Die hiermit verbundenen Nachteile liegen auf der Hand: Große Bauform, hohe Kosten, große Verlustleistung. Die damaligen Computer waren so groß wie Kleiderschränke, hatten eine Stromaufnahme, die mit mehreren hundert heutiger PCs vergleichbar ist und boten für 6-stellige Dollar-Beträge eine Rechenleistung, für die heute vermutlich niemand auch nur einen Euro bezahlen würde.

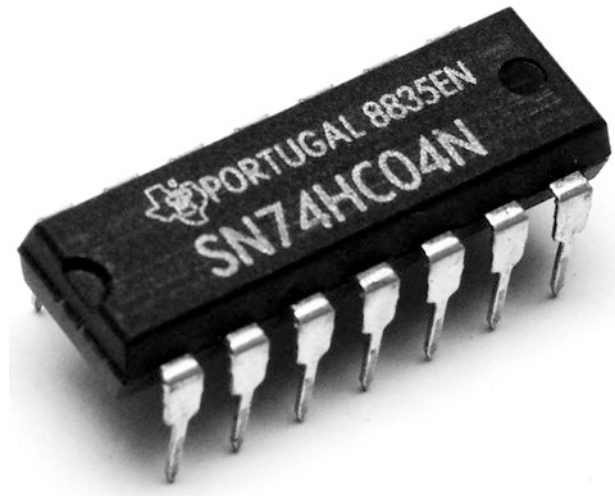


Abb. 7.1 Baustein 74HC04: Sechs Inverter in einem gemeinsamen Gehäuse

Obwohl die Standardlogik-Komponenten heute keine Bedeutung für die Realisierung ganzer Systeme mehr haben, haben sie dennoch ihre Daseinsberechtigung. Sie werden zum Beispiel dann eingesetzt, wenn einfache logische Funktionen mithilfe von ein paar wenigen Gattern realisiert werden sollen. Ebenso können einige dieser Bausteine auch als Leitungstreiber oder zur Pegelanpassung zwischen Komponenten mit unterschiedlichen Versorgungsspannungen eingesetzt werden.

Zur Verdeutlichung, welche logischen Funktionen in der 74er-Serie zur Verfügung stehen, sind einige ausgewählte Funktionen in Tab. 7.1 zusammengestellt. Eine umfassende Dokumentation der verfügbaren digitalen Funktionen kann von den Herstellern (Texas Instruments, NXP, STM, u.v.a.) bezogen werden.

Die ersten Standard-Logikbausteine der 74er-Serie wurden mithilfe von Bipolar-Transistoren realisiert. Inzwischen hat auch in diesem Bereich die CMOS-Technologie (vgl. Kapitel 10) die reine bipolare Implementierung verdrängt. Einige Familien werden auch mit einer Kombination von bipolaren und MOS-Transitoren realisiert. Die Eingänge sowie die logische Funktion werden dann mithilfe der CMOS-Technik

Tab. 7.1 Ausgewählte Logikfunktionen der 74er-Serie

Baustein *(letzte Ziffern)*	Funktion
00	4 NAND2
02	4 NOR2
04	6 Inverter
07	6 Treiber/Buffer (mit OC-Ausgang)
08	4 AND2
10	3 NAND3
25	2 NOR4
46	BCD nach Siebensegment Decoder
74	2 D-Flip-Flops mit Set- und Reset-Eingängen
138	3:8 Demultiplexer/Decoder
148	8:3 Prioritätsencoder
165	8 Bit Parallel-In/Serial-Out Schieberegister
190	4 Bit Aufwärts-/Abwärtszähler
244	8 Bit Leitungstreiber mit Tristate-Ausgängen
245	8 Bit Bidirektionaler Bustreiber mit Tristate-Ausgängen
373	8 pegelgesteuerte D-Flip-Flops mit Tristate-Ausgängen
374	8 flankengesteuerte D-Flip-Flops mit Tristate-Ausgängen
573	8 pegelgesteuerte D-Flip-Flops mit Tristate-Ausgängen
574	8 flankengesteuerte D-Flip-Flops mit Tristate-Ausgängen
595	8 Bit Serial-in/Parallel-out Schieberegister mit Tristate-Ausgängen

implementiert, während für die Ausgangstreiber Bipolar-Transistoren eingesetzt werden. So wird gegenüber einer reinen CMOS-Implementierung eine höhere Treiberleistung und eine geringere Abhängigkeit von der Lastkapazität erreicht. Eine Übersicht über verschiedene Familien der 74er-Serie folgt in Abschn. 7.1.5.

Nicht alle Grundfunktionen der 74er-Serie werden in allen Familien angeboten. Im Einzelfall muss geprüft werden, ob eine gewünschte Funktion zur Verfügung steht.

Als eine Ergänzung zu der weitverbreiteten 74er-Serie bietet beispielsweise die Firma NXP konfigurierbare Logikgatter in platzsparenden Gehäusen an. Damit kann ein einzelnes NAND- oder NOR-Gatter mit zwei Eingängen realisiert werden, während ein typischer Baustein der 74er-Serie vier dieser Gatter enthält. Die konfigurierbaren Logikgatter sind in den Familien LVC, AUP (Advanced Ultra-Low-Power) und AXP (Advanced Extremely Low-Power) verfügbar. Die Logikfunktion der Gatter ist durch die äußere Beschaltung wählbar.

7.1.1 Charakteristische Eigenschaften digitaler Schaltkreise

Bevor ein Baustein für den Entwurf eines digitalen Systems ausgewählt wird, müssen dessen Merkmale bekannt sein. In den Datenblättern integrierter Schaltungen wird meist eine Reihe von Kenndaten angegeben, die die Eigenschaften des Bausteins beschreiben. Neben dem erlaubten Versorgungsspannungsbereich sind unter anderem die Pegel sowie die zulässigen Ströme an Ein- und Ausgängen von Bedeutung (vgl. Abb. 7.2).

Für diese Parameter definieren die Datenblätter die zulässigen Wertbereiche. Einige der wichtigsten Parameter sind in Tab. 7.2 zusammengefasst. Die Formelzeichen entsprechen denen, die in englischsprachigen Datenblättern verwendet werden. Daher wird hier der Buchstabe *V* als Formelzeichen für die elektrische Spannung verwendet.

7.1.2 Lastfaktoren

Die Treiberstärke einer Ausgangsleitung muss für die angeschlossene Belastung durch die nachfolgenden Bausteine ausreichen. Die Belastung, die ein Ausgang durch einen

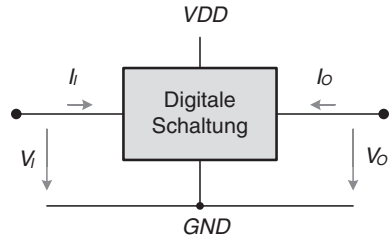


Abb. 7.2 Anschlussbezeichnung digitaler Schaltungen

Tab. 7.2 Wichtige Parameter zur Charakterisierung digitaler Schaltkreise

Formelzeichen	Bedeutung	Bemerkungen
GND	Masse	Alternative Bezeichnung: V_{SS}
VDD	Versorgungsspannung	Alternative Bezeichnung: V_{CC}
V_I	Eingangsspannung	
I_I	Eingangsstrom	
V_O	Ausgangsspannung	
I_O	Ausgangsstrom	
$V_{I,H\min}$	Minimale Eingangsspannung, die als High-Pegel erkannt wird	Abhängig von Versorgungsspannung
$V_{I,L\max}$	Maximale Eingangsspannung, die als Low-Pegel erkannt wird	
$I_{I,H}$	Eingangsstrom bei High-Pegel	Bei CMOS-Schaltkreisen meist vernachlässigbar
$I_{I,L}$	Eingangsstrom bei Low-Pegel	
$V_{O,H\min}$	Garantierte minimale Ausgangsspannung bei High-Pegel	Abhängig von Versorgungsspannung und Ausgangsstrom
$V_{O,L\max}$	Garantierte maximale Ausgangsspannung bei Low-Pegel	
$I_{O,H\max}$, $I_{O,L\max}$	Maximal zulässiger Ausgangsstrom bei High- bzw. Low-Pegel	

Eingang innerhalb der gleichen Schaltkreisfamilie erfährt, wird durch den sogenannten Lastfaktor beschrieben. Hierzu wird der Eingangsstrom eines typischen Gatters der Bausteinfamilie *(Einheitsgatter)* definiert. Es ergeben sich für Low- und High-Pegel die beiden charakteristischen Größen $I_{I,HN}$ und $I_{I,LN}$, die den Strom angeben, welcher in den Eingang des Einheitsgatters hineinfließt.

Auf Basis der Eigenschaften eines Einheitsgatters lassen sich die beiden charakteristischen Größen Fan-in und Fan-out definieren.

Fan-in (Eingangslastfaktor)
Der Fan-in eines Eingangs gibt an, um welchen Faktor die Stromaufnahme größer ist als beim Einheitsgatter derselben Schaltkreisfamilie.

$$F_{I,H} = \frac{I_{I,H}}{I_{I,HN}} F_{I,L} = \frac{I_{I,L}}{I_{I,LN}} F_I = \max\left[F_{I,H}, F_{I,L}\right]$$

Innerhalb einer Schaltkreisfamilie gilt ein Eingang als einfache Last, wenn er den gleichen Strom aufnimmt wie das Einheitsgatter ($F_I = 1$).

Fan-out (Ausgangslastfaktor)
Der Fan-out gibt an, mit wie vielen Eingängen eines Einheitsgatters derselben Schaltkreisfamilie der entsprechende Ausgang belastet werden darf.

$$F_{O,H} = \frac{I_{O,H\max}}{I_{I,HN}} F_{O,L} = \frac{I_{O,L\max}}{O_{I,LN}} F_O = \min\left[F_{O,H}, F_{O,L}\right]$$

7.1.3 Störspannungsabstand

Als *Störspannungsabstand* bezeichnet man die Spannung, um die ein Digitalausgang variieren darf, ohne dass ein angeschlossener Eingang derselben Logikfamilie in einen verbotenen Pegelbereich gelangt. Der Störspannungsabstand wird für High- und Low-Pegel getrennt angegeben (Abb. 7.3).

$$S_H = U_{O,H\min} - U_{I,H\min}$$

$$S_L = U_{O,L\max} - U_{I,L\max}$$

7.1.4 Schaltzeiten

Beim Einsatz eines digitalen Bausteins ist unter anderem die Verzögerungszeit, die teilweise auch als *Schaltzeit* bezeichnet wird, von großer Bedeutung. Um die Verzögerungszeiten zu bestimmen, wird üblicherweise eine Rechteckspannung an den Eingang des Bausteins angelegt und der zeitliche Verlauf der Ausgangspannung gemessen. Das Ausgangssignal ist nicht rechteckförmig und der Wechsel des logischen Signals am Ausgang nimmt eine gewisse Zeit in Anspruch. Die Zeit setzt sich zusammen aus einer Verzögerung im Inneren des Logikbausteins sowie der Zeit für die Umladung der Last am Ausgang.

Wird die Zeit gemessen, die der Ausgang benötigt, um von 10 % auf 90 % des Ausgangspegels anzusteigen bzw. von 90 % auf 10 % abzufallen, erhält man die Anstiegszeit *(rise time, t_R)* bzw. Abfallzeit *(fall time, t_F)*. Häufig werden diese Zeiten auch zusammenfassend als *transition time (t_T)* angegeben.

Möchte man die Verzögerungszeit eines Bausteins angeben, so wird hierfür als Referenzpunkt genau die Mitte zwischen Minimal- und Maximalpegel gewählt. Die Zeit, die zwischen dem Erreichen des 50 %-Eingangspegels vergeht, bis der Ausgang seinerseits 50 %

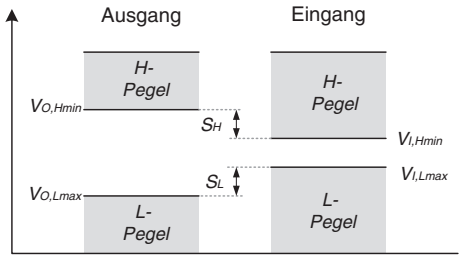


Abb. 7.3 Störspannungsabstand

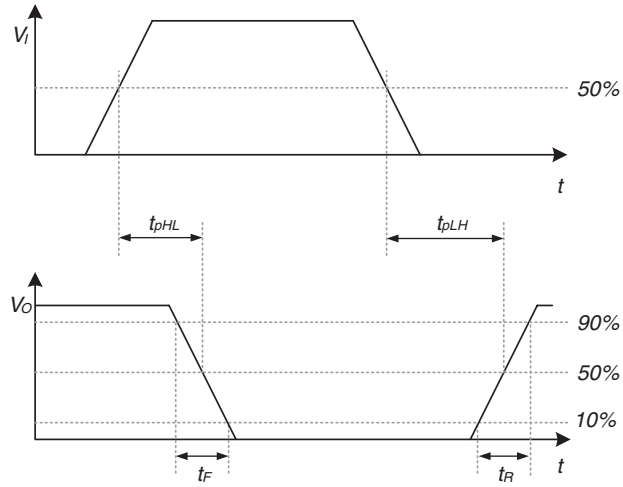


Abb. 7.4 Verzögerungszeiten einer digitalen Schaltung am Beispiel eines Inverters

des Pegels erreicht hat, ergibt also die Verzögerungszeit *(propagation delay, t_P)*. Diese kann auch für steigende und fallende Flanken getrennt angegeben werden kann *(t_{PLH}, t_{PHL})*.

In Abb. 7.4 sind die Schaltzeiten für das Beispiel eines Inverters dargestellt.

7.1.5 Logikfamilien

In Tab. 7.3 sind einige ausgewählte Familien der 74er-Serie mit Versorgungsspannungsbereich und Schaltzeiten eines 74*xx*00 (vier NAND2-Gatter) zusammengefasst. Die

Tab. 7.3 Übersicht über einige Familien der 74er-Serie: Versorgungsspannungsbereich und typische Schaltzeiten für einen 74*xx*00-Baustein

Abkürzung	Bezeichnung	V_{CC} (V)	t_T (ns)	t_P (ns)	Bemerkungen
(Keine)	Standard TTL *(veraltet)*	4,5 ~ 5,5	7	9	$V_{CC} = 5{,}0V$; $C_L = 15pF$
LS	Low-Power Schottky *(veraltet)*	4,5 ~ 5,5	7	10	$V_{CC} = 5{,}0V$; $C_L = 15pF$
HC	High-Speed CMOS	2,0 ~ 6,0	6	7	$V_{CC} = 5{,}0V$; $C_L = 15pF$
HCT	HC, TTL-compatible	4,5 ~ 5,5	7	8	$V_{CC} = 5{,}0V$; $C_L = 15pF$
AHC	Advanced High-Speed CMOS	2,0 ~ 5,5	3	4,5	$V_{CC} = 5{,}0V$; $C_L = 15pF$
LVC	Low Voltage CMOS	1,65 ~ 3,6	2	3,0	$V_{CC} = 3{,}3V$; $C_L = 50pF$
ALVC	Adv. Low Voltage CMOS	1,65 ~ 3,6	2	2,1	$V_{CC} = 3{,}3V$; $C_L = 50pF$
ABT	Adv. BiCMOS, TTL-compatible	4,5 ~ 5,5	2,5	2,3	$V_{CC} = 3{,}3V$; $C_L = 50pF$
AUC	Adv. Ultra Low Voltage CMOS	0,8 ~ 2,7	1	1,5	$V_{CC} = 1{,}8V$; $C_L = 30pF$

Schaltzeiten gelten für die angegebenen Randbedingungen, insbesondere Versorgungsspannung und Lastkapazität *(C_L).* Darüber hinaus können die Schaltzeiten auch auf Grund von Streuungen bei der Fertigung der Bausteine variieren. In den meisten Datenblättern wird daher neben den typischen Zeiten auch ein Maximalwert angegeben.

7.2 Komponenten für digitale Systeme

Für die Implementierung einer digitalen Schaltung kommen verschiedene Strategien in Betracht, die in diesem Abschnitt vorgestellt werden. Reale digitale Systeme verwenden häufig eine Kombination dieser Strategien.

7.2.1 ASICs

Möchte man ein digitales System realisieren, kann man einen speziellen Halbleiterbaustein fertigen lassen, der die gewünschte Funktion ausführt. In diesem Fall spricht man von sogenannten *ASICs (Application Specific Integrated Circuit).* Beim Entwurf eines ASICs wird auch ein digitales System aus logischen Grundelementen erstellt. Statt jedoch die Grundfunktionen auf einer Platine (wie zum Beispiel bei Verwendung von Bausteinen der 74er-Serie) vorzunehmen, erfolgt die Platzierung und Verdrahtung der Gatter beim ASIC-Entwurf auf einer wenige Quadratmillimeter großen Siliziumfläche. Diese Realisierung ist viel kompakter als bei Verwendung standardisierter Logikbausteine. Darum ist ein ASIC häufig schneller und besitzt eine geringere Verlustleistung. Da die Anzahl und die Position der Gatter während des Entwurfs frei gewählt werden können, kann der Baustein für den jeweiligen Anwendungsfall optimiert werden.

Für den Entwurf eines ASICs wird der sogenannte Standardzellentwurf eingesetzt. Bei dieser Entwurfsmethodik stehen die logischen Grundelemente als Bibliothek in elektronischer Form zur Verfügung. Aus dieser Bibliothek können Bauelemente ausgewählt, auf dem Chip platziert und anschließend verdrahtet werden.

Die Auswahl und das Verbinden der einzelnen Gatter zu einem komplexen System erfolgt mithilfe einer Hardwarebeschreibungssprache wie VHDL. Mit einem Syntheseprogramm wird die VHDL-Beschreibung in eine sogenannte *Gatternetzliste* überführt. Diese Netzliste gibt an, welche Logikelemente verwendet werden und wie diese verdrahtet sind. Die Synthese hat also die Aufgabe die VHDL-Beschreibung zu analysieren und eine möglichst optimale Implementierung auf Basis der Grundelemente der Bibliothek zu finden. Optimal heißt in diesem Fall, dass die spezifizierten maximalen Verzögerungszeiten eingehalten werden und eine möglichst kleine Chipfläche benötigt wird. Darüber hinaus können auch Aspekte wie die Verlustleistung Berücksichtigung finden. Dieser Entwurfsschritt wird häufig auch als *Frontend-Design* bezeichnet.

Nachdem das Frontend-Design abgeschlossen ist, erfolgt das *Backend-Design.* In diesem Schritt werden mit speziellen Layoutprogrammen die Platzierung und die

Verdrahtung der Elemente aus der Gatternetzliste vorgenommen. Hierfür ist in der Bibliothek für jedes Element der Netzliste eine Implementierung aus einzelnen Transistoren hinterlegt.

Auf den ersten Blick klingt der Ansatz des ASIC-Entwurfs vielleicht als ideale Lösung zur Realisierung digitaler Systeme. Aufgrund der Optimierung können die Schaltkreise mit einer relativ kleinen Siliziumfläche und damit kostengünstig produziert werden. Allerdings sind die Produktionskosten nicht der einzige Kostenfaktor eines ASIC-Entwurfs, denn es fallen in einem deutlichen Umfang einmalige Kosten (engl. *non-recurring engineering costs* bzw. *NRE*) an. Diese Kosten entstehen zum einen durch den hohen Arbeitsaufwand im Frontend- und Backend-Design. Zum anderen ist die Erstellung von Belichtungsmasken, die zur Produktion des Schaltkreises in der Halbleiterfabrik benötigt werden, ein weiterer wichtiger Kostenfaktor. Aufgrund der kleinen Strukturen heutiger Produktionsprozesse werden extrem präzise Masken benötigt, sodass die Vorbereitung der Produktion eines ASICs mehrere Millionen Euro kosten kann. Berücksichtigt man diese Kosten, wird deutlich, dass vor der Produktion eines ASICs eine intensive Überprüfung des Designs erforderlich ist, damit die Wahrscheinlichkeit eines Designfehlers verringert wird.

Nehmen wir als Beispiel an, dass die NRE-Kosten eines ASIC-Projekts etwa 15 Mio EUR betragen. Wenn der Baustein in einer Stückzahl von 100.000 produziert werden soll, ergibt sich umgerechnet auf einen einzelnen Baustein ein Anteil von 150 EUR. Diese Kosten sind für viele Anwendungsgebiete unattraktiv, sodass nur bei sehr hohen Stückzahlen eine ASIC-Entwicklung wirtschaftlich sinnvoll ist.

7.2.2 ASSPs

Eine Alternative zur Entwicklung eines eigenen Bausteins können sogenannte *Application Specific Standard Products (ASSPs)* sein. Ein ASSP hat den gleichen Aufbau wie ein ASIC, wird allerdings nicht selbst entworfen, sondern ist ein frei am Markt erhältlicher Schaltkreis. Er kann für eine sehr spezielle Funktion (zum Beispiel WLAN, Steuerung von Motoren) optimiert sein oder aber auch als *System-on-Chip (SoC)* mehrere Funktionen integrieren und so die kostengünstige Implementierung eines Gesamtsystems ermöglichen. Ein Beispiel für ein System-on-Chip sind die ASSPs, die in heutigen Fernsehern verbaut werden: Fast die gesamte Funktionalität vom Empfang des Fernsehsignals über Satellit, Kabel oder WLAN bis hin zur Anzeige auf einem Display ist in einem hochintegrierten Baustein vereinigt.

7.2.3 FPGAs und CPLDs

Die Produktion eines ASICs ist ein sehr attraktiver Weg zur Realisierung eines digitalen Systems – wenn sie nicht mit erheblichen Grundkosten verbunden wäre. Wäre es also

vielleicht ein möglicher Ausweg, wenn man Bausteine hätte, deren Hardware zwar fest ist, aber deren digitale Funktion erst vom Anwender festgelegt würde? Diese Bausteine kann man (aufgrund der festen Hardware) in großen Stückzahlen günstig herstellen und dennoch kann der Anwender die digitale Funktion, wie bei einem ASIC, nach seinen Bedürfnissen festlegen.

Diese Überlegungen wurden bereits sehr früh angestellt und die Idee, Schaltkreise zu realisieren, deren logische Funktion in VHDL „programmiert“ werden kann, wurde schon in den 1970er Jahren aufgegriffen und ist bis heute immer weiter verfeinert worden.

Die Besonderheit dieser Bausteine ist, dass ihre logische Funktion noch im Feld (zum Beispiel nach dem Einsetzen in eine Platine) konfiguriert werden kann. Daher werden sie als *Field Programmable Gate Arrays (FPGAs)* bezeichnet. Neben FPGAs werden auch *Complex Programmable Logic Devices (CPLDs)* beziehungsweise *Simple Programmable Logic Devices (SPLDs)* angeboten. CPLDs eignen sich besonders für programmierbare logische Funktionen mit einer relativ geringen Komplexität, während mit FPGAs ganze Rechnersysteme realisiert werden können. Die gesamte Gruppe dieser Bausteine wird auch unter dem Begriff *Programmierbare Logik* zusammengefasst.

Sind also FPGAs die ideale Lösung zur Realisierung einer digitalen Funktion? In vielen Fällen kann man diese Frage tatsächlich bejahen: Mit heutigen FPGAs können sehr komplexe Systeme zu einem relativ günstigen Preis realisiert werden. Insbesondere bei kleinen bis mittleren Stückzahlen können FPGAs ihre Kostenvorteile gegenüber ASICs ausspielen. Daher werden programmierbare Logikbausteine in vielen Bereichen eingesetzt.

7.2.4 Mikrocontroller

Kann man eine digitale Funktion statt mit Gattern auf einer Platine oder in Form eines ASICs auf einem Stück Silizium vielleicht auch in Software realisieren? Schließlich ist doch das Grundprinzip eines jeden Rechnerprogramms das Einlesen von Eingabewerten, die Verarbeitung der Werte und die anschließende Ausgabe von Ergebnissen. Und letztlich macht ein logisches Gatter oder auch ein komplexes System nichts anderes: Es betrachtet sozusagen die Eingänge und bestimmt nach einer festgelegten Rechenvorschrift die Ausgangssignale. Also müsste es möglich sein, eine beliebige digitale Funktion auch mithilfe eines Rechners zu realisieren.

Sie mögen vielleicht einwenden, dass es wenig sinnvoll ist, wenn man beispielsweise die Funktion eines einfachen UND-Gatters durch ein Programm auf einem PC ersetzt. Sicher, die Kosten der PC-basierten Lösung wären viel zu hoch und auch die Bauform und die benötigte leistungsfähige Spannungsversorgung wären nachteilig. Ein Rechnersystem auf Basis eines PCs ist also aus verschiedensten Gründen für viele Anwendungsgebiete nicht gut geeignet.

Aber es existieren Alternativen zu einem Standard-PC: Bereits in den 1970er Jahren erkannten die Halbleiterhersteller den Bedarf an kostengünstigen, stromsparenden Rechnersystemen, die sich auf einem Stück Silizium unterbringen ließen. Diese Bausteine sind nicht als PC-Ersatz gedacht, sondern werden häufig dort eingesetzt, wo sich Steuerungs- und Regelungsaufgaben elegant in Software realisieren lassen und nur moderate Rechenleistungen benötigt werden. Aufgrund dieses Anwendungsbereiches bürgerte sich schnell die Bezeichnung Mikrocontroller für diese Art von Bausteinen ein.

Mikrocontroller enthalten in einem einzelnen Gehäuse alles, was einen Rechner ausmacht: Einen Mikroprozessor zur Abarbeitung eines Programms, Speicher für Programme und Daten und Ein-/Ausgabe-Schnittstellen für die Kommunikation mit der Außenwelt.

Obwohl das Grundkonzept eines PCs und eines Mikrocontrollers ähnlich ist, unterscheiden sie sich doch erheblich: Während PCs für interaktives Arbeiten ausgelegt sind und vorrangig eine hohe Rechenleistung bieten sollen, stehen bei Mikrocontrollern vor allem der Preis und eine kompakte Bauform im Vordergrund. Mikrocontroller besitzen daher eine (im Vergleich zu einem aktuellen PC) geringe Rechenleistung und einen deutlich kleineren Speicher. Trotz dieser Einschränkungen werden jedes Jahr mehrere Milliarden Mikrocontroller verbaut (Abb. 7.5).

Wenn Sie einen Gang durch Ihren Haushalt machen, werden Sie vermutlich viele Geräte entdecken, die einen Mikrocontroller enthalten. Betrachten wir als ein Beispiel eine Waschmaschine: Die Aufgaben an die Steuerung sind vielfältig. Es wird eine Benutzerschnittstelle in Form von Tastern, Drehschaltern und Displays benötigt. Die Drehrichtung und Geschwindigkeit des Trommelmotors müssen geregelt werden. Und nicht zuletzt müssen Wasserzu- und -ablauf sowie die Heizung korrekt angesteuert werden. Besitzt man einen Rechnerbaustein mit digitalen Ein- und Ausgängen kann die Steuerung auf elegante Weise in Software implementiert werden. Die Rechenleistung heutiger Mikrocontroller reicht für die Regelungsalgorithmen einer typischen Waschmaschine völlig aus.

Das Einsatzgebiet der Mikrocontroller ist natürlich nicht auf den Haushalt beschränkt. Überall wo Steuerungen und Regelungen benötigt werden, werden Mikrocontroller

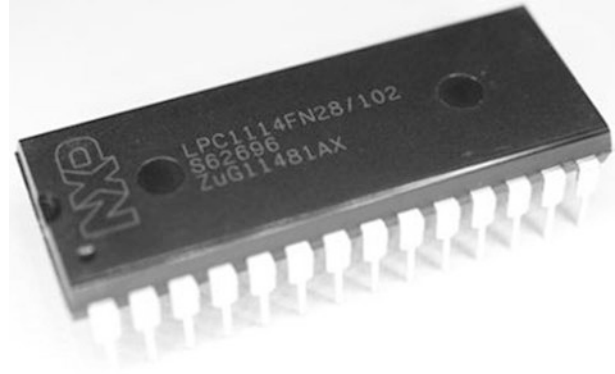


Abb. 7.5 Beispiel eines Mikrocontrollers: Von außen ist nicht zu erkennen, dass es sich um einen kompletten Rechner handelt

eingesetzt. Häufig sind diese Rechnersysteme nicht sofort erkennbar, weshalb sie auch als eingebettete Systeme *(Embedded System)* bezeichnet werden.

7.2.5 Vergleich der Alternativen

Die möglichen Alternativen für die Implementierung einer digitalen Schaltung unterscheiden sich in Flexibilität, Entwicklungszeit, Entwicklungskosten und Stückkosten. Tab. 7.4 gibt einen groben Vergleich der Alternativen ASIC, ASSP, Mikrocontroller (µC) und FPGA. Die Symbole zur Bewertung bedeuten sehr gut (+ +), gut (+), mittel (○), schlecht (–), sehr schlecht (– –).

Die Wahl einer Alternative ist abhängig von den Randbedingungen des Entwicklungsprojektes, also unter anderem Komplexität der Schaltung, Zeitdruck, Kostendruck, Konkurrenzsituation. Die Entscheidung für ein Implementierungskonzept ist daher in der Praxis das Ergebnis einer ausführlichen Analyse und wird zwischen Entwicklungsteam, Produktmarketing und Unternehmensleitung abgestimmt.

7.2.6 Kombination von Komponenten

In komplexeren digitalen Systemen wird die Systemfunktion häufig auf verschiedene Bausteine verteilt. Die zentrale Komponente ist dann häufig ein programmierbarer Baustein, der einen Mikroprozessor enthält und mit Programmiersprachen wie C/C++ programmiert werden kann. Der Mikroprozessor kann durch programmierbare Logikbausteine, wie FPGAs oder CPLDs ergänzt werden. Auf diese Weise können einige Systemfunktionen in der programmierbaren Logik implementiert werden, wodurch der zentrale Mikroprozessor entlastet wird.

Wenn das System einen Speicherbedarf von einigen Megabyte oder mehr besitzt, werden zusätzlich spezielle Speicherbausteine benötigt, die als eigenständige Komponenten auf der Systemplatine untergebracht werden.

Tab. 7.4 Alternativen zur Implementierung digitaler Schaltungen

	ASIC	ASSP	µC	FPGA
Hohe Flexibilität	+	-	+	+ +
Geringe Entwicklungszeit	– –	+	+ +	○
Geringe Entwicklungskosten	– –	+	+ +	○
Geringe Stückkosten	+ +	+	+ +	○
Rechenleistung	+ +	+ +	○	+
Verlustleistung	+ +	+ +	○	○
Geringe Stückzahlen möglich	– –	+ +	+ +	+ +
Hohe Stückzahlen möglich	+ +	+ +	+ +	+

Ein-/Ausgabe-Komponenten, die nicht bereits durch den Mikroprozessor zur Verfügung gestellt werden, können entweder in der programmierbaren Logik oder als zusätzliche Systemkomponenten, zum Beispiel in Form eines ASSPs, integriert werden. Insbesondere Spezialfunktionen wie WLAN, USB oder Ethernet können durch derartige zusätzliche Bausteine realisiert werden.

Für einfache Anwendungen ist eine Systemrealisierung auf Basis mehrerer Einzelkomponenten häufig nicht sinnvoll, da sie zu kostenintensiv sind oder die Verlustleistung zu groß wäre. Für diese Anwendungsfälle bietet die Halbleiterindustrie die in Abschn. 7.2.4 vorgestellten Mikrocontroller an, die sich insbesondere für eine kostengünstige Realisierung von Systemen mit relativ geringen Anforderungen an die Rechenleistung realisieren lassen.

Die unterschiedlichen Komponenten digitaler Systeme werden in verschiedenen Kapiteln genauer vorgestellt: Kapitel 9 vertieft Aspekte der programmierbaren Logikbausteine. Kapitel 10 beschreibt die Grundlagen der Halbleitertechnik. In Kapitel 11 werden Speicherbausteine vorgestellt. Die Kapitel 12 vorgestellten Analog-Digital- und Digital-Analog-Umsetzer werden immer dann benötigt, wenn die Ein-/Ausgabe in analoger Form erfolgen soll. Kapitel 13 und Kapitel 14 gehen auf die Realisierung softwareprogrammierbarer Bausteine ein, wobei der Schwerpunkt auf Mikrocontrollern liegt.

7.3 VHDL-basierter Systementwurf

Für den Entwurf digitaler Systeme wird Software eingesetzt, die den Entwicklungsprozess auf dem Weg von der Idee zum fertigen System unterstützt. Der rechnergestützte Schaltungsentwurf wird als *Electronic Design Automation (EDA)* und die Programme für die Schaltungsentwicklung als *EDA-Programme* oder *EDA-Tools* bezeichnet. Mithilfe dieser Programme kann VHDL-Code eingegeben, simuliert und in Hardware überführt werden. Das Ergebnis des Entwurfsprozesses ist eine binäre Datei, die mithilfe eines Programmiergerätes auf ein FPGA übertragen bzw. zur Fertigung eines ASICs an die Halbleiterfabrik übergeben wird.

Im Folgenden wird der VHDL-basierte Systementwurf näher beschrieben. Aufgrund der großen Bedeutung von programmierbaren Logikbausteinen, erfolgt die Beschreibung für ein FPGA-Design.

7.3.1 Designflow

Der Entwurf eines Systems auf Basis eines FPGAs beinhaltet immer zwei Aspekte: Zum einen muss die gewünschte Funktion in VHDL beschrieben und mithilfe der Entwurfssoftware in eine Programmierdatei für das FPGA übersetzt werden. Daneben ist es von wesentlicher Bedeutung, dass die einzelnen Entwurfsschritte durch Verifikation begleitet

werden. Besondere Bedeutung kommt hierbei der frühzeitigen Simulation des eingegebenen VHDL-Codes zu.

Eine schematische Übersicht über den FPGA-Entwurf zeigt Abb. 7.6. Die einzelnen Schritte werden in den folgenden Abschnitten näher erläutert.

Der Ablauf eines VHDL-basierten Entwurfs besitzt teilweise Ähnlichkeiten zur Entwicklung von Software. Die gewünschte Funktion wird in Form einer Textdatei beschrieben. Diese Datei wird dann durch einen Compiler bzw. ein Synthesetool optimiert und in ein ausführbares Programm bzw. eine Programmierdatei für das FPGA übersetzt. Es ist jedoch zu beachten, dass ein FPGA ein paralleles System ist, auf dem eine Vielzahl von Funktionen gleichzeitig ablaufen. Außerdem ist das Zeitverhalten von wesentlicher Bedeutung. Ist die Verzögerungszeit der Kombinatorik zwischen zwei Flip-Flops zu groß, wird das System fehlerhaft arbeiten. Daher ist der VHDL-basierte Entwurfsablauf, trotz der Ähnlichkeiten zur Softwareentwicklung, als Hardwareentwurf anzusehen.

7.3.2 VHDL-Eingabe

Die Hardwarebeschreibungssprache VHDL wurde in vorangegangenen Kapiteln bereits vorgestellt. Sie kennen bereits die Syntax der Sprache und wissen auch, wie Sie beispielsweise endliche Automaten in VHDL beschreiben können. Für die Entwicklung eines FPGA-Designs muss berücksichtigt werden, dass der VHDL-Code in der Regel ein synchrones System beschreibt, das aus Flip-Flops und kombinatorischer Logik besteht.

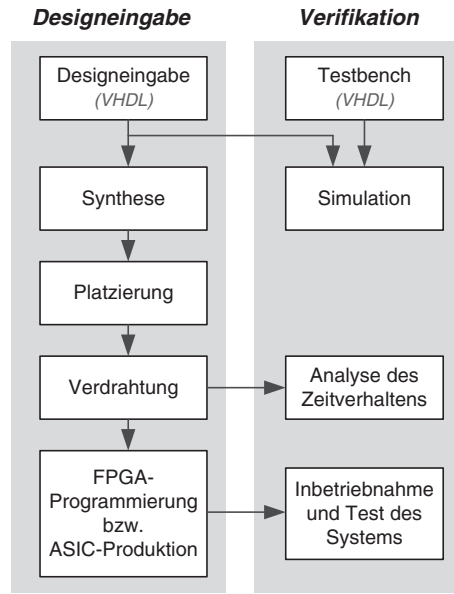


Abb. 7.6 FPGA-Designflow mit VHDL

In Kapitel 6 wurde bereits erläutert, dass die meisten digitalen Schaltungen eine Kombination von Registern und Kombinatorik zwischen den Registerstufen darstellen *(Register-Transfer-Level-Design* oder kurz *RTL-Design).* Die Grundstruktur der entsprechenden Hardware ist in Abb. 7.7 dargestellt.

Mit der Eingabe des VHDL-Codes werden die Registerstufen und die logische Funktion zwischen zwei Registerstufen festgelegt. Dabei muss auch das Zeitverhalten der späteren Hardware berücksichtigt werden. Für einfache Designs kann dies häufig als unkritisch angesehen werden. Für Entwürfe mit hohen Anforderungen an die Rechenleistung (und damit häufig einer hohen Taktfrequenz) nimmt die Bedeutung des Zeitverhaltens zu. Den größten Einfluss auf das Zeitverhalten hat der VHDL-Code. Alle nachfolgenden Schritte im Designflow können eventuelle Probleme im Zeitverhalten der Schaltung nur in einem begrenzten Umfang korrigieren.

7.3.3 Simulation

Die Simulation des VHDL-Codes ist einer der wichtigsten Schritte, um die Korrektheit der beschriebenen digitalen Funktion frühzeitig sicherzustellen. Prinzipiell bieten VHDL Simulatoren die Möglichkeit, durch Kommandos Signale auf definierte Werte zu setzen. Die verwendeten Kommandos sind nicht standardisiert und variieren mit den eingesetzten Simulatoren. Beispielsweise wird bei Verwendung des Simulators XSIM der Firma Xilinx ein Signal mit dem Namen *my_sig* mit dem Kommando *add_force my_sig 1* auf den Wert 1 gesetzt werden. Um die Reaktion der VHDL-Beschreibung sichtbar zu machen, muss anschließend mithilfe des Run-Kommandos (zum Beispiel *run 10 ns*) etwas Simulationszeit vergehen. Der zeitliche Verlauf sowohl von Eingangs- und Ausgangssignalen als auch von internen Signalen einer VHDL-Beschreibung wird während der Simulation mithilfe sogenannter Waveform-Viewer grafisch dargestellt (vgl. Kapitel 3).

Das Anlegen unterschiedlicher Eingangswerte durch Simulator-Kommandos und die Überprüfung der Schaltungsreaktion anhand der grafischen Ausgabe wird in der Praxis allerdings kaum verwendet. Wird der VHDL-Code des Systems erweitert, muss die

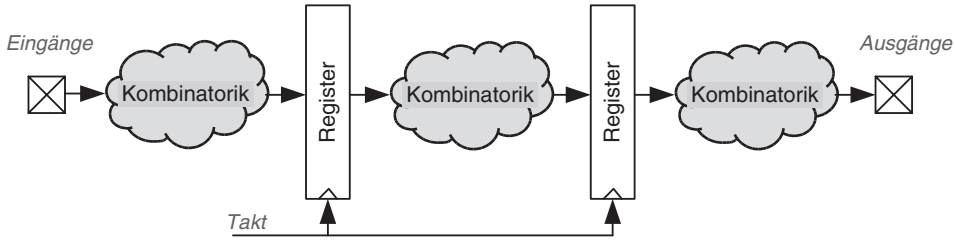


Abb. 7.7 Struktur eines RTL-Designs

Simulation wiederholt werden. Die Eingabe-Kommandos müssen wiederholt werden, was zeitaufwendig und fehlerträchtig ist. In der Praxis wird daher meist eine Methode gewählt, bei der der zu prüfende VHDL-Entwurf in eine Testbench eingebunden wird. Auch die Testbench wird in VHDL programmiert.

Für kleinere Entwürfe benötigt man häufig nur einfache Testbenches, die Eingangsdaten *(Stimuli)* für den zu testenden VHDL-Code erzeugen. Die Korrektheit des Entwurfs wird durch die manuelle Inspektion der Signalverläufe überprüft. Diese interaktive Simulation ist jedoch mit dem Nachteil verbunden, dass die Überprüfung manuell erfolgt und daher auch Fehler übersehen werden können.

Die bessere Variante ist eine selbstüberprüfende *(self-checking)* Testbench, bei der die Ausgaben des getesteten Codes mit erwarteten Ergebnissen verglichen werden. Hierzu müssen die erwarteten Werte zum Beispiel als Textdatei zur Verfügung stehen.

Die Stimuli werden von der Testbench aus einer Datei eingelesen und an das zu überprüfende Design angelegt. Die erwarteten Ausgabewerte des Systems werden durch ein sogenanntes *Known-Good-Device,* zum Beispiel eine Beschreibung als C-Programm, erzeugt. Die erwartete Ausgabe wird ebenfalls von der Testbench eingelesen und mit den Ausgabewerten des Designs verglichen. Eventuell auftretende Differenzen werden während der Simulation in einer Protokolldatei aufgezeichnet und können anschließend zur Fehlersuche verwendet werden. Das Prinzip der self-checking Testbench verdeutlicht Abb. 7.8.

Eine self-checking Testbench bietet unter anderem den Vorteil, dass Simulationen automatisiert gestartet werden können und so selbst aufwendige Tests ohne interaktiven Eingriff möglich sind. Dies ist insbesondere für komplexe Systeme vorteilhaft, deren Simulationszeit mehrere Stunden beträgt.

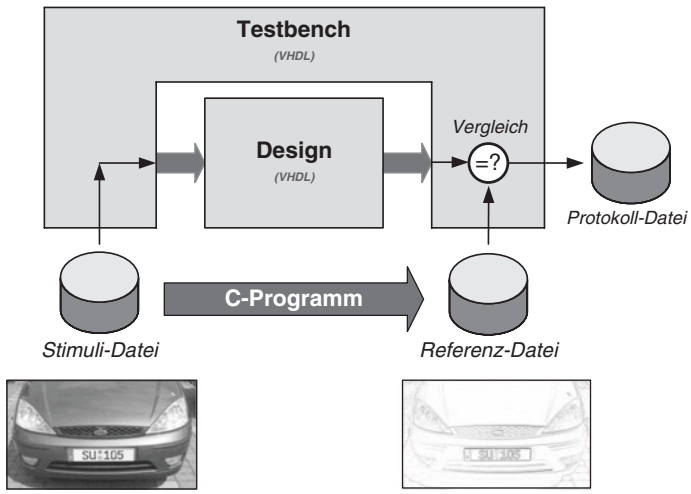


Abb. 7.8 Struktur einer selbstcheckenden Testbench

7.3.4 Synthese

Die Synthese umfasst das Einlesen und Analysieren des VHDL-Codes mit einer anschließenden Umsetzung der beschriebenen Funktion auf die verfügbaren digitalen Grundelemente. Das Ergebnis der Synthese ist eine sogenannte *Netzliste,* die Informationen über die benötigten Grundelemente und die Verbindungen zwischen den Elementen enthält.

Die genaue Platzierung der Elemente sowie deren exakte Verdrahtung bleiben bei diesem Schritt unberücksichtigt. Um die Verzögerungen durch die spätere Verdrahtung bereits bei der Synthese berücksichtigen zu können, werden statistische Modelle *(Wireload Models)* eingesetzt.

Die Synthese analysiert die VHDL-Beschreibung auch im Hinblick auf konstante Signale. Wird der Wert eines Signals als konstant erkannt, kann dieses zur Optimierung ausgenutzt werden, da die Logik, die an diesem Signal angeschlossen ist, vereinfacht oder im besten Fall komplett entfernt werden kann. Dieser Optimierungsschritt wird als *Constant Propagation* bezeichnet.

Ein Beispiel für die Optimierung von Konstanten zeigt das nachfolgende Codefragment. Für den Vergleich von *count* und *buf_size* realisiert die Synthese eine optimierte Hardware, die den Vergleich eines 4-Bit-Wertes mit der Konstanten 10 durchführt. Wäre *buf_size* dagegen ein Signal, das verschiedene Werte annehmen kann, müsste ein Vergleicher (also letztlich eine Subtraktion) von der Synthese implementiert werden.

```
architecture behave of my_module is
   constant buf_size : integer := 10;
   signal   count    : signed (3 downto 0);
begin
   process begin
        wait until rising_edge (clk);
        …
        if count > buf_size then -- Hier nutzt die Synthese aus, dass
             …                   -- buf_size eine Konstante ist
        end if;
   end process;
end;
```

Code ohne eine digitale Funktion wird von der Synthese erkannt und ignoriert. Im nachfolgend dargestellten Codeausschnitt wird dem Signal *q* auf eine etwas umständliche Weise der Wert Null zugewiesen. Dieses würde das Syntheseprogramm erkennen und das Design entsprechend optimieren. Nachdem von der Synthese *q* als konstant erkannt wurde, kann diese Information auch für weitere Optimierungsschritte auf Basis der *Constant Propagation* verwendet werden.

```
process (a,b,c)
   variable v1 : std_logic;
   variable v2 : std_logic;
begin
   v1 := a and b;
   v2 := (not a) and (not c);
   q <= v1 and v2 and c;
end process;
```

7.3.5 Platzierung und Verdrahtung

Nach dem Syntheseschritt erfolgt die Platzierung *(Placement* bzw. *Place)* und Verdrahtung *(Routing* bzw. *Route)* der identifizierten Grundelemente. Das Programm wählt für jedes Grundelement der Netzliste ein physikalisch vorhandenes Element des FPGA-Chips aus. Nach diesem Platzierungs-Schritt sind die Positionen aller Netzlistenelemente festgelegt. Nun werden die Ein- und Ausgänge der Elemente verbunden. Dazu muss das Routing-Programm die durch das Syntheseergebnis vorgeschriebenen Verbindungen herstellen.

Nachdem die Verdrahtung abgeschlossen ist, kann eine genauere Abschätzung des Zeitverhaltens erfolgen, da nun die exakten Verbindungsleitungen bekannt sind.

7.3.6 Timinganalyse

Bereits bei der Synthese sowie während Platzierung und Verdrahtung wird das Zeitverhalten der Schaltung überwacht und gegebenenfalls optimiert. Nach Abschluss der Verdrahtung steht das genaue Zeitverhalten der Schaltung fest und wird abschließend einer Timinganalyse unterzogen.

Das wichtigste Ergebnis der Timinganalyse ist die Information ob die Timing-Anforderungen eingehalten werden und wie groß der *Worst Negative Slack (WNS)* ist. Dieser Wert gibt die „Luft“ im kritischen Pfad des Designs an. Wenn beispielsweise ein *WNS* von 1 *ns* ausgegeben wird, bedeutet dies, dass alle Signale auch 1 *ns* später an den Eingängen der Flip-Flops erscheinen könnten, ohne dass es zu einer Verletzung der Setup-Zeit käme. Ist der WNS-Wert dagegen negativ, liegt ein Timingproblem vor. Die Kombinatorik der Schaltung ist zu langsam. Wenn man die Taktfrequenz nicht reduzieren kann, sind häufig Änderungen im VHDL-Code erforderlich (zum Beispiel der Einsatz von Pipelining, vgl. Kapitel 6).

Als Zusammenfassung wird auch der *Total Negative Slack (TNS)* angeben. Hierbei handelt es sich um die Summe aller Pfade, deren Zeitverhalten die Setup-Zeit der Flip-Flops verletzt. Pfade, deren Zeitverhalten nicht verletzt ist, werden bei der TNS-Analyse nicht berücksichtigt. Somit ist der TNS-Wert entweder negativ oder Null (falls keine Setup-Time-Verletzungen vorliegen).

In Analogie zur Analyse der Setup-Zeit wird auch eine Hold-Time-Analyse durchgeführt und der *WHS*- bzw. *THS*-Wert *(Worst Hold Slack* bzw. *Total Hold Slack)* ausgeben.

Diese Form der Analyse wird als *statische Timinganalyse* bezeichnet. Der Begriff „statisch" meint, dass das Zeitverhalten ohne die genaue Kenntnis des dynamischen Verhaltens der Signale, also ohne das Anlegen von Eingangsstimuli, durchgeführt wird.

Normalerweise ist diese Form der Analyse ausreichend. Allerdings ist zu beachten, dass die statische Timinganalyse pessimistisch ist. Sie überprüft alle Pfade eines Designs auf mögliche Verletzungen des Zeitverhaltens. Manchmal werden jedoch einige Pfade des Designs im praktischen Betrieb gar nicht verwendet. In diesem Fall kann eine *dynamische Timinganalyse* in Betracht gezogen werden. Darüber hinaus kann es in besonderen Fällen, zum Beispiel wenn das Design kritische Taktübergänge enthält, sinnvoll sein, eine dynamische Timinganalyse durchzuführen.

Für eine dynamische Timinganalyse wird das Design inklusive einer Modellierung der Verzögerungen der Grundelemente in einer Simulation überprüft. Hierzu müssen geeignete Eingangsstimuli definiert werden, die alle relevanten Pfade testen. Außerdem ist zu bedenken, dass die Komplexität der Simulation aufgrund der Modellierung des Zeitverhaltens deutlich höher ist als für die Simulation des VHDL-Quellcodes und daher eine größere Rechenzeit für die Simulation benötigt wird.

7.3.7 Inbetriebnahme

Nachdem ein Entwurf durch Simulation verifiziert wurde, kann er, wenn er als ASIC realisiert werden soll, in einer Halbleiterfabrik produziert werden. Soll das System auf Basis eines CPLDs oder eines FPGAs realisiert werden, erfolgt nach der Simulation die Programmierung des Bausteins mithilfe eines entsprechenden Programmiergerätes. Ein Beispiel einer Experimentierplatine mit angeschlossenem Programmiergerät ist in Abb. 7.9 dargestellt.

Trotz sorgfältiger Simulation kann es in der Praxis Fälle geben, die eine Fehlersuche im laufenden Betrieb erfordern. Dies kommt vor, wenn in der Anwendung Fälle auftreten, die in der Simulation nicht beachtet wurden oder aus Zeitgründen nicht simuliert werden konnten. Auch bei der Ansteuerung von externen Bauelementen, beispielsweise einem Speicher, kann es passieren, dass sich der reale Baustein etwas anders verhält, als dies in der Simulation vorhergesehen wurde.

Zur Fehlersuche, insbesondere bei komplexen FPGAs, ist es häufig nicht ausreichend, wenn nur die äußeren Anschlüsse des Systems zugänglich sind und der zeitliche Verlauf von internen Signalen nicht sichtbar ist. Um die Fehlersuche im Betrieb zu erleichtern, können dem Entwurf spezielle Module hinzugefügt werden, die in der Lage sind, den zeitlichen Verlauf interner Signale aufzuzeichnen und über eine Debug-Schnittstelle auszugeben. Auf diese Weise können die Zustände der internen Signale ähnlich wie in einer VHDL-Simulation visualisiert werden.

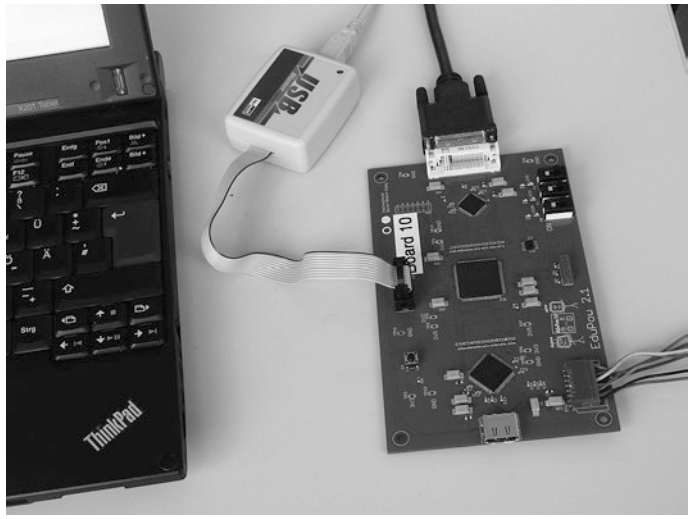


Abb. 7.9 FPGA-Experimentierplatine mit Programmiergerät

Der Vorteil dieses Vorgehens ist es, dass auch FPGA-interne Signale im laufenden Betrieb analysiert werden können. Auf der anderen Seite benötigt dieses Vorgehen aber mehr Ressourcen des FPGAs. So wird zum Beispiel für die Speicherung des zeitlichen Verlaufs der beobachteten Signale interner Speicher benötigt. Um den Hardwareaufwand für die Verifikation im Betrieb klein zu halten, wird daher meist nur ein relativ kurzes Zeitfenster aufgezeichnet. Darüber hinaus werden nur wenige besonders wichtige Signale für die Beobachtung im laufenden Betrieb ausgewählt. Da die Beobachtbarkeit der Signale gegenüber einer Simulation deutlich eingeschränkt ist, stellt dieses Vorgehen keinen Ersatz, sondern eine Ergänzung zur Simulation dar.

7.3.8 Der digitale Entwurf als iterativer Prozess

Die in diesem Kapitel beschriebenen Entwurfsschritte müssen bei komplexeren Designs unter Umständen mehrfach durchlaufen werden. Zeigt der erste Syntheselauf, dass das angestrebte Zeitverhalten nicht eingehalten werden kann oder das geplante Ressourcenbudget überschritten wird, kann bei kleinen Zielabweichungen versucht werden, durch geeignete Einstellungen der Entwurfsprogramme ein besseres Ergebnis zu erzielen. Bei größeren Abweichungen bleibt meist nur der Schritt zurück zum VHDL-Code, um zum Beispiel den zeitlich kritischen Pfad im Design zu optimieren. Bei sehr anspruchsvollen Designs können diese Änderungen nun wiederum Probleme an anderen Stellen des Codes nach sich ziehen, sodass der Designflow vom Schreiben des VHDL-Codes bis zur Platzierung und Verdrahtung mehrfach durchlaufen werden muss.

Für erste Schritte im FPGA-Design wird meist kein iteratives Vorgehen benötigt: Sind die Anforderungen an die Taktfrequenz moderat gewählt und die Anforderungen an den maximalen Ressourcenbedarf einer Schaltung von untergeordneter Bedeutung, wird man häufig bereits mit dem ersten Syntheseversuch ein zufriedenstellendes Ergebnis erzielen.

7.4 Übungsaufgaben

Prüfen Sie sich selbst mit den Fragen am Kapitelende. Die Lösungen und Antworten finden Sie am Ende des Buches.

Sofern nicht anders vermerkt, ist nur eine Antwort richtig.

Aufgabe 7.1
Welche Aussage ist im Hinblick auf einen Vergleich der Bausteine 74HC00 und 74AHC00 korrekt?

a) Beide Bausteine besitzen den gleichen Versorgungsspannungsbereich.
b) Die logischen Funktionen der Bausteine sind identisch.
c) Die logische Funktion der Bausteine ist vom Hersteller abhängig.
d) Der minimale High-Pegel an den Eingängen der Bausteine ist identisch.

Aufgabe 7.2
Was beschreibt der Begriff *Fan-out?*

a) Die Anzahl der Ausgänge eines Schaltkreises.
b) Die Anzahl der Leitungen die an einen Ausgang angeschlossen werden dürfen.
c) Ein Maß für die Last, die die Ausgänge des Bausteins treiben können.
d) Ein Maß für die Last, die ein Eingang des Bausteins darstellt.

Aufgabe 7.3
Was gilt für die unterschiedlichen Bausteine einer Familie (zum Beispiel „HC“) der 74er-Serie?

a) Alle Bausteine besitzen die gleiche Verzögerungszeit.
b) Eingänge der Bausteine müssen immer mit Ausgängen der gleichen Familie verbunden werden.
c) Für alle Bausteine wird vom Hersteller eine maximale Schaltzeit unabhängig von der Ausgangsbelastung garantiert.
d) Alle Bausteine besitzen den gleichen Versorgungsspannungsbereich.

Aufgabe 7.4
Welche Aussage trifft auf ASICs zu? *(Mehrere Antworten sind richtig)*

a) Für den Entwurf eines ASICs werden meist Bibliotheken mit Standardzellen verwendet.
b) Ein ASIC-Entwurf ist sowohl für kleine als auch für große Stückzahlen sinnvoll.
c) Ein ASIC-Entwurf ist mit relativ hohen Fixkosten verbunden.
d) Die digitale Funktion eines ASICs kann nicht mithilfe von VHDL beschrieben werden.

Aufgabe 7.5
Welche Aussagen treffen für den Vergleich eines Mikrocontrollers mit einem PC zu? *(Mehrere Antworten sind richtig)*

a) Mikrocontroller besitzen im Gegensatz zu einem PC keine Ein-/Ausgabe-Schnittstellen.
b) Mikrocontroller sind kostengünstiger als PCs.
c) Typische Mikrocontroller besitzen eine geringere Rechenleistung als PCs.
d) Typische Mikrocontroller besitzen eine geringere Speicherkapazität als PCs.

Aufgabe 7.6
Was meint der Begriff „Programmierbare Logik“?

a) Die Bausteine können Programme ausführen, die in Sprachen wie C oder Java geschrieben sind.
b) ASICs, die einen softwareprogrammierbaren Mikroprozessor beinhalten.
c) Die logische Funktion der Hardware des Bausteins kann durch den Anwender programmiert werden.
d) Logische Funktionen, die mithilfe eines Programms auf einem PC simuliert werden.

Aufgabe 7.7
Welches ist typische Reihenfolge der Entwurfsschritte?

a) Synthese, Platzierung, Verdrahtung
b) Platzierung, Verdrahtung, Synthese
c) Platzierung, Synthese, Verdrahtung
d) Synthese, Verdrahtung, Platzierung

Aufgabe 7.8
Welche Kombinationen von Worst Negative Slack (WNS) und Total Negative Slack (TNS) können in der Praxis auftreten? *(Mehrere Antworten sind richtig)*

a) WNS: -3 ns; TNS: -4 ns
b) WNS: -3 ns; TNS: 0 ns
c) WNS: +3 ns; TNS: +5 ns
d) WNS: 0 ns; TNS: 0 ns

VHDL-Vertiefung

8

In Kapitel 3 wurden die wichtigsten Sprachelemente von VHDL vorgestellt und Sie sind damit bereits in der Lage, digitale Schaltungen in VHDL zu entwerfen. In diesem Kapitel werden vertiefende Aspekte der Hardwarebeschreibung mit VHDL dargestellt. Einige dieser Sprachelemente eröffnen neue Möglichkeiten zur Beschreibung von Hardwarekomponenten. Andere können helfen, den Code besser zu strukturieren und lesbarer zu gestalten. Darüber hinaus werden in diesem Kapitel VHDL-Konstrukte vorgestellt, die zur Überprüfung der von Ihnen erstellten Hardwarebeschreibungen eingesetzt werden können. Nach dem Studium dieses Kapitels haben Sie die wichtigsten Aspekte der Sprache VHDL kennengelernt und können auch komplexere Schaltungen in VHDL realisieren.

8.1 Weitere Datentypen

Einige wichtige Datentypen sind bereits aus Kapitel 3 bekannt. In diesem Abschnitt werden weitere nützliche Datentypen behandelt.

8.1.1 Natural und Real

Der Datentyp *natural* dient zur Darstellung natürlicher Zahlen im Bereich von 0 bis $+2^{31}-1$, also dem Bereich der positiven Zahlen, der sich auch mit dem Datentyp *integer* darstellen lässt. Ergänzend zu den ganzzahligen Datentypen, bietet VHDL auch die Verwendung von Gleitkommazahlen an, die mit dem Datentyp *real* definiert werden können.

Im Gegensatz zum Datentyp *real* sind die Ganzzahl-Datentypen synthetisierbar. VHDL-Beschreibungen auf Basis dieser Datentypen können also in eine digitale

W. Gehrke et al., *Digitaltechnik,* Springer-Lehrbuch,
DOI 10.1007/978-3-662-49731-9_8

Hardware überführt werden, während die Verwendung von Gleitkommadatentypen auf Testbenches beschränkt ist.

8.1.2 Boolean

Wie viele Programmiersprachen unterstützt VHDL den Datentyp *boolean*. Diesem Datentyp können nur die Werte *true* oder *false* zugewiesen werden. Ein Objekt dieses Datentyps entspricht in Hardware einem einzelnen Bit. Die Bezeichnung der Werte erfolgt jedoch nicht mit 0 oder 1. Dies wäre dagegen syntaktisch inkorrekt (da es sich bei 0 und 1 um Werte vom Typ *integer* handelt) und würde zu Fehlermeldungen führen.

Ein häufiger Anwendungsfall für diesen Datentyp ist die Abfrage von Bedingungen. Werden beispielsweise zwei Werte verglichen, so ist das Ergebnis dieses Vergleichs vom Datentyp *boolean.* Selbstverständlich können auch Objekte, zum Beispiel Signale, mit diesem Datentyp angelegt werden, die dann in einer Abfrage ausgewertet werden.

8.1.3 Time

VHDL unterstützt die Verwendung von physikalischen Datentypen. Die Werte dieses Datentyps setzen sich aus einem Zahlenwert und einer Einheit zusammen. Der wichtigste physikalische Datentyp ist *time*. Dieser Datentyp erlaubt die Angabe von Zeiten mit den Einheiten Femtosekunde (fs), Picosekunde (ps), Nanosekunde (ns), Mikrosekunde (ms), Millisekunde (msec), Sekunde (sec), Minute (min) oder Stunde (hr).

Der Datentyp *time* ist nicht synthesefähig, da Zeitangaben im Zuge der Synthese ignoriert werden. Für Testbenches ist der Datentyp jedoch sehr hilfreich um das zeitliche Verhalten von Signalen nachzubilden. Ein Beispiel für die Verwendung des Datentyps *time* ist im nachfolgenden Codeausschnitt dargestellt. Das Signal *clk* wird durch eine Not-Anweisung invertiert. Durch Angabe einer zeitlichen Verzögerung mithilfe des Schlüsselworts *after* ergibt sich ein Signal, welches alle 5 Nanosekunden invertiert wird. Auf diese Weise wird also ein digitales Taktsignal modelliert, welches eine Periodendauer von 10 ns besitzt. Die Definition des Signals *clk* beinhaltet die initiale Zuweisung des Wertes 0. Auf diese Weise wird sichergestellt, dass *clk* zu Beginn der Simulation einen definierten Wert erhält.

```
signal clk : std_logic := '0';
…
clk <= not clk after 5 ns;
```

Auch die Definition eigener physikalischer Datentypen ist in VHDL möglich. Allerdings wird hiervon selten Gebrauch gemacht, sodass dieser Aspekt hier nicht weiter vertieft wird.

8.1.4 Std_ulogic, Std_ulogic_vector

Neben den Datentyp *std_logic* und *std_logic_vector* wird im IEEE-Paket auch der Datentyp *std_**u**logic* und *std_**u**logic_vector* definiert. Es handelt sich dabei um eine Alternative zu den Datentypen *std_logic* und *std_logic_vector.* Diese bereits vorgestellten Datentypen haben eine sogenannte Auflösungsfunktion (engl. *resolution function*). Die Auflösungsfunktion ist immer dann relevant, wenn einem Signal gleichzeitig zwei Werte zugewiesen werden. Mithilfe der beim Datentyp *std_logic* definierten Auflösungsfunktion wird für diese Fälle der sich ergebende Wert des Signals bestimmt. Wird einem Signal beispielsweise gleichzeitig der Wert 0 und der Wert 1 zugewiesen, wäre das Ergebnis bei Verwendung von *std_logic* der Wert *X (unknown).*

In den Datentypen *std_**u**logic* und *std_**u**logic_vector* steht das „u“ für *unresolved* und drückt aus, dass für diesen Datentyp keine Auflösungsfunktion existiert. Werden einem Signal gleichzeitig zwei Werte zugewiesen, würden die Entwurfswerkzeuge bereits beim Übersetzungsvorgang der VHDL-Beschreibung einen Fehler ausgeben. Es ist eine individuelle Entscheidung, ob diese Eigenschaft als ein Vorteil angesehen wird. In der Praxis werden die meisten VHDL-Beschreibungen auf Basis des Datentyps *std_logic* geschrieben. Daher wird in diesem Buch auf die Verwendung des Datentyps *std_ulogic* verzichtet.

8.1.5 Benutzerdefinierte Datentypen

Mithilfe des Schlüsselwortes *Type* können in VHDL auch benutzerdefinierte Datentypen, zum Beispiel für die Codierung der Zustände eines endlichen Automaten (vgl. Kapitel 5) angelegt werden.

Die Definition des benutzerdefinierten Typs Farbe kann zum Beispiel wie folgt formuliert werden:

```
type farbe is (rot,gruen,blau,lila);
```

8.1.6 Zeichen und Zeichenketten

Für einzelne Zeichen bietet der VHDL-Standard den Datentyp *character* an. Dieser Datentyp ist ein Aufzählungstyp, der insgesamt 256 Werte umfasst, wobei die ersten 128 Werte dem 7-Bit-ASCII-Code (vgl. Kapitel 2) entsprechen und die letzten 128 Werte Umlaute und Sonderzeichen enthalten. Da die Definition des Datentyps im Paket *std* erfolgt, kann der Datentyp ohne Use-Anweisung in allen VHDL-Beschreibungen eingesetzt werden. Die Typdefinition zeigt der folgende Codeausschnitt:

```
type character is (
NUL, SOH, STX, ETX, EOT, ENQ, ACK, BEL,
BS, HT, LF, VT, FF, CR, SO, SI,
DLE, DC1, DC2, DC3, DC4, NAK, SYN, ETB,
CAN, EM, SUB, ESC, FSP, GSP, RSP, USP,
' ', '!', '"', '#', '$', '%', '&', ''',
'(', ')', '*', '+', ',', '-', '.', '/',
'0', '1', '2', '3', '4', '5', '6', '7',
'8', '9', ':', ';', '<', '=', '>', '?',
'@', 'A', 'B', 'C', 'D', 'E', 'F', 'G',
'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O',
'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W',
'X', 'Y', 'Z', '[', '\', ']', '^', '_',
'`', 'a', 'b', 'c', 'd', 'e', 'f', 'g',
'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o',
'p', 'q', 'r', 's', 't', 'u', 'v', 'w',
'x', 'y', 'z', '{', '|', '}', '~', DEL,
-- weitere 128 Werte
);
```

Ähnlich wie für den Datentyp *std_logic* existiert ein zugehöriger vektorieller Datentyp mit dem Namen *string,* in dem Zeichenketten abgelegt werden können. Der folgende Code zeigt einige Beispiele zur Verwendung der Datentypen.

```
signal i : integer;
signal my_char : character;
signal my_string : string(1 to 10) := "Hallo Welt";
my_string(7 to 10) <= "VHDL";   -- my_string enthält danach "Hallo VHDL"
my_string(6) <= '_';            -- my_string enthält danach "Hallo_Welt"
my_char <= my_string(1);        -- my_char enthält danach 'H'
```

8.1.7 Subtypes

Man kann von deklarierten Typen weitere Typen *(subtype)* ableiten. Ein Subtype ist ein Datentyp mit eingeschränktem Wertebereich im Vergleich zum Basistyp. Die Syntax zur Definition eines Subtypes lautet:

```
subtype <subtype_name> is <subtype_indication>;
```

Die *subtype_indication* enthält den Namen des Basisdatentyps und optional eine Einschränkung, welcher Bereich des Basisdatentyps dem neu definierten Subtype zur Verfügung stehen soll.

```
-- Subtype Beispiele:
subtype dezimal_ziffer is integer range 0 to 9; -- Bereichseinschränkung
subtype byte is std_logic_vector (7 downto 0); -- Indexeinschränkung
subtype ganze_zahl is integer; -- ganze_zahl = anderer Name für Integer

-- Beispiele für vordefinierte Subtypes:
subtype natural is integer range 0 to integer'high;
subtype positive is integer range 1 to integer'high;
subtype std_logic is resolved std_ulogic;
subtype X01 is resolved std_ulogic range 'X' to '1'; -- ('X','0','1')
```

Die Angabe *resolved* bedeutet, dass für den hier definierten Datentyp eine Auflösungsfunktion definiert ist.

Bei der Definition der Subtypes *natural* und *positive* wird das Attribut *high* verwendet. Mithilfe dieses Attributs wird der größte Zahlenwert des Typs *integer* ausgewählt. Der Ausdruck *integer'high* ist also gleichbedeutend mit +2147483647.

8.1.8 Arrays

Wie alle Programmiersprachen unterstützt auch VHDL Arrays, also Felder von beliebigen Datentypen. Die Definition eines Arrays ist in VHDL etwas umständlicher gelöst als in den meisten Programmiersprachen, da man zunächst das gewünschte Array als neuen Datentyp definieren muss. Erst anschließend darf dieser neue Datentyp für die Definition von Signalen oder Variablen verwendet werden. Die Typdefinition eines Arraydatentyps sieht wie folgt aus:

```
type <type_name> is array (range) of <element_data_type>;
```

Nehmen wir an, Sie möchten ein Array aus 10 Integer-Werten anlegen. Dann sehen die Typdefinition und die Definition eines entsprechenden Array-Signals zum Beispiel so aus:

```
type my_int_array_type is array (1 to 10) of integer; -- neuer Typ
signal my_ints : my_int_array_type; -- Signal auf Basis des neuen Typen
```

Ein Zugriff auf das Array erfolgt dann genauso wie beim Zugriff auf einzelne Elemente eines Signals vom Typ *std_logic_vector* (denn der Datentyp *std_logic_vector* ist auch ein Array-Datentyp):

```
my_ints(6) <= 24;
```

Selbstverständlich kann man auch mehrdimensionale Arrays anlegen, wenn man die Typdefinitionen verschachtelt:

```
type my_int_array_type_1D is array (1 to 20) of integer;
type my_int_array_type_2D is array (1 to 10) of my_int_array_type_1D;
signal my_2D_ints : my_int_array_type_2D;
…
my_2D_ints(7)(5) <= 12; -- zweidimensionaler Arrayzugriff
```

Arrays werden häufig benötigt, um Speicher zu modellieren. Sie wollen zum Beispiel einen Speicher der Größe 1 kByte modellieren. Dies erreichen Sie mit folgendem Code:

```
type my_mem_type is array (0 to 1023) of std_logic_vector (7 downto 0);
signal mem : my_mem_type;
```

8.1.9 Records

VHDL unterstützt Records, also das Zusammenfassen mehrerer Werte in einem neuen Datentyp. Dies ist mit *Structs* vergleichbar, die Sie vielleicht aus einer Programmiersprache bereits kennen. Die allgemeine Form einer Record-Definition sieht wie folgt aus:

```
type <record_type_name> is
   element_name : element_typ;
   {element_name : element_typ;} -- Ggf. beliebig viele weitere Elemente
end record [record_type_name];  -- record_type_name ist optional
```

Die Definition und Verwendung von Records wird durch die nachfolgenden Beispiele verdeutlicht:

```
type bus_mosi is
   addr : std_logic_vector(31 downto 0);
   data : std_logic_vector(31 downto 0);
   rd   : std_logic;
   wr   : std_logic;
end record;

type bus_miso is
   data  : std_logic_vector(31 downto 0);
   ready : std_logic;
end record;
```

Wenn Sie Records angelegt haben, dürfen Sie den Datentyp wie jeden anderen Datentyp verwenden. Sehr praktisch kann es sein, Records für die Ports eines Moduls einzusetzen: Wenn viele Signale gemeinsam zu verdrahten sind (zum Beispiel Bussignale, die von einem Master an mehrere Slaves anzuschließen sind), können Records die Lesbarkeit des Codes verbessern.

Der Zugriff auf die Elemente eines Records erfolgt über *selected names,* den „Punkt-Operator“:

```
signal bus_out : bus_mosi;
signal bus_in : bus_miso;
… -- weiterer VHDL-Code
bus_out.addr <= x"1234_5678"; -- Zugriff auf die Elemente des Records
bus_out.rd   <= '1';
bus_out.wr   <= '0';
…
data_in <= bus_in.data;
```

8.2 Sprachelemente zur Code-Strukturierung

VHDL unterstützt den Entwicklungsprozess mit einigen nützlichen Sprachelementen bei der Strukturierung des Codes. Einige der Konstrukte sind in ähnlicher Form auch in Software-Programmiersprachen vorhanden.

8.2.1 Function

Eine VHDL-Funktion (Schlüsselwort: *function*) dient dazu, aus einem oder mehreren Übergabeparametern einen Rückgabewert zu berechnen. Wichtige Eigenschaften von Funktionen sind:

- Funktionen haben immer exakt einen Rückgabewert. Die Rückgabe erfolgt mithilfe des Schlüsselwortes *return.*
- Die Parameter dürfen innerhalb der Funktion nur gelesen werden. Schreibzugriffe sind nicht erlaubt.
- Innerhalb von Funktionen können lokale Variablen oder Konstanten definiert werden. Die Variablen werden mit jedem Funktionsaufruf neu initialisiert. Mit anderen Worten: Wird einer Variablen ein Wert zugewiesen, steht dieser beim nächsten Aufruf der Funktion nicht mehr zur Verfügung.
- Funktionen dürfen keine Wait-Anweisungen enthalten.
- Funktionen dürfen keine Signalzuweisungen enthalten.
- Funktionen dürfen sowohl Funktionen als auch Prozeduren (s. u.) aufrufen. Auch rekursive Aufrufe (eine Funktion ruft sich selbst auf) sind erlaubt.

Die syntaktische Struktur einer VHDL-Funktion stellt der nachfolgende Code dar.

```
function <Funktionsname> ({<Parameterliste>}) return <Typ_Rückgabe-
wert> is
   <Deklarationen>
begin
   <Anweisungen>
end function;
```

Funktionen dürfen im Deklarationsteil einer Architecture (also vor dem *begin*) oder in Paketen definiert werden.

Als ein Beispiel ist im Folgenden eine VHDL-Funktion zur Umwandlung vom Gray-Code in eine Dualzahl dargestellt.

Die Funktionsdefinition verwendet den Datentyp *std_logic_vector* ohne die Länge des Vektors zu spezifizieren. Auf diese Weise können durch die Funktion Vektoren mit einer beliebigen Länge verarbeitet werden. Allerdings wird für die Implementierung der Funktion die Länge des jeweils bei Aufruf der Funktion übergebenen Vektors benötigt. Diese lässt sich sehr elegant mithilfe des *length*-Attributs des Vektors bestimmen. Die Schreibweise *gray_val'length* liefert die Länge (Anzahl der Elemente) des Vektors *gray_val* und wird zu Beginn der Funktion genutzt.

```
-- Definition der Funktion Gray2Bin
function Gray2Bin (gray_val : std_logic_vector) return std_logic_vector
is
   constant vlen  : integer := gray_val'length;
   variable temp : std_logic_vector(vlen-1 downto 0);
begin
   temp := gray_val;
   if vlen > 1 then
        for i in vlen-2 downto 0 loop
            temp(i) := gray_val(i) xor temp(i+1);
        end loop;
   end if;
   return temp(vlen-1 downto 0);
end function;

-- Beispiel für den Aufruf der Funktion Gray2Bin
     …
     bin <= Gray2Bin(gray);
     …
```

8.2.2 Procedure

VHDL-Prozeduren können ebenso wie Funktionen im Deklarationsteil einer Architecture oder in Paketen definiert werden.

Im Gegensatz zu Funktionen können Prozeduren mehrere Rückgabewerte besitzen. Die Rückgabe der Ergebnisse einer Prozedur erfolgt durch Modifikation der Werte der übergebenen Parameter und es ist daher erlaubt, auf die übergebenen Parameter schreibend zuzugreifen. Um festzulegen, ob ein Parameter nur gelesen, nur beschrieben oder sowohl gelesen als auch beschrieben werden darf, wird mit den Parametern eines der Schlüsselwörter *in, out* oder *inout* angegeben.

Als Parameter können Variablen, Signale oder Konstanten verwendet werden. Bei der Definition einer Prozedur muss festgelegt werden, welcher der drei Parameterklassen von der Prozedur erwartet wird.

Ein weiterer Unterschied zu Funktionen ist, dass innerhalb einer Prozedur Zuweisungen an Signale erlaubt sind, wenn die Prozedur innerhalb eines Prozesses definiert wird.

Darüber hinaus dürfen Wait-Anweisungen in Prozeduren verwendet werden. Allerdings sind diese Prozeduren dann nicht mehr synthetisierbar und der Einsatz solcher Prozeduren bleibt auf Testbenches beschränkt.

Der grundlegende Aufbau einer VHDL-Prozedur ist einer Funktion recht ähnlich:

```
procedure <Prozedurname> (<Parameterliste>) is
   <Deklarationen>
begin
   <Anweisungen>
end procedure;
```

Ein Beispiel für eine VHDL-Prozedur zeigt der nachfolgende Code, der eine Sortierung von drei Signalen implementiert.

```
-- Prozedur sort_u3
-- Sortiert 3 Werte vom Datentyp unsigned
procedure sort_u3 (signal val1 : in  unsigned;
                   signal val2 : in  unsigned;
                   signal val3 : in  unsigned;
                   signal min  : out unsigned;
                   signal med  : out unsigned;
                   signal max  : out unsigned) is

   variable min_v : unsigned(min'length-1 downto 0);
   variable med_v : unsigned(med'length-1 downto 0);
   variable max_v : unsigned(max'length-1 downto 0);
   variable tmp_v : unsigned(min'length-1 downto 0);

begin
   max_v := val1;
   med_v := val2;
   min_v := val3;
```

```
   if min_v >= med_v then -- min/med tauschen?
      tmp_v := med_v;
      med_v := min_v;
      min_v := tmp_v;
   end if;

   if med_v >= max_v then -- max/med tauschen?
      tmp_v := max_v;
      max_v := med_v;
      med_v := tmp_v;
   end if;

   if min_v >= med_v then -- und noch einmal ggf. min/med tauschen
      tmp_v := med_v;
      med_v := min_v;
      min_v := tmp_v;
   end if;

   min <= min_v;
   med <= med_v;
   max <= max_v;
end procedure;

-- Beispiel für den Aufruf der Procedure
…
sort_u3 (sig_1,sig_2,sig_3,sig_min,sig_med,sig_max);
-- alle sechs Signale müssen vom Typ unsigned sein
-- und die gleiche Wortbreite besitzen
…
```

8.2.3 Entity-Deklaration mit Generics

Stellen Sie sich vor, Sie möchten eine logische Funktion in VHDL realisieren, die Signale vom Typ *std_logic_vector* verknüpft. Da es sich um eine grundlegende Funktion handelt, die Sie häufig benötigen, muss Sie für Vektoren mit unterschiedlicher Wortbreite zur Verfügung stehen.

Natürlich kann man für jede benötigte Wortbreite ein eigenes Entity-Architecture-Paar realisieren. Allerdings kann dies sehr aufwendig werden, wenn viele unterschiedliche Wortbreiten benötigt werden. Es wäre eleganter, wenn man der Instanz des Moduls „irgendwie“ die benötigte Wortbreite als Parameter mitteilen könnte. Wenn dieser Parameter in der Entity und der Architecture des instanziierten Moduls entsprechend berücksichtigt werden würde, kann die Erstellung eines einzelnen Entity-Architecture-Paares ausreichend sein.

Um einem Modul während der Instanziierung Parameterwerte übergeben zu können, muss die Entity des Moduls neben einer Port-Liste eine auch eine Parameter-Liste (Schlüsselwort *Generic*) enthalten.

Diese Parameter *(Generics)* können dann in symbolischer Form bei der Beschreibung des Moduls verwendet werden. Erst mit der Instanziierung des Moduls werden die (für diese Instanz) zu verwendenden Werte der Parameter festgelegt.

In der Praxis werden Generics häufig mit dem Datentyp *integer* oder *natural* definiert. Aber auch alle anderen VHDL-Datentypen sind zulässig und können für bei der Definition eines Generics eingesetzt werden.

Ein Beispiel soll die Vorgehensweise verdeutlichen: Angenommen Sie möchten ein Modul erstellen, das ein Signal um eine bestimmte Anzahl von Taktzyklen verzögern soll. Dieses Modul soll möglichst flexibel sein und für beliebige Wortbreiten oder Verzögerungen einsetzbar sein. Das Modul kann mithilfe von Generics wie folgt realisiert werden:

```
library ieee;
use ieee.std_logic_1164.all;

entity delay_unit is
   generic (D   : natural := 3;  -- Anzahl der Verzögerungszyklen (D>0 !)
            N   : natural := 8); -- Breite der verzögerten Werte (N>0 !)
   port (clk   : in  std_logic;
         d_in  : in  std_logic_vector(N-1 downto 0);
         d_out : out std_logic_vector(N-1 downto 0));
end;

architecture behave of delay_unit is
     -- Hier legen wir ein Array mit D Einträgen an
     -- Jeder Eintrag nimmt N Bits auf
     --
     -- Durch die Synthese wird eine Kette von D Registern (also D-FFs)
     -- mit der Wortbreite N implementiert
   type d_arr_type is array (0 to D-1) of std_logic_vector(N-1 downto
0);
   signal d_array : d_arr_type;
begin
   process begin
      wait until rising_edge(clk);
      for i in 0 to (D-2) loop -- Werte in der FF-Kette verschieben
           d_array(i) <= d_array(i+1);
      end loop;
      d_array(D-1) <= d_in; -- Eingangswert an oberster Position
      -- der FF-Kette abspeichern
   end process;
   d_out <= d_array(0); -- ältesten Wert ausgeben
end;
```

Bei der Instanziierung des Moduls erfolgt nun neben der Portzuordnung *(port map)* auch die Zuordnung der verwendeten Generics *(generic map)*. Ist bei der Deklaration des Parameters in der Entity ein Default-Wert angegeben worden, kann die Parameterzuordnung auch entfallen. In diesem Fall wird für diese Instanz der angegebene Default-Wert verwendet.

Die Werte, die den Generics bei der Instanziierung zugeordnet werden, müssen zur Übersetzungszeit des bekannt VHDL-Codes berechenbar sein. Werte, die sich erst während der Simulation ergeben, sind nicht erlaubt. So ist es beispielsweise nicht möglich, einem Generic ein Signal zuzuweisen.

Der folgende Code zeigt die Instanziierung des oben beschriebenen Moduls.

```
…
-- Verwendung der Default-Werte für die Parameter D und N,
-- also D=3 und N=8
u0 : delay_unit port map (clk => clk, d_in => x_sv8, d_out => q_sv8);

-- Überschreiben der Default-Werte: D=5, N=32
-- Die Ein- und Ausgänge dieser Instanz haben die Wortbreite 4
u1 : delay_unit
   generic map (D=> 5, N => 32)
   port map (clk => clk, d_in => x_sv32, d_out => q_sv32);
…
```

8.2.4 Generate-Anweisung

In manchen Fällen lassen sich Parameter sehr elegant in einer Generate-Anweisung verwenden. Die Generate-Anweisung existiert in den beiden Varianten *if-generate* und *for-generate* und dient der bedingten beziehungsweise wiederholten Ausführung nebenläufiger Anweisungen wie Signalzuweisungen, Prozesse oder Instanziierungen.

Die allgemeine Schreibweise der beiden Generate-Anweisungen lautet

```
<Name>: if <Bedingung> generate
   <Nebenläufige Anweisungen>
end generate;

<Name>: for <Laufindex> in <Bereich> generate
   <Nebenläufige Anweisungen>
end generate;
```

Mithilfe der If-Generate-Anweisung können nebenläufige Anweisungen mit einer Bedingung versehen werden. Nur wenn die Bedingung erfüllt ist, ist dieser Code aktiv. Auf diese Weise können zum Beispiel Instanziierungen oder Prozesse in Abhängigkeit von Generics aktiviert werden.

Betrachten wir hierzu das Beispiel des Moduls *delay_unit* aus dem vorangegangenen Abschnitt. Das Modul kann nur eingesetzt werden, wenn die Verzögerung mindestens einen Taktzyklus beträgt, also $D > 1$ gilt. Würde D zu 0 gewählt werden, würde die Zuweisung

```
d_array(D-1) <= d_in;
```

auf *d_array(−1)* zugreifen. Dieser Feldindex existiert jedoch nicht, da der kleinste mögliche Index 0 ist. Eine Fehlermeldung wäre die Folge.

Möchte man auch die Auswahl $D = 0$ (also keine Verzögerung des Signals) ermöglichen, kann dies mithilfe der If-Generate-Anweisung realisiert werden. Da bei der If-Generate-Anweisung kein *else* unterstützt wird, werden zwei If-Generate-Anweisungen benötigt. Der VHDL-Code kann wie folgt aussehen:

```
entity my_module is
   generic (delay_count : natural := 1);
   port (clk : in std_logic;
          -- weitere Ports
         );
end;

architecture behave of my_module is
   signal q_sv32, x_sv32 : std_logic_vector (31 downto 0);
begin
-- Prozesse und nebenläufige Zuweisungen dieses Moduls
GEN_D0: if delay_count = 0 generate -- Ein Label muss sein
        -- delay_count = 0, also direkte Zuweisung
        q_sv32 <= x_sv32;
   end generate;

GEN_D1: if delay_count > 0 generate
   -- delay_count > 0, also das Modul einbauen
   -- für die Wortbreite N wird der Defaultwert (32)
   -- aus der Entity-Definition der Delay_Unit genutzt
   u1 : delay_unit
         generic map (D => delay_count)
         port map (clk => clk, d_in => x_sv32, d_out => q_sv32);
   end generate;
end;
```

Die For-Generate-Anweisung wird für eine wiederholte Ausführung nebenläufiger Zuweisungen oder Modul-Instanziierungen eingesetzt. Der Einsatz dieser Anweisung wird im Folgenden anhand eines sehr einfachen Beispiels verdeutlicht. Nehmen wir an, Sie haben ein AND2-Modul, also ein UND-Gatter mit zwei Eingängen realisiert und möchten dieses für die VHDL-Beschreibung eines UND-Gatters mit N Eingängen verwenden.

Eine mögliche Lösung mithilfe der For-Generate-Anweisung kann dann wie folgt formuliert werden:

```
architecture for_gen_arch of and_n is
begin
   AND2GEN: for i in 0 to N-1 generate
        ui : and_2 port map (a => a(i), b => b(i), q => q(i));
   end generate;
end;
```

Beide Formen der Generate-Anweisung sollten nicht mit ähnlichen Sprachkonstrukten für Prozesse verwechselt werden. Die If- und For-Anweisungen in Prozessen beinhalten sequenziell ausgeführten Code, der Teil eines Prozesses ist. Die Generate-Anweisung bezieht sich dagegen immer auf nebenläufigen Code, beispielsweise Signalzuweisungen, Prozesse oder Instanziierungen.

Insbesondere müssen die Bereichsgrenzen der For-Generate-Anweisung beziehungsweise die Bedingung der If-Generate-Anweisung zum Zeitpunkt der Übersetzung des Moduls berechenbar sein. Der Grund hierfür ist, dass aus dem VHDL-Code Hardware generiert wird und daher bekannt sein muss, wie viele und welche Schaltungselemente erzeugt werden sollen. Es wäre beispielsweise nicht möglich, in einer If-Generate-Bedingung den Wert eines Signals abzufragen. Da sich der Wert des Signals erst während der Simulation oder während des Betriebs der Hardware ergibt, ist die Bedingung zum Übersetzungszeitpunkt des Moduls nicht auflösbar und würde Fehlermeldungen bei der Übersetzung des VHDL-Codes zur Folge haben.

8.2.5 Attribute

Mit Attributen lassen sich Eigenschaften von Objekten und Typen abfragen. VHDL-Beschreibungen können hiermit teilweise kürzer oder eleganter realisiert werden. Der Wert eines Attributs kann in einem VHDL-Modell weiter verwendet werden. Attribute lassen sich auf viele Datentypen anwenden, beispielsweise lässt sich die Anzahl der Elemente in einem Vektor bestimmen. Die generelle Syntax für Verwendung von Attributen lautet:

```
<typ_name>'<attribut_bezeichner>
```

Die Werte der Attribute unterscheiden sich von den Datenobjektwerten. VHDL unterscheidet vordefinierte und benutzerdefinierte Attribute. Die wichtigsten vordefinierte Attribute sind: '*left*, '*right*, '*high*, '*low*, '*length*, '*pos*, '*val* und '*range*.

Der folgende Code zeigt einige Beispiele zur Verwendung von Attributen:

```
process
   type farben_typ is (rot, gruen, blau, gelb, lila);
   variable farbe : farben_typ;
   variable i     : integer;
   variable c     : character := 'A';
   variable slv   : std_logic_vector (7 downto 0);
begin
   farbe := farben_typ'left;   -- liefert: rot
   farbe := farben_typ'right;  -- liefert: lila
   i := slv'low;               -- liefert: 0 (kleinster Indexwert)
   i := slv'high;              -- liefert: 7 (höchster Indexwert)
   i := slv'length;            -- liefert: 8 (Länge des Vektors)
   i := character'pos(c);      -- liefert: 65 (= ASCII-Wert von 'A')
   c := character'val(65);     -- liefert: 'A'(= Zeichen an Position 65)
   wait;
end process;
```

In manchen VHDL-Beschreibungen findet sich das Attribut '*event* in Verbindung mit Signalen. Falls innerhalb eines VHDL-Modells eine Flanke des Signals *clk* eine Aktion bewirken soll, so lässt sich diese Flanke auch durch die Bedingung *if clk'event and clk = '1' then* abfragen.

Die folgenden Schreibweisen beschreiben beispielsweise ein D-Flip-Flop:

```
-- D-FF mit der IEEE-Funktion rising_edge()
process begin
   wait until rising_edge(clk);
   q <= d;
end process;

-- D-FFs mit Abfrage des Attributs 'event
-- Diese Schreibweise ist nicht empfehlenswert
process begin
   -- Prozess unterbrechen bis ein Ereignis (Zuweisung eines neuen
   -- Wertes) auf dem Signal clk stattgefunden hat UND das Signal
   -- den Wert 1 angenommen hat
   wait until clk'event and clk='1';
   q <= d;
end process;
```

In manchen VHDL-Beschreibungen ist die Schreibweise *clk'event and clk = '1'*zu finden. Allerdings deckt diese Schreibweise alle Signalwechsel ab, bei denen das abgefragte Signal *clk* von einem Wert ungleich *'1'* auf *'1'* wechselt und sollte daher nicht verwendet werden.

So würde beispielsweise ein Wechsel von *'H'* zu *'1'* in der Simulation als steigende Flanke interpretiert. Dies ist jedoch inkorrekt, da *'H'* eine „schwache Eins" und *'1'* eine „starke 1" darstellt. Der Wechsel von *'H'* zu *'1'* stellt also keine steigende Flanke dar. Demgegenüber würde beispielsweise ein Wechsel von *'0'* zu *'H'* welcher eine steigende Flanke darstellt, nicht als solche erkannt werden.

Die falsch interpretierten Signalwechsel wirken sich nur in der Simulation aus. Die synthetisierte Hardware, die ja nur Nullen und Einsen kennt, würde sich dagegen korrekt – und damit anders als die Simulation – verhalten.

Für die Erkennung einer Taktflanke wird darum die Verwendung der Funktion *rising_edge()* (beziehungsweise *falling_edge()* für fallende Signalflanken) empfohlen, die expliziter und damit besser lesbar ist.

8.2.6 Instanziierung mit der Component-Anweisung

In Kapitel 3 wurde die Instanziierung von Modulen durch Angabe der Bibliothek und der Entity bereits vorgestellt. Im Folgenden wird eine alternative Vorgehensweise zur Instanziierung von Modulen beschrieben, die ebenfalls sehr häufig angewendet wird. Daher wird Ihnen diese Variante dann begegnen, wenn Sie beispielsweise VHDL-Code aus Internet-Quellen verwenden möchten.

Angenommen Sie haben ein Modul beschrieben und möchten dieses in einem anderen Modul verwenden. Als Beispiel verwenden wir ein einfaches UND-Modul mit zwei Eingängen. Die Entity des Grundmoduls kann wie folgt aussehen:

```
entity and_2 is
port (a : in  std_logic;
      b : in  std_logic;
      q : out std_logic);
end;
```

In der alternativen Beschreibung ohne Angabe der VHDL-Bibliothek wird eine Component-Anweisung verwendet. Diese Anweisung macht das zu instanziierende Modul in der Architecture bekannt und anschließend kann das Modul beliebig oft in der VHDL-Architecture verwendet werden.

Die Component-Anweisung beschreibt im Wesentlichen die Anschlüsse des zu instanziierenden Moduls und ist der Entity-Deklaration des Moduls sehr ähnlich: Im Gegensatz zur Entity-Deklaration wird statt des Schlüsselwortes *entity* das Schlüsselwort *component* verwendet.

Die Component-Anweisung des UND-Gatters würde wie folgt aussehen:

```
component and_2 is
port (a : in  std_logic;
```

```
      b : in  std_logic;   -- Sieht fast wie die Entity aus…
      q : out std_logic  ) -- Aber: Nach der Klammer kein Semikolon
end component;
```

Die Instanziierung des damit bekannt gemachten Moduls beginnt (wie bei der bereits bekannten Entity-Instanziierung) mit einem eindeutigen Namen für diese Instanz. Nach einem Doppelpunkt wird die Komponente (in diesem Beispiel *and_2*) angeben. Darauf folgt die Zuordnung der Anschlüsse, die mit den Schlüsselwörtern *port map* eingeleitet wird.

Für das Beispiel eines Vierfach-UND-Moduls, welches UND-Gatter instanziiert, können Entity und Architecture wie folgt beschrieben werden:

```
library ieee;
use ieee.std_logic_1164.all;

entity and_4x2 is
   port (a : in  std_logic_vector (3 downto 0);
         b : in  std_logic_vector (3 downto 0);
         q : out std_logic_vector (3 downto 0));
end;

architecture behave of and_4x2 is

   component and_2 is
        port (a : in  std_logic;
              b : in  std_logic;
              q : out std_logic);
   end component;

begin
     u0 : and_2 port map (a => a(0), b => b(0), q => q(0));
     u1 : and_2 port map (a => a(1), b => b(1), q => q(1));
     u2 : and_2 port map (a => a(2), b => b(2), q => q(2));
     u3 : and_2 port map (a => a(3), b => b(3), q => q(3));
end;
```

Die in Kapitel 3 eingeführte Entity-Instanziierung und Instanziierung mit der Component-Anweisung sind gleichwertig und letztlich eine Frage des bevorzugten „Coding-Styles“. Dennoch sollte man die Varianten kennen, da beide in der Praxis verwendet werden.

8.2.7 Pakete

Einige häufig verwendete Bibliotheken und die darin enthaltenen Pakete *(Packages)* wurden in den vorangegangenen Abschnitten bereits verwendet. Pakete sind immer dann

sinnvoll, wenn grundlegende Funktionen oder Datentypen in mehreren VHDL-Dateien verwendet werden sollen.

In einem Paket können unterschiedliche VHDL-Elemente abgelegt sein. Dies sind in der Praxis neben selbst definierten Datentypen, Funktionen oder Prozeduren häufig auch Component-Anweisungen. Wird beispielsweise ein Paket, das Component-Anweisungen enthält, in einer VHDL Beschreibung durch geeignete Library- und Use-Anweisungen bekannt gemacht, können die hierin enthaltenen Component-Anweisungen im nachfolgenden Code entfallen. Der Code wird dadurch kürzer und übersichtlicher.

Pakete werden in einen Header- und einen Body-Teil aufgespalten. Der Header enthält die „von außen“ sichtbaren Deklarationen, zum Beispiel welche Aufrufparameter eine Prozedur besitzt. Der Package-Body legt die Implementierung der im Header deklarierten Elemente fest.

Der Package-Header wird mit dem Schlüsselwort *package* eingeleitet, während ein Package-Body durch *package body* gekennzeichnet wird:

```
package <Paketname> is
   <Typdefinitionen>
   <Definition oder Deklaration von Konstanten>
   <Signaldefinitionen>
   <Deklaration von Funktionen und Prozeduren>
   <Component-Anweisungen>
end package;

package body <Paketname> is
   <Definition von Konstanten, falls im Header nur deklariert>
   <Definitionen von Funktionen und Prozeduren>
end package body;
```

Als ein Beispiel für die Anwendung von Paketen zeigt der nachfolgende Code ein Paket, das Funktionen zur Umwandlung des Gray-Codes in Dualzahlen und umgekehrt enthält.

```
library ieee;
use ieee.std_logic_1164.all;

------
-- Package Header
------
package gray_pkg is

   -- Funktionsdeklarationen --
   function gray2bin (gray_val : std_logic_vector)
        return std_logic_vector;
```

```
   function bin2gray (bin_val : std_logic_vector)
        return std_logic_vector;

end package;

------
-- Package Body
------
package body gray_pkg is
   -- Implementierung: Gray2Bin --
   function gray2bin (gray_val : std_logic_vector)
        return std_logic_vector is
        constant vlen : integer := gray_val''length;
        variable temp : std_logic_vector(vlen-1 downto 0);
   begin
        temp := gray_val;
        if vlen > 1 then
            for i in vlen-2 downto 0 loop
                 temp(i) := gray_val(i) xor temp(i+1);
            end loop;
        end if;
        return temp(vlen-1 downto 0);
   end function;

   -- Implementierung: Bin2Gray --
   function bin2gray (bin_val : std_logic_vector)
        return std_logic_vector is
        constant vlen : integer := bin_val'length;
   begin
        return ('0' & bin_val(vlen-1 downto 1)) xor bin_val;
   end function;

end package body;
```

8.2.8 Einbindung von Spezialkomponenten

Für FPGAs und ASICs sind Spezialkomponenten wie Multiplizierer, Speicher oder Elemente zur Taktaufbereitung verfügbar. Doch wie können diese Elemente in einem VHDL-basierten Design eingesetzt werden? Hierzu werden zwei Ansätze unterschieden: Die Instanziierung und die Inferenz (engl. *instantiation* beziehungsweise *inference*). Beide Ansätze werden im Folgenden näher erläutert.

Instanziierung beim FPGA-Entwurf
Bei der Instanziierung wird ein bestimmtes Modul, zum Beispiel ein Multiplizierer, explizit als eine Komponente aufgerufen. Damit wird dem Synthesetool vorgeschrieben dieses konkret benannte Modul zu verwenden.

Für die Instanziierung stellen die FPGA-Hersteller spezielle VHDL-Bibliotheken zur Verfügung, in denen alle Grundelemente hinterlegt sind. Man kann also auf die verfügbaren Hardwarekomponenten explizit zugreifen. Theoretisch könnten auch einzelne Logikzellen ausgewählt und durch den Designer verdrahtet werden. Da man hiermit aber die Intelligenz der Synthesetools nicht nutzen würde, wird von dieser Möglichkeit in der Praxis kein Gebrauch gemacht. Die Instanziierung wird im Allgemeinen nur dort eingesetzt, wo dies unumgänglich ist, weil die gewünschten Elemente nicht automatisch durch die Synthese ausgewählt werden können. Ein Beispiel hierfür sind PLLs zur Taktaufbereitung. Für diese Elemente existiert keine Entsprechung in VHDL und daher müssen sie per Instanziierung ausgewählt werden.

Die Parameter der jeweiligen Instanz werden im VHDL-Code durch Übergabe von Generics festgelegt. Da dies in einigen Fällen etwas umständlich ist, werden grafische Blockgeneratoren angeboten. Mithilfe der Generatoren ist es möglich, die Eigenschaften des zu instanziierenden Blocks interaktiv über eine grafische Oberfläche festzulegen. Als Ergebnis liefern die Generatoren einen Block, der in einer VHDL-Beschreibung als Komponente instanziiert werden kann.

Inferenz beim FPGA-Entwurf
In einigen Fällen kann man auch auf die „Intelligenz“ des Synthesetools setzen: Für bestimmte VHDL-Konstrukte erkennt die Synthese automatisch, dass hier ein Hardmakro (zum Beispiel ein Multiplizierer-Modul oder ein FPGA-interner Speicher) in Betracht kommt. Da sich die Verwendung der Makros aus dem VHDL-Code ergibt, wird dieses Vorgehen als Inferenz bezeichnet.

Die Syntheseprogramme unterstützen meist die Inferenz von Speichern, Multiplizieren und einfachen arithmetischen Komponenten wie zum Beispiel die in der Signalverarbeitung häufig vorkommende Kombination eines Multiplizierers mit einem nachfolgenden Addierer. Für die Inferenz eines Multiplizierers genügt es beispielsweise, die entsprechende Operation im VHDL-Code zu verwenden.

Die Instanziierung und Inferenz wird im Folgenden anhand des Beispiels eines FPGA-internen Speichers für einen FPGA-Baustein der Xilinx Serie 7 näher beleuchtet.

8.2.8.1 Beispiel: Instanziierung eines Speichers

Der nachfolgend dargestellte VHDL-Code zeigt die Instanziierung eines Speichers. Es wird das Modul BRAM_SDP_MACRO, welches in der von der Firma Xilinx zur Verfügung gestellten Bibliothek *unisim* vorliegt, aufgerufen und mit den Signalen des Designs verbunden. Über Generics lassen sich verschiedene Parameter, wie die Wortbreite oder die Größe des Speichers, auswählen.

```
library unisim;
use unisim.vcomponents.all;
library unimacro;
use unimacro.vcomponents.all;
…
my_ram_instance : bram_sdp_macro
generic map (
   bram_size    => "18Kb",     -- Auswahl Speichergroesse: "18Kb", "36Kb"
   device       => "7SERIES", -- Zielbaustein-Serie
   write_width => 8,          -- Wortbreite Schreibport
   read_width  => 8,          -- Wortbreite Leseport
   do_reg       => 0,          -- Zusaetzliches Register am Daten-Ausgang?
   init_file    => "NONE",     -- evtl. Datei mit Initialwerten
   sim_collision_check => "NONE", -- Simulation: Schreib/Leseoperation
                                  -- auf gleiche Adresse checken?
   srval        => x"0000000000000000", -- Ausgabe nach Reset
   write_mode  => "WRITE_FIRST"   -- Auswahl Kollisionsbehandlung
   )
port map (
   rst    => rst,    -- Reseteingang
   rdclk  => rdclk,  -- Taktsignal Leseport
   rdaddr => rdaddr, -- Leseadresse
   rden   => rden,   -- Enable: Lesen
   regce  => '1',    -- Enable für Ausgangsregister
   do     => do,     -- Lesedaten
   wrclk  => wrclk,  -- Taktsignal Schreibport
   wraddr => wraddr, -- Schreibadresse
   wren   => wren,   -- Enable-Signal für Schreiboperation
   we     => we,     -- Byte-weises Enable-Signal
   di     => di      -- Schreibdaten
);
```

Ein Nachteil der Instanziierung ist, dass man unter anderem die Größe der Speichermodule auf dem FPGA kennen muss. Wird ein Speicher benötigt, der größer als ein einzelner Speicherblock ist, muss die entsprechende Anzahl an Speichermodulen instanziiert werden. Darüber hinaus lässt sich VHDL-Code, der die Instanziierung von Elementen verwendet, nicht unbedingt auf andere FPGAs übertragen. So könnten sich zum Beispiel die Eigenschaften der Speichermodule einer nachfolgenden FPGA-Generation ändern. Der VHDL-Code wäre damit nicht mehr zu dem neuen FPGA kompatibel und müsste entsprechend angepasst werden.

8.2.8.2 Beispiel: Instanziierung eines Speichers mit Blockgenerator

Alternativ stellen die FPGA-Hersteller Modul-Generatoren zur Verfügung um Speicher-Module über eine grafische Oberfläche zu konfigurieren. Der Blockgenerator erstellt

dann eine Komponente, die im VHDL-Code eingebunden werden kann. Der Vorteil dabei ist, dass der Blockgenerator auch größere Speicher aus mehreren Speicherblöcken zusammenstellen kann. Falls zusätzliche kombinatorische Logik erforderlich ist, wird auch diese erzeugt.

Im unten stehenden Beispiel wird ein FIFO-Speicher aufgerufen, der Datenworte um eine feste Anzahl an Takten verzögert. FIFO steht dabei für First-In-First-Out. Der Blockgenerator erzeugt die VHDL-Dateien des Moduls *fifo_memory*. Neben Speicher-Modulen können Generatoren auch andere Funktionen erzeugen, beispielsweise Divisionsschaltungen oder Filter.

```
my_fifo_instance : fifo_memory
port map (
   clk   => clk,
   d_in  => d_in,
   d_out => d_out);
```

Wie bei der Instanziierung von Modulen aus der FPGA-Bibliothek kann ein Untermodul nicht unbedingt auf andere FPGAs übertragen werden.

Dieser Nachteil lässt sich durch die Inferenz von Speichern umgehen. Hierzu muss der VHDL-Code so geschrieben werden, dass er den Eigenschaften des Speichers entspricht.

8.2.8.3 Beispiel: Inferenz eines Speichers

Der nachfolgende Code zeigt die Realisierung eines Speichers. Die Wortbreite und die Größe des Speichers kann über Generics ausgewählt werden. Da der Lesezugriff synchron implementiert ist, wählen die Syntheseprogramme die auf dem FPGA-Baustein verfügbaren RAM-Speicherelemente (sogenanntes *Block-RAM*) aus.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity dmem_sp is
   generic (
        DW   : integer := 16;      -- Data Width
        AW   : integer := 10);     -- Address Width

     port (
         clk : in  std_logic;      -- Clock
         en  : in  std_logic;      -- Enable
         we  : in  std_logic;      -- Write enable
         a   : in  std_logic_vector(AW-1 downto 0);   -- Address
         d   : in  std_logic_vector(DW-1 downto 0);   -- Data in
```

```
        q   : out std_logic_vector(DW-1 downto 0)); -- Data out
end;

architecture rtl of dmem_sp is
   type tmem is array(0 to 2**AW-1) of std_logic_vector(DW-1 downto 0);
   signal mem : tmem;
begin
   process begin
        wait until rising_edge(clk);
        q <= mem(to_integer(unsigned(a)));
        if en = '1' then
            if we = '1' then
                mem(to_integer(unsigned(a))) <= d;
            end if;
        end if;
   end process;
end;
```

Da keine Aussagen über die FPGA-Technologie im Code vorgenommen werden, ist die Speicherinferenz auch auf andere FPGAs übertragbar. Darüber hinaus kann die Speichergröße und Wortbreite flexibel über die Generics angegeben werden, ohne eine genauere Kenntnis der zugrunde liegenden FPGA-Technologie zu haben.

Möchte man dagegen statt der Block-RAM-Module lieber Flip-Flops als Speicher verwenden, ist nur eine kleine Änderung des Codes erforderlich. Zieht man die Zuweisung an den Datenausgang *q* vor den Prozess, wird ein asynchroner Lesezugriff beschrieben. Mit einer derartigen VHDL-Beschreibung werden dann Flip-Flops als Speicherelemente (sogenanntes *Distributed Memory*) ausgewählt. Dies kann zum Beispiel vorteilhaft sein, wenn nur ein sehr kleiner Speicher benötigt wird: Block-RAMs stehen meist nur in Vielfachen von 1 oder 2 kByte zur Verfügung. Benötigt man zum Beispiel nur 256 Bit Speicherplatz und sind die Block-RAM-Ressourcen knapp, ist der Einsatz von Distributed Memory erwägenswert.

Die entsprechenden Änderungen für die Verwendung von Distributed Memory sind im folgenden Code-Ausschnitt dargestellt.

```
begin
   q <= mem(to_integer(unsigned(a))); -- Asynchroner Lese-Zugriff
   process begin
        wait until rising_edge(clk);
        if en = '1' then
        …
```

In der Regel sollte die Inferenz bevorzugt werden, da diese übersichtlicher ist und sich der Code leichter auf andere FPGAs übertragen lässt. Für einige Module, beispielsweise PLLs, hat man nicht die Wahl zwischen Instanziierung und Inferenz.

Diese Spezialmodule müssen entweder durch eine VHDL-Instanziierung oder durch einen Blockgenerator im System eingebaut werden. Die näheren Einzelheiten über die zu verwendenden Bibliothek oder den Aufruf des entsprechenden Moduls in VHDL ist bei Bedarf in der Dokumentation der Anbieter der Synthesetools zu finden.

8.2.8.4 Beispiel: Inferenz eines Dual-Port-Speichers

FPGAs stellen meist auch sogenannte Dual-Port-Speicher zur Verfügung. Hierbei handelt es sich um Speicher, die zwei getrennte Anschlüsse für Lese- und Schreibzugriffe besitzen. Es kann also gleichzeitig von zwei unterschiedlichen Modulen auf die Elemente des Speichers zugegriffen werden.

Dual-Port-Speicher erlauben es, beide Module mit unterschiedlichen Taktfrequenzen zu betrieben. In diesem Fall muss die Inferenz des Dual-Port-Speichers mithilfe zweier getrennter VHDL-Prozesse (ein Prozess für jeden der beiden Schreib-Lese-Ports) beschrieben werden.

Da beide Prozesse auch einen Schreibzugriff auf die Speicherelemente unterstützen müssen, ergibt sich hier eine Besonderheit: Das Speicher-Array kann nicht durch eine Variable innerhalb **einer** der beiden Prozesse realisiert werden, da dann der andere Prozess keinen Zugriff auf die Variable hätte. Aber auch die Realisierung mithilfe eines VHDL-Signals ist nicht möglich: Beide Prozesse würden schreibend auf das Array-Signal zugreifen, was während der Synthese zu Fehlermeldungen führen würde.

Um diese Problematik zu lösen, können Variablen eingesetzt werden, die (wie Signale) im Deklarationsteil der Architecture definiert werden und in allen Prozessen der Architecture sichtbar sind. Diese Art der Variablen wird in VHDL als *Shared Variables* bezeichnet. Die Beschreibung eines synchronen Dual-Port-Speichers kann wie folgt realisiert werden:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity bmem_dp is
   generic (
        DW  : integer := 16;   -- Data Width
        AW  : integer := 10); -- Address Width

   port (
        -- Port 1
        clk1 : in  std_logic;  -- Clock
        we1  : in  std_logic;  -- Write enable
        a1   : in  std_logic_vector(AW-1 downto 0); -- Address
        d1   : in  std_logic_vector(DW-1 downto 0); -- Data in
        q1   : out std_logic_vector(DW-1 downto 0); -- Data out
```

```
        -- Port 2
        clk2 : in  std_logic; -- Clock
        we2  : in  std_logic; -- Write enable
        a2   : in  std_logic_vector(AW-1 downto 0); -- Address
        d2   : in  std_logic_vector(DW-1 downto 0); -- Data in
        q2   : out std_logic_vector(DW-1 downto 0)  -- Data out
        );
end;

architecture rtl of bmem_dp is
   type tmem is array(0 to 2**AW-1) of std_logic_vector(DW-1  downto  0);
   -- Hier wird die "shared variable" definiert
   shared variable mem : tmem := ((others=> (others=>'0')));
   signal q1_sig : std_logic_vector(DW-1 downto 0) := (others=>'0');
   signal q2_sig : std_logic_vector(DW-1 downto 0) := (others=>'0');

begin
   q1 <= q1_sig;
   q2 <= q2_sig;

   -- Port 1
   process begin
        wait until rising_edge(clk1);
        if (we1 = '1') then
            mem(to_integer(unsigned(a1))) := d1;
        end if;
        q1_sig <= mem(to_integer(unsigned(a1)));
   end process;

   -- Port 2
   process begin
        wait until rising_edge(clk2);
        if (we2 = '1') then
            mem(to_integer(unsigned(a2))) := d2;
        end if;
        q2_sig <= mem(to_integer(unsigned(a2)));
   end process;
end;
```

Natürlich kann dieser Code auch eingesetzt werden, wenn die beiden Module, die auf den Speicher zugreifen, identische Taktsignale verwenden. In diesem Fall wird an die Taktanschlüsse *clk1* und *clk2* einfach das gleiche Taktsignal angelegt.

Achtung: Lassen Sie sich nicht dazu verleiten, *Shared Variables* als Ersatz für VHDL-Signale einzusetzen. *Shared Variables* können zwar von typischen Synthesetools – mit entsprechenden Warnmeldungen – in Hardware übersetzt werden, aber das Verhalten von

Schreibzugriffen aus zwei Prozessen heraus ist für *Shared Variables* nicht eindeutig definiert. Im obigen Fall der Beschreibung eines Speichermoduls ist dies akzeptabel und wird vom Synthesetool korrekt in einen entsprechenden Dual-Port-Speicher überführt. In den meisten anderen Fällen kann die Verwendung von *Shared Variables* zu Unterschieden zwischen Simulation und synthetisierter Hardware führen.

8.3 Sprachelemente zur Verifikation

Wie bereits in Kapitel 7 beschrieben, ist die Simulation mit einer Testbench ein wesentlicher Schritt zur Verifikation von VHDL-Code. VHDL bietet dabei die Möglichkeit während der Simulation auf Dateien zuzugreifen. Dieses kann zum Beispiel sinnvoll sein, um Ausgabewerte oder Statusmeldungen während der Simulation in einer Datei abzulegen, die anschließend auch ohne erneuten Simulationsaufruf zur Verfügung stehen.

Grundsätzlich ist die binäre Ein-/Ausgabe und die Ein-/Ausgabe von Textdateien zu unterscheiden. Binäre Dateien enthalten die gespeicherten Werte in binärer Form während die gespeicherten Werte in Textdateien im ASCII-Code vorliegen und mithilfe eines Editors betrachtet und modifiziert werden können.

8.3.1 Binäre Ein-/Ausgabe

Um auf eine Datei zugreifen zu können, muss in VHDL zunächst ein Dateidatentyp angelegt werden. Dies erfolgt mithilfe der Definition eines benutzerdefinierten Datentyps. Anschließend wird mithilfe dieses Datentyps ein sogenannter *Dateideskriptor* angelegt, welcher für alle weiteren Zugriffe auf die verwendet wird. Das nachfolgende Beispiel zeigt die erforderlichen Definitionen für eine Datei, die mit dem Datentyp *integer* arbeitet.

```
type my_file_type is file of integer;
file my_file : my_file_type;
```

Das eigentliche Öffnen der Datei erfolgt anschließend mithilfe der Prozedur *file_open()*. Diese Prozedur erwartet vier Parameter. Der erste Parameter ist vom Datentyp *FILE_OPEN_STATUS*. Ihm wird der Status nach dem Öffnen der Datei zugewiesen. War das Öffnen der Datei erfolgreich, erhält der Parameter den Wert *OPEN_OK*. Für eventuelle Fehlerfälle stehen die Werte *STATUS_ERROR, NAME_ERROR, MODE_ERROR* zur Verfügung. Der zweite Parameter ist vom Datentyp *FILE*. Hier wird der zuvor definierte Dateidatentyp übergeben. Der Dateiname wird als dritter Parameter angeben. Ob die Datei zum Lesen oder Schreiben geöffnet wird, legt der vierte Parameter fest: Mit *READ_MODE* wird eine Datei zum Lesen geöffnet, während *WRITE_MODE* eine zu schreibende Datei öffnet. Sollen Daten an den Inhalt einer bestehenden Datei angehängt werden, wird als vierter Parameter *APPEND_MODE* verwendet.

Ein mögliches Beispiel für das Öffnen einer Datei zeigt der nachfolgende Codeausschnitt:

```
file_open(my_file_status, my_file, "my_values.dat", WRITE_MODE);
```

Für die Ein-/Ausgabe stellt VHDL die Prozeduren *read()* und *write()* zur Verfügung. Als Parameter werden der Dateideskriptor und eine Variable übergeben, die den auszugebenden Wert enthält *(write)* oder welcher der eingelesene Wert zugewiesen wird *(read).*

Ein Beispiel wie in einem Prozess eine binäre Datei geöffnet und der Schreibzugriff realisiert wird, zeigt der nachfolgende Code:

```
process
   type my_file_type is file of integer;
   file my_file : my_file_type;
   variable cnt : integer:= 64;
   variable my_file_status : FILE_OPEN_STATUS;
begin
   -- Datei öffnen
   file_open(my_file_status, my_file, "my_values.dat", WRITE_MODE);
   if my_file_status = OPEN_OK then -- Datei erfolgreich geöffnet?
        for i in 1 to 10 loop
            write(my_file, cnt); -- Werte in die Datei schreiben
            cnt := cnt+1;
        end loop;
        file_close(my_file); -- Datei schließen
   end if;
   wait; -- Diesen Prozess mit einfacher Wait-Anweisung beenden
end process;
```

8.3.2 Ein-/Ausgabe mit Textdateien

Während für die binäre Ein-/Ausgabe keine besonderen Pakete benötigt werden, muss für den Zugriff auf Textdateien das standardisierte Paket *textio,* welches ein Teil der Standardbibliothek *std* ist, mithilfe einer Use-Anweisung bekannt gemacht werden. Dieses Paket umfasst die textuelle Ein-/Ausgabe für die im VHDL-Standard definierten Datentypen. Sollen Daten vom Typ *std_logic* eingelesen oder ausgegeben werden, steht das zusätzliche Paket *std_logic_textio* aus der Bibliothek *ieee* zur Verfügung.

Die textuelle Ein-/Ausgabe erfolgt zeilenbasiert. So wird bei der Ausgabe zunächst eine Textzeile (vom Datentyp *line*) mit der Write-Prozedur beschrieben. Ist eine Textzeile erstellt, kann diese mit der Prozedur *writeline()* ausgeben werden. Entsprechendes gilt für die Eingabe: Zunächst wird eine Zeile mit der Prozedur *readline()* eingelesen und anschließend mithilfe der Read-Prozedur auf den Inhalt der Zeile zugegriffen.

Eine Besonderheit ist zu beachten, wenn Zeichenketten (strings) ausgegeben werden sollen. Die folgenden Zeilen würden zu einer Fehlermeldung führen:

```
write (my_line, "Hallo"); -- Fehler! Ist dies wirklich eine Zeichenkette?
write (my_line, "10010");-- Auch falsch! String oder std_logic_vector?
                         -- oder etwas anderes ???
```

Bei der ersten Zeile ist es für einen Menschen sofort offensichtlich, dass es sich um eine Zeichenkette vom Datentyp *string* handelt. Bei der zweiten Zeile ist dies weniger offensichtlich. Schließlich könnte es sich beispielsweise auch um einen Wert vom Typ *std_logic_vector* handeln. Damit nun die korrekte Implementierung der *Write*-Prozedur aufgerufen werden kann, muss der Datentyp in diesem Fall explizit angeben werden. Dies gilt auch für die eigentlich für einen Menschen offensichtlichen Fälle. Die explizite Kennzeichnung des Datentyps erfolgt über einen sogenannten *Type-Qualifier,* dessen allgemeine Form wie folgt aussieht:

```
<Datentyp>'(<Wert>)
```

Für die obigen Beispiele würde der korrekte Code also wie folgt lauten:

```
write (my_line, string'("Hallo")); -- Ok! Mit expliziter Typangabe …
write (my_line, string'("10010"));   -- … kann die richtige write-Funktion
                                     -- identifiziert werden
```

Ein Beispiel zur Verwendung der Textausgabe zeigt der nachfolgende Prozess.

```
process
   -- Für die Angabe des Dateityps kann der im textio-Paket definierte
   -- Datentyp text verwendet werden
   file my_txt_file : text;

   variable cnt     : integer:= 64;
   variable cnt_slv : std_logic_vector (7 downto 0);
   variable l       : line;
   variable my_file_status : FILE_OPEN_STATUS;
begin
   -- Datei öffnen
   file_open(my_file_status, my_txt_file, "my_values.txt", WRITE_MODE);
   if my_file_status = OPEN_OK then -- Datei erfolgreich geöffnet?
        for i in 1 to 5 loop
            write(l, cnt); -- Integer in die Datei schreiben
            write(l,string'(" "));
            cnt_slv := std_logic_vector(to_unsigned(cnt,8));
            write(l,cnt_slv); -- Wert als std_logic_vector schreiben
```

```
            writeline(my_txt_file,l);
            cnt := cnt+1;
        end loop;
        file_close(my_txt_file); -- Datei schließen
    end if;
    wait; -- Prozess beenden
end process;
```

Die Simulation initialisiert die Variable *cnt* mit dem Wert 64. In einer Schleife wird *cnt* als Integer und *std_logic_vector* fünfmal ausgegeben und dabei jeweils um 1 erhöht. Nach Durchführung der Simulation würde die Datei *my_values.txt* den folgenden Inhalt besitzen:

```
64 01000000
65 01000001
66 01000010
67 01000011
68 01000100
```

Beim Einlesen von Dateien kommen den Funktionen *endfile()* und *endline()* eine wichtige Bedeutung zu. Ihnen wird als Parameter ein Dateideskriptor oder eine Zeile übergeben. Wenn der Rückgabewert (Typ: *boolean*) der Funktion den Wert *true* besitzt, wurde das Ende der Datei beziehungsweise der Zeile erreicht.

Mithilfe der vorgestellten Ein-/Ausgabekonzepte können auch Ein- und Ausgaben auf der Simulatorkonsole erfolgen. Hierfür sind die Symbole *INPUT* und *OUTPUT* vordefiniert:

```
write(l,string'("Hallo Konsole!"));
writeline(OUTPUT,l);
```

8.3.3 Wait-Anweisungen in Testbenches

In den vorangegangenen Kapiteln wurde die Wait-Anweisung bereits eingeführt. Die Wait-Anweisung wurde verwendet, um sequenzielle Schaltungen vom einfachen D-Flip-Flop bis hin zu komplexeren endlichen Automaten zu beschreiben. Zur Erinnerung ist hier noch einmal die VHDL-Beschreibung eines Prozesses angegeben, der die Funktion eines D-Flip-Flops realisiert:

```
process begin
    wait until rising_edge(clk);
    q <= d;
end process;
```

In diesem Beispiel wird die Ausführung unterbrochen bis eine bestimmte Bedingung, hier das Auftreten einer steigenden Flanke des Taktsignals *clk,* wahr ist. Für synthetisierbaren VHDL-Code ist diese Form der Wait-Anweisung ist die am häufigsten verwendete Variante. Es gibt jedoch noch weitere Varianten der Wait-Anweisung, die insbesondere für die Erstellung von Testbenches nützlich sind. Die vier Varianten der Wait-Anweisung sind in Tab. 8.1 zusammengefasst.

Es ist zu beachten, dass Wait-Anweisungen und Sensitivitätslisten einander ausschließen. Besitzt ein Prozess eine Sensitivitätsliste, darf er keine Wait-Anweisung enthalten. Wird dagegen eine Wait-Anweisung verwendet, darf der Prozess keine Sensitivitätsliste besitzen. Darüber hinaus darf synthetisierbarer Code nur eine einzelne Wait-until-Anweisung pro Prozess enthalten. Testbench-Prozesse, die dagegen nur für die Simulation verwendet werden, dürfen beliebig viele Wait-Anweisungen enthalten. Mithilfe der Wait-Anweisung kann eine Testbench auf recht einfache Weise erstellt werden. Der nachfolgende Abschnitt zeigt hierzu ein Beispiel.

8.3.4 Testbench mit interaktiver Überprüfung

Eine Testbench besitzt keine Eingangs- oder Ausgangssignale. Daher kann die Entity sehr einfach realisiert werden. Sie besteht im Allgemeinen aus zwei Zeilen:

```
entity tb is
end;
```

Im Deklarationsteil der Architecture werden die Signale definiert, die an die Ein- und Ausgänge des zu überprüfenden Moduls angeschlossen werden. Im Anweisungsteil der Architecture wird der Prüfling instanziiert und es werden mithilfe eines Prozesses unterschiedliche Testvektoren an die zu testende Komponente angelegt.

Die Architecture einer Testbench für einen Encoder, welcher einen 4-Bit-Binärwert in ein 7-Bit-Codewort für eine Sieben-Segment-Anzeige umsetzt, kann wie folgt realisiert werden:

Tab. 8.1 Formen der Wait-Anweisung

Struktur	Beispiel	Erläuterung
wait;	wait;	„Für immer warten“: Der Prozess wird unterbrochen und nie fortgesetzt
wait for <Zeitangabe>;	wait for 10 ns;	Prozessunterbrechung für einen bestimmten Zeitraum
wait on <Signalliste>;	wait on A, B;	Prozessunterbrechung bis ein Wechsel eines Signals der Signalliste detektiert wird
wait until <Bedingung>;	wait until A = B;	Unterbrechung des Prozesses bis die angegebene Bedingung wahr ist

```
architecture tb_arch of tb is
   signal bin_val : std_logic_vector(3 downto 0);
   signal sev_seg_code : std_logic_vector(6 downto 0);
begin
   dut : entity work.bin2sevenseg -- DUT: Device Under Test
   port map (
        bin      => bin_val,
        sevenseg => sev_seg_code);

   process begin -- Prozess zum Anlegen der Stimuli
        bin_val <= "0000";
        wait for 10 ns; -- Kurze Wartezeit
        bin_val <= "0001";
        wait for 10 ns;
        bin_val <= "0010";
        wait for 10 ns;
        -- Hier ggf. weitere Stimuli
        wait; -- Test durchlaufen. Der Prozess kann beendet werden.
   end process;
end;
```

Da die Testbench keine Überprüfung der Ausgabewerte des Prüflings vornimmt, muss die Korrektheit durch eine manuelle Überprüfung der erzeugten Waveform erfolgen. Dieses Vorgehen besitzt den Vorteil, dass der Testbench-Code auf die Erzeugung von Stimuli beschränkt bleibt und daher relativ einfach zu realisieren ist. Ein Nachteil ist, dass bei der Überprüfung ein mögliches Fehlverhalten des zu testenden Moduls übersehen werden könnte.

8.3.5 Testbench mit Assert-Anweisungen

Sind die erwarteten Ausgabewerte des Prüflings bekannt, kann die Verifikation im Rahmen auch durch die Testbench selbst erfolgen. Hierzu kann die Assert-Anweisung eingesetzt werden. Diese Anweisung überprüft während der Simulation eine angegebene Bedingung. Ist diese nicht erfüllt, wird eine Meldung ausgegeben. Der Schweregrad der Verletzung der angegebenen Bedingung kann explizit angegeben werden. Zur Auswahl stehen hierbei *note, warning, error* und *failure.* Welcher Schweregrad zu einem Abbruch der Simulation führt, kann mithilfe der Aufrufparameter des Simulators ausgewählt werden. Erfolgt keine Auswahl, führen in der Regel die Schweregrade *error* und *failure* zu einem Abbruch der Simulation.

Die folgenden Beispiele zeigen den typischen Aufbau von Assert-Anweisungen:

```
-- Signal a wird gegen einen erwarteten Wert a_exp getestet
assert a /= a_exp report "Fehler in der Simulation" severity error;

-- Eine Warnung ausgeben falls der Wert von i 10 überschreitet
assert i <= 10 report "i ist groesser als 10" severity warning;

-- Eine Simulation mit Hilfe der Assert-Anweisung beenden
assert false report "Simulation wird beendet" severity failure;
```

Die Verwendung der Assert-Anweisung für die Verifikation eines UND-Gatters zeigt der folgende Code. Die erwarteten Ausgabewerte werden in der Variablen *q_expected* abgelegt und mit den Ausgabewerten des Prüflings verglichen. Die Variable *q_expected* beschreibt, dass die erwartete Ausgabe für die Eingangswerte 00, 01 und 10 jeweils 0 ist. Nur für die Eingabe 11 wird am Ausgang des UND-Gatters eine 1 erwartet. Tritt ein Fehler auf, wird mithilfe einer Assert-Anweisung eine entsprechende Meldung ausgegeben.

```
process
   variable i_sv       : std_logic_vector (1 downto 0);
   variable q_expected : std_logic_vector (3 downto 0) := "1000";
begin
   for i in 0 to 3 loop
        i_sv := std_logic_vector(to_unsigned(i,2));
        a <= i_sv(0);
        b <= i_sv(1);
        wait for 10 ns;
        assert q = q_expected(i) report "Fehler!" severity error;
   end loop;
   wait;
end process;
```

Die Anwendung der Assert-Anweisung ist nicht auf Testbench-Code beschränkt. Auch in synthetisierbaren VHDL-Beschreibungen können Assert-Anweisungen eingesetzt werden, um beispielsweise das Einhalten eines erwarteten Wertebereichs zu überprüfen. Bei der Synthese der VHDL-Beschreibung wird aus den Assert-Anweisungen keine Hardware generiert. Sie werden vom Syntheseprogramm ignoriert.

8.3.6 Testbench mit Dateiein-/-ausgabe

Häufig entsteht bei dem Entwurf eines digitalen Systems der Wunsch Stimuli oder erwartete Ausgabewerte aus Dateien einzulesen oder Ausgaben der Simulation in einer Datei abzulegen. Dieses Vorgehen hat verschiedene Vorteile:

- Die Stimuliwerte sind übersichtlich in einer Datei zusammengefasst und können leicht geändert werden.

- Stimuli- und Erwartungswerte können rechnergestützt erstellt werden. Dies ist insbesondere dann interessant, wenn ein funktionales Modell des zu entwerfenden Systems in einer Hochsprache (meist C/C++) erstellt wurde.
- Simulationen benötigen keine interaktiven Eingriffe.
- Die Simulationsergebnisse können rechnergestützt ausgewertet werden.
- Stimuli und Resultate einer Simulation liegen in einfach lesbarer Form vor und können zu Dokumentationszwecken aufbewahrt werden.

Diesen Vorteilen steht gegenüber, dass der Aufwand zum Erstellen einer Testbench größer ist als bei den zuvor skizzierten Ansätzen. In vielen Fällen kann der zusätzliche Aufwand gering gehalten werden, wenn eine bereits zuvor eingerichtete Testbench wiederverwendet werden kann und nur leicht abgewandelt werden muss.

Der nachfolgende Code stellt eine komplette Testbench mit Dateiein-/-ausgabe für ein einfaches logisches Gatter dar. Der Code lässt sich auch auf komplexere Problemstellungen erweitern.

```
use std.textio.all; -- bei Benutzung der Standard-Bibliothek
                    -- ist keine Library-Anweisung erforderlich
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.std_logic_textio.all;

entity tb is
end;

architecture tb_arch of tb is
   signal bin_val : std_logic_vector(3 downto 0);
   signal sev_seg_code : std_logic_vector(6 downto 0);
begin
   dut : entity work.bin2sevenseg -- DUT: Device Under Test
        port map (
            bin      => bin_val,
            sevenseg => sev_seg_code);

process -- Prozess zum Anlegen von Stimuli und zum Ueberprufen
        -- der Ausgabewerte des „device under test (DUT)"

   file stimuli_file         : text; -- Filedeskriptoren anlegen
   file resultat_file        : text;
   variable stim_file_status : FILE_OPEN_STATUS; -- Filestatus
   variable res_file_status  : FILE_OPEN_STATUS;
   variable l                : line; -- Variable vom Typ line fuer
Text-IO
```

```
   variable stim             : std_logic_vector(3 downto 0);
   variable exp              : std_logic_vector(6 downto 0);
   variable wait_time        : time;
   variable errors_detected  : natural := 0;
begin

   -- Dateien öffnen
   file_open(stim_file_status, stimuli_file, "stimuli.txt", READ_MODE);
   file_open(res_file_status, resultat_file, "result.txt", WRITE_MODE);

   -- Dateien erfolgreich geöffnet?
   if stim_file_status = OPEN_OK and res_file_status = OPEN_OK then
        while not endfile(stimuli_file) loop -- Dateiende?
            readline(stimuli_file,l); -- Eine Zeile lesen
            read(l,stim); -- Stimuli lesen
            bin_val <= stim;
            read(l,wait_time); -- Wartezeit lesen
            wait for wait_time; -- Warten
            read(l,exp); -- Erwarteten Ausgabewert lesen
            write (l,stim); -- Stimuli und Ausgabewerte
            write (l,string'(" ")); -- in Resultat-Datei schreiben
            write (l,sev_seg_code);
            write (l,string'(" "));
            assert sev_seg_code = exp
                   report "Simulation error detected" severity warning;
            if sev_seg_code = exp then -- in Resultat-Datei schreiben
                 write (l,string'("Ok"));
            else
                 write (l,string'("Error -- Expected: "));
                 write (l,exp);
                 errors_detected := errors_detected + 1; -- Fehlerzaehler+1
            end if;
            writeline(resultat_file,l);
        end loop;
        -- Am Ende der Simulation den Fehlerzaehler ausgeben
        write (l,string'("--------"));
        writeline(resultat_file,l);
        write (l,string'("Total Error Count: "));
        write (l,errors_detected);
        writeline(resultat_file,l);
        write (l,string'("--------"));
        writeline(resultat_file,l);
        file_close(stimuli_file); -- Dateien schliessen
        file_close(resultat_file);
   end if;
```

```
   -- Simulation mit Assert-Anweisung beenden
   assert false report "Simulation finished." severity failure;
end process;
end;
```

Die Stimulidatei *stimuli.txt* besitzt ein recht übersichtliches zeilenorientiertes Format. In einer Zeile stehen zunächst die Stimuliwerte. Daran schließt sich die Angabe der Zeit an, die zwischen Anlegen der Stimuliwerte und Auswertung der Ausgangswerte vergehen soll. Am Ende der Zeile ist der erwartete Ausgabewert des Prüflings angegeben.

```
0000 10 ns 0111111
0001 10 ns 0000110
0010 10 ns 1011011
0011 10 ns 1001111
0100 10 ns 1100110
0101 10 ns 1101101
0110 10 ns 1111101
0111 10 ns 0000111
1000 10 ns 1111111
1001 10 ns 1101111
1010 10 ns 1110111
1011 10 ns 1111100
1100 10 ns 0111001
1101 10 ns 1011110
1110 10 ns 1111001
1111 10 ns 1110001
```

Die durch die Simulation erzeugte Ergebnisdatei sieht beispielsweise wie folgt aus:

```
0000 0111111 Ok
0001 0000110 Ok
0010 1011011 Ok
0011 1001111 Ok
0100 1100110 Ok
0101 1101101 Ok
0110 1111101 Ok
0111 0000101 Error -- Expected: 0000111
1000 1111111 Ok
1001 1101111 Ok
1010 1110111 Ok
1011 1111100 Ok
1100 0111001 Ok
1101 1011110 Ok
```

```
1110 0000001 Error -- Expected: 1111001
1111 1110001 Ok
--------
Total Error Count: 2
--------
```

8.4 Übungsaufgaben

Haben Sie den Inhalt des Kapitels verstanden? Prüfen Sie sich selbst mit den folgenden Aufgaben. Am Ende des Buches finden Sie die Lösungen.

Sofern nicht anders vermerkt, ist nur eine Antwort richtig.

Aufgabe 8.1
Ein Taktsignal soll mithilfe des VHDL-Signals *clk* modelliert werden. Die Frequenz des Taktsignals beträgt 100 MHz. Welche der folgenden Codezeilen ist korrekt?

a) clk <= not clk;
b) clk <= not clk after 5 ns
c) clk <= clk after 10 ns
d) clk <= not clk after 10 ns

Aufgabe 8.2
Welche Aussagen über die Datentypen *std_logic* und *std_ulogic* sind korrekt? *(Mehrere Antworten sind richtig)*

a) Der Datentyp *std_logic* besitzt eine „Auflösungsfunktion“ *(resolution function),* der Datentyp *std_ulogic* dagegen nicht.
b) Ein Signal vom Datentyp *std_ulogic* wird zu Beginn einer Simulation immer auf ‘U’ *(undefined)* gesetzt. Ein Signal vom Datentyp *std_logic* erhält zu Beginn der Simulation immer den Wert ‘0’.
c) Die beiden Datentypen sind Teil des VHDL-Standards. Daher können sie auch ohne die Verwendung von Library- und Use-Anweisungen in VHDL-Beschreibungen eingesetzt werden.
d) Erfolgen Zuweisungen an ein Signal vom Datentyp *std_logic* aus zwei Prozessen heraus, führt dies in der Simulation nicht zu einer Fehlermeldung.

Aufgabe 8.3
Welche Aussage über die Generics sind korrekt?

a) Bei der Instanziierung eines Moduls können auch Signale an die Generics „angeschlossen“ werden.

b) Die Werte, die an die Generics übergeben werden, müssen zur Übersetzungszeit berechenbar (zum Beispiel Konstanten) sein.
c) Generics sind immer vom Datentyp *integer*.
d) Wird ein Generic verwendet, muss bei der Instanziierung des entsprechenden Moduls dem Generic immer ein Wert zugewiesen werden.

Aufgabe 8.4
Wie kann ein Prozess mithilfe der Wait-Anweisung (für immer) beendet werden?

a) wait forever;
b) wait;
c) wait until ();
d) wait on;

Aufgabe 8.5
Was gilt für Prozesse in Testbenches?

a) Eine Testbench darf nur einen einzelnen Prozess beinhalten.
b) Testbench-Prozesse dürfen mehrere Wait-Anweisungen beinhalten.
c) Testbench-Prozesse dürfen eine Sensitivitätsliste besitzen und gleichzeitig eine Wait-Anweisung beinhalten.
d) Testbench-Prozesse dürfen nur synthetisierbaren Code beinhalten.

Aufgabe 8.6
Gegeben ist der nachfolgende VHDL-Prozess.

```
process
   file my_file : text;
   variable my_f_status : FILE_OPEN_STATUS;
   variable l : line;
   variable slv : std_logic_vector (3 downto 0);
begin
   file_open(my_f_status, my_file, "test.txt", WRITE_MODE);
   if my_f_status = OPEN_OK then
        for i in 1 to 5 loop
            write (l,i);
            write (l,string'(" "));
            slv := std_logic_vector(to_unsigned(i,4));
            write (l,slv);
            writeline(my_file,l);
        end loop;
   end if;
   wait;
end process;
```

Welche Ausgabe erwarten Sie in der Datei *test.txt*?

a.

```
1
2
3
4
5
```

b.

```
1 001
2 010
3 011
4 100
5 101
```

c.

```
1 0001
2 0010
3 0011
4 0100
5 0101
```

d.

```
1
0001
2
0010
3
0011
4
0100
5
0101
```

Programmierbare Logik

9

In Kapitel 7 wurden programmierbare Logikbausteine bereits kurz vorgestellt. Diese Bausteine zeichnen sich dadurch aus, dass ihre logische Funktion durch den Anwender festgelegt werden kann. Viele programmierbare Logikbausteine lassen sich mehrfach programmieren, so dass sich eventuelle Designfehler innerhalb kurzer Zeit durch eine Neuprogrammierung beheben lassen. Ebenso können beispielsweise auch Änderungen der Spezifikation des Zielsystems selbst in späten Phasen des Entwicklungsprozesses eingearbeitet werden. Auf Grund dieser Vorteile haben sich programmierbare Logikbausteine in vielen Bereichen durchgesetzt. Mit einigen dieser Bausteine lassen sich nur wenige Gatter ersetzen, andere ermöglichen dagegen die Realisierung von komplexen digitalen Systemen.

Zur Beschreibung der gewünschten logischen Funktion wird meist VHDL verwendet. Der VHDL-Code wird von Software-Tools, die teilweise kostenlos von den Baustein-Herstellern zur Verfügung gestellt werden, interpretiert und für den Zielbaustein optimiert. Das Ergebnis ist eine binäre Datei, die auf die programmierbare Logikkomponente geladen wird. Erst durch diesen Programmiervorgang erhält der Baustein seine finale digitale Funktion.

Die Preise der Bausteine unterscheiden sich erheblich: Während einfache Bausteine für wenige Cent erworben werden können, müssen für komplexere Bausteine zwei- oder dreistellige Eurobeträge aufgebracht werden. Auch für extrem komplexe Spezialanwendungen stehen Bausteine zur Verfügung. Da diese Bausteine jedoch eine relativ große Siliziumfläche benötigen und sie nur in relativ kleinen Stückzahlen verkauft werden, erreichen die Preise dieser Komponenten vier- oder sogar fünfstellige Eurobeträge.

Auch wenn der Begriff *Programmierbarkeit* eine Nähe zu Software-Programmen nahelegt, handelt es sich dennoch um unterschiedliche Konzepte. Ein Software-Programm wird auf einen Computer geladen und dann sequenziell vom Prozessor des Rechners ausgeführt. Im Fall programmierbarer Logik wird zwar auch die Information über die auszuführende Funktion auf den Baustein geladen, die Ausführung dieser Funktion geschieht jedoch direkt in Hardware und nicht durch eine sequenzielle Interpretation

W. Gehrke et al., *Digitaltechnik,* Springer-Lehrbuch,
DOI 10.1007/978-3-662-49731-9_9

der Befehle eines Computerprogramms. Um den Unterschied der Konzepte deutlich zu machen, werden programmierbare Logikbausteine auch als *konfigurierbare Logik* bezeichnet.

Im Rahmen der folgenden Abschnitte werden zunächst die technischen Grundkonzepte programmierbarer Logikbausteine erläutert. Diese werden anschließend aufgegriffen und es werden unterschiedliche Typen programmierbarer Logikbausteine vorgestellt.

9.1 Grundkonzepte programmierbarer Logik

Für die Realisierung eines Bausteins, dessen Funktion erst durch den Anwender festgelegt wird, können zwei grundlegende Konzepte verfolgt werden, die im Folgenden näher erläutert werden.

9.1.1 Zweistufige Logik

Eine beliebige kombinatorische Funktion lässt sich mithilfe des KV-Diagramms – oder bei komplexeren Funktionen mithilfe eines geeigneten Computerprogramms – in eine zweistufige Darstellung überführen. Wird beispielsweise eine disjunktive Darstellung der Funktion angestrebt, besteht die erste Logikstufe aus UND-Verknüpfungen während in einer zweiten Stufe ODER-Verknüpfungen verwendet werden. Um einen Baustein zu realisieren, dessen logische Funktion vom Anwender in disjunktiver Form programmiert werden kann, muss dieser Baustein also eine zweistufige UND-/ODER-Struktur enthalten. Durch die Auswahl, ob ein Eingangssignal in der UND-Stufe berücksichtigt wird, können die Produktterme der gewünschten Funktion in der UND-Stufe realisiert werden. Die Produktterme werden mit der ODER-Stufe zum Ausgangssignal der Funktion zusammengefasst.

Um die Auswahl der zu berücksichtigenden Eingangssignale und Produktterme zu ermöglichen, werden neben UND- und ODER-Gattern elektrische Schalter benötigt, die die Eingangssignale beziehungsweise Produktterme mit den Eingängen der Gatter verbinden. Soll ein Gattereingang unberücksichtigt bleiben, wird der Schalter so programmiert, dass eine logische 1 (bei UND-Gattern) beziehungsweise eine logische 0 (bei ODER-Gattern) zugeführt wird.

Die Grundstruktur eines solchen programmierbaren Logikbausteins ist in Abb. 9.1 dargestellt. Der Baustein besitzt die drei Eingänge X1, X2 und X3. Die an diesen Eingängen anliegenden Signale können den UND-Gattern negiert oder nicht-negiert zugeführt werden. In dem dargestellten Beispiel können mithilfe der beiden UND-Gatter insgesamt zwei Produktterme gebildet werden. Wird nur ein Term benötigt, kann einer der Eingänge des nicht benötigten UND-Gatters auf Null gesetzt werden. Auf diese Weise wird sichergestellt, dass der Ausgang des UND-Gatters, unabhängig von den Werten der anderen Eingänge, den Wert 0 besitzt und somit in der nachfolgenden ODER-Stufe nicht berücksichtigt wird.

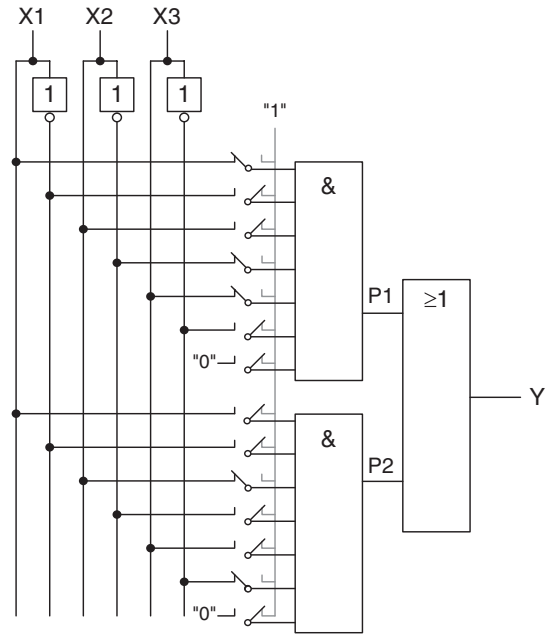


Abb. 9.1 Struktur eines zweistufigen programmierbaren Logikbausteins mit 3 Eingängen und einem Ausgang

Mit der in Abb. 9.1 gezeigten Beispielprogrammierung werden die Terme *P1* und *P2* durch die folgenden logischen Gleichung beschrieben:

$$P1 = X1 \,\&\, \overline{X2} \,\&\, X3$$

beziehungsweise

$$P2 = X2 \,\&\, \overline{X3}$$

Damit ergibt sich der Ausgangswert für *Y* zu

$$Y = P1 \vee P2 = (X1 \,\&\, \overline{X2} \,\&\, X3) \vee (X2 \,\&\, \overline{X3})$$

Mithilfe der dargestellten Schaltung lassen sich beliebige kombinatorische Funktionen realisieren, wenn diese maximal drei Eingangsvariablen besitzen und sie sich mithilfe von maximal zwei Termen beschreiben lassen.

Um auch komplexere logische Funktionen realisieren zu können, kann die Grundschaltung mit mehr UND-Gattern ausgestattet werden. Sollen darüber hinaus auch mehrere Ausgangssignale gleichzeitig berechnet werden, werden weitere ODER-Gatter hinzugefügt. Es ist nachvollziehbar, dass eine vollständige grafische Darstellung eines solchen Bausteins schnell unübersichtlich werden kann. Daher wird häufig eine kompaktere Darstellung gewählt, bei der die Eingänge der UND-Gatter in einem einzelnen Strich zusammengefasst werden. Hierbei entfällt auch die explizite Darstellung der Schalter. Diese werden durch Punkte ersetzt. Ein gesetzter Punkt deutet an, dass

der zugehörige Schalter so programmiert ist und damit eine Verbindung zwischen dem jeweiligen Eingangssignal und der UND-Stufe hergestellt ist. Fehlt der Punkt dagegen, liegt an dem zugehörigen Eingang des UND-Gatters eine 1 an.

Für das obige Beispiel ist die kompakte Darstellung in Abb. 9.2 abgebildet.

Das in diesem Abschnitt vorgestellte Grundprinzip wird bei sogenannten *Programmable Logic Devices (PLDs)* verwendet, die in den Abschn. 9.2 und 9.3 näher vorgestellt werden. Ist neben dem UND-Array auch das ODER-Feld programmierbar, wird meist der Begriff *Programmable Logic Arrays (PLA)* verwendet.

Der Vorteil des programmierbaren ODER-Feldes eines PLAs ist es, dass die Produktterme allen ODER-Verknüpfungen zugeführt werden. Wird ein Produktterm für die Berechnung von mehr als einem Ausgang benötigt, muss der Term daher nur einmal durch die entsprechende UND-Verknüpfung gebildet werden. Dieser Vorteil der PLA-Struktur muss mit der Programmierbarkeit des ODER-Feldes erkauft werden, was letztlich zu einem höheren Flächenbedarf des Bausteins und damit zu höheren Kosten führt.

Ein Beispiel soll die mehrfache Verwendung eines Produktterms verdeutlichen: Es werden die Funktionen

$$Y1 = P1 \vee P2 = (X1 \,\&\, \overline{X2} \,\&\, X3) \vee (X2 \,\&\, \overline{X3})$$

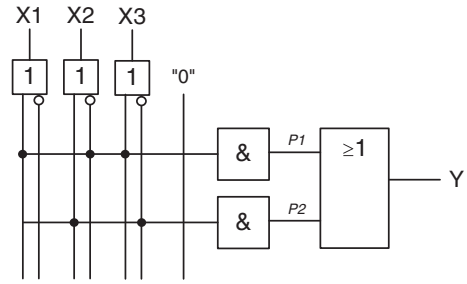


Abb. 9.2 Beispiel eines programmierbaren Logikbausteins in kompakter grafischer Darstellung

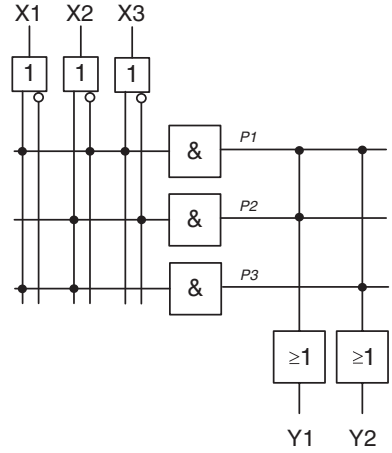


Abb. 9.3 Programmierbarer Logikbaustein mit mehrfach verwendetem Produktterm

und

$$Y2 = P1 \vee P3 = (X1 \,\&\, \overline{X2} \,\&\, X3) \vee (X1 \,\&\, X2)$$

mithilfe eines PLAs realisiert.

Die Programmierung des PLAs kann dann wie in Abb. 9.3 dargestellt realisiert werden.

9.1.2 Tabellenbasierte Logikimplementierung

Eine logische Funktion kann auch durch eine Tabelle definiert werden, welche die möglichen Eingangswerte mit den zugehörigen Ausgangswerten auflistet. Diese tabellarische Darstellungsform kann für eine direkte Implementierung in Hardware verwendet werden. Als Grundelemente werden in diesem Fall statt Gatter sogenannte Lookup-*Tabellen* (engl. *look-up table, LUT*) verwendet. Eine Lookup-Tabelle ist ein kleiner Speicher in dem für alle Eingangskombinationen die jeweiligen Ausgangswerte abgelegt sind.

Besitzt die LUT beispielsweise vier Eingänge, müssen für die Implementierung der Tabelle 16 Speicherstellen bereitgestellt werden. Die Auswahl, welche der gespeicherten Werte am Ausgang erscheint, erfolgt durch Anlegen eines 4 bit breiten Wertes an die Eingänge der LUT.

Möchte man eine LUT aus digitalen Grundelementen aufbauen, kann dies beispielsweise mithilfe von D-Flip-Flops und einem Multiplexer erfolgen. Ein Beispiel für eine Realisierung einer solchen LUT ist in Abb. 9.4 dargestellt. Dabei wird auf eine genauere Darstellung der Logik zum Schreiben der gespeicherten Werte aus Gründen der Übersichtlichkeit verzichtet.

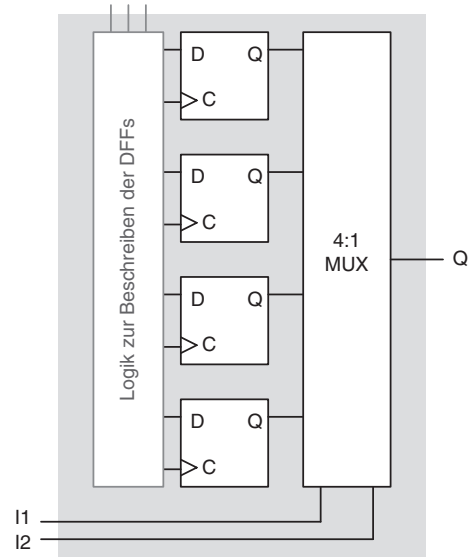


Abb. 9.4 Implementierung einer Lookup-Tabelle mit D-Flip-Flops

Auch mithilfe einer LUT-basierten Implementierung lassen sich also beliebige logische Funktionen realisieren, sofern die Anzahl der LUT-Eingänge ausreichend groß gewählt ist.

In Abb. 9.5 ist die Realisierung eines UND- und eines ODER-Gatters auf Basis der LUT mit zwei Eingängen dargestellt. Für alle möglichen Kombinationen der Eingänge *I0* und *I1* werden die entsprechenden Ausgangswerte in den Flip-Flops abgespeichert (0,0,0,1 für ein UND-Gatter und 0,1,1,1 für ein ODER-Gatter). Der Multiplexer wählt anhand der Eingangswerte *I0* und *I1* einen der vier Flip-Flop-Ausgänge aus. In dem Beispiel in Abb. 9.5 liegen am Eingang der LUT die Werte 1 und 0 an. Hiermit wird das zweite Flip-Flop ausgewählt, in dem im Fall einer UND-Verknüpfung eine 0 beziehungsweise im Fall eines ODER-Gatters eine 1 abgelegt ist.

Besitzt die zu realisierende Funktion mehr Eingänge als die verwendeten LUTs, müssen mehrere LUTs durch Parallelschaltung und Kaskadierung kombiniert werden. Welche LUTs wie kombiniert werden müssen, hängt von der zu implementierenden logischen Funktion ab.

Ein programmierbarer Logikbaustein auf Basis von LUTs muss also neben den programmierbaren LUTs auch konfigurierbare Verbindungen zwischen den einzelnen LUTs zur Verfügung stellen. So können dann auch komplexe Funktionen, bei denen mehrere LUTs kombiniert werden müssen, mithilfe des Bausteins realisiert werden.

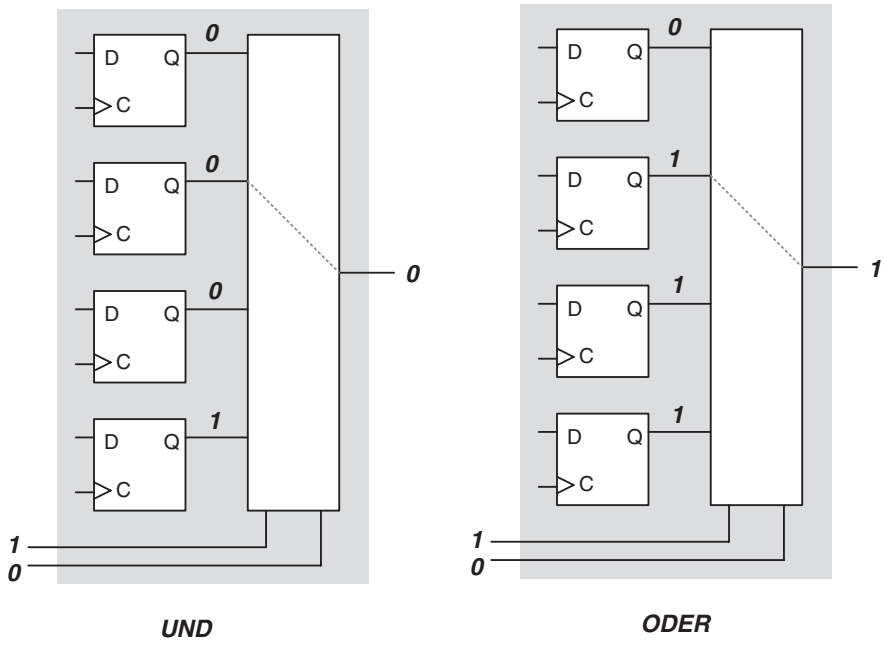


Abb. 9.5 LUT-basierte Realisierung eines UND- und eines ODER-Gatters

Das in diesem Abschnitt skizzierte Prinzip der LUT-basierten Implementierung in Kombination mit einem programmierbaren Verdrahtungskonzept setzen die Field Programmable Gate Arrays (FPGA) ein. Reale FPGAs realisieren die Speicherelemente der LUTs zur Reduktion der benötigten Chipfläche auf Basis von speziellen Speichertechnologien (zum Beispiel SRAM). Eine detailliertere Diskussion der FPGA-Technologie ist in Abschn. 9.4 zu finden.

9.2 Simple Programmable Logic Device (SPLD)

Die ersten programmierbaren Bausteine wurden bereits 1971 von der Firma Monolithic Memories Inc. entwickelt und unter dem Namen *PAL (Programmable Array Logic)* vermarktet. Heute werden diese Bausteine und ihre Nachfolger häufig auch als *Simple Programmable Logic Device (SPLD)* bezeichnet. Mit diesen Bausteinen lassen sich kombinatorische Schaltungen in disjunktiver Form realisieren.

Die Eingangssignale werden hierzu in einer Eingangsstufe verstärkt und in negierter und nicht-negierter Form für die weitere Verarbeitung zur Verfügung gestellt. Die aufbereiteten Eingangsgrößen werden einem UND-Array zugeführt, welches die benötigten Produktterme berechnet. Eine feste Verdrahtung der UND-Ausgänge mit den ODER-Eingängen sorgt für die gewünschte disjunktive Verknüpfung der Produktterme. Die Grundstruktur eines PALs entspricht also den in Abschn. 9.1.1 dargestelltem Ansatz einer zweistufigen disjunktiven Logikimplementierung, bei der die Programmierbarkeit durch konfigurierbare Verbindungen im UND-Array erreicht wird, während das ODER-Feld festverdrahtet ist.

Darüber hinaus bieten die Bausteine die Möglichkeit, einige der Ausgänge wahlweise auch als Eingang zu nutzen. So können auch komplexere Funktionen, die eine höhere Anzahl an Eingangssignalen benötigen, mithilfe des Bausteins realisiert werden. Die Ausgänge werden hierzu mit Tri-State-Treibern versehen, deren Ausgänge durch eine entsprechende Programmierung des Bausteins in einen hochohmigen Zustand versetzt werden können. An diesen Anschlüssen können dann Eingangssignale angelegt werden, deren Werte ebenfalls im UND-Feld verarbeitet werden können.

Die Grundstruktur eines PAL-Bausteins mit Eingängen *(I),* Ausgängen *(O)* und Ein-/Ausgängen *(I/O)* ist in Abb. 9.6 dargestellt.

Da mithilfe derartiger Bausteine auch endliche Automaten realisiert werden sollen, ist es sinnvoll, die hierfür notwendigen Register auf dem Chip vorzusehen. Daher wurden neben PALs mit der in Abb. 9.6 gezeigten Struktur auch Bausteine entwickelt, die bereits D-Flip-Flops enthalten. Die Grundstruktur eines solchen Bausteins zeigt Abb. 9.7.

Eine besondere Eigenschaft von PALs ist es, dass eine einmal programmierte Funktion nicht modifiziert werden kann. Dieser Nachteil wurde mithilfe der sogenannten *GALs (Generic Array Logic)* vermieden. Das Grundprinzip dieser Bausteine ist allerdings sehr ähnlich. Teilweise können GALs auch als Ersatz für PALs eingesetzt werden.

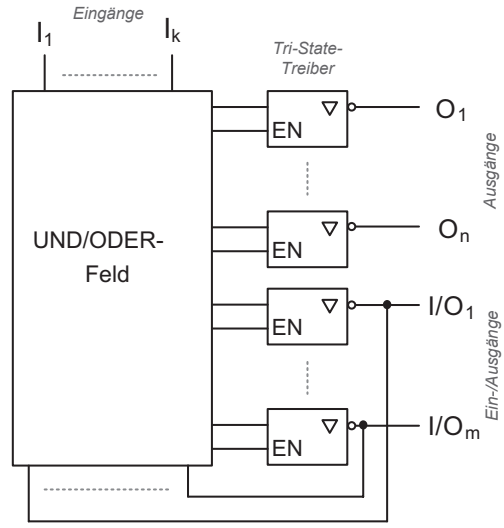


Abb. 9.6 Struktur eines PAL-Bausteins

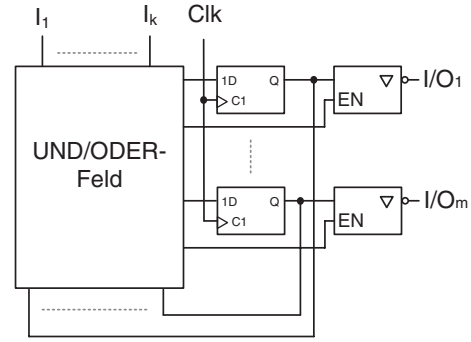


Abb. 9.7 Grundstruktur eines programmierbaren Logikbausteins mit UND/ODER-Struktur und Registern

Die Bedeutung von PALs und GALs ist in den letzten Jahren zurückgegangen und sie werden in Neuentwicklungen meist nicht mehr eingesetzt. Obwohl die Bausteine noch angeboten werden, haben einige Hersteller die Bausteinfamilien bereits abgekündigt. Statt der PALs werden heute meist die im nachfolgenden Abschnitt vorgestellten CPLDs verwendet (Tab. 9.1).

Tab. 9.1 Beispiele einiger PAL-Bausteine

Bezeichnung	Anzahl Eingänge	Anzahl Ein-/Ausgänge		Anzahl Minterme je Ausgang
		Ohne Register	Mit Registern	
PAL16L8	10	8	0	7
PAL16R4	8	4	4	7
PAL16R8	8	0	8	7
PAL20R8	12	0	8	8

9.3 Complex Programmable Logic Device (CPLD)

Eine Weiterentwicklung des PLA-Konzeptes stellen die sogenannten *Complex Programmable Logic Devices (CPLDs)* dar. Viele dieser Bausteine bedienen sich dem PLA-Konzept und kombinieren mehrere PLA-Blöcke mit einer programmierbaren Verbindungsmatrix, die es ermöglicht, die Ausgänge eines PLA-Blocks mit den Eingängen eines anderen Blocks zu verbinden. Auf diese Weise ist die Implementierung der logischen Funktion nicht allein auf die disjunktive Form beschränkt. Es können auch mehrere disjunktive Stufen kaskadiert werden. Dies kann insbesondere bei komplexeren Funktionen zu einer günstigeren Realisierung führen.

Die Grundstruktur eines CPLDs ist in Abb. 9.8 dargestellt. Neben den programmierbaren UND/ODER-Strukturen *(PLA)* besitzen CPLDs sogenannte Makrozellen *(Macro Cell, MC).* Die Makrozellen können als eine Erweiterung der Registerstufen einfacher PLA-Bausteine aufgefasst werden. Der schematische Aufbau der Makrozelle eines CPLDs der Coolrunner-II-Serie (Fa. Xilinx) ist in Abb. 9.9 dargestellt. Der Kern der Makrozelle ist ein D-Flip-Flop, dessen D-Eingang mit der PLA-Struktur verbunden ist. Mithilfe eines Exklusiv-Oder-Gatters kann entschieden werden, ob der durch die UND/ODER-Struktur berechnete Term nicht-invertiert oder invertiert an das D-Flip-Flop weitergereicht wird. Die Rückführung des Terms in die Verbindungsmatrix kann sowohl asynchron (Abgriff vor dem Flip-Flop) oder synchron (Abgriff hinter dem Flip-Flop) erfolgen. Ebenso kann für die Ausgabe eines Wertes ausgewählt werden, ob diese asynchron oder synchron erfolgen soll. Das Flip-Flop der dargestellten Makrozelle besitzt Enable-, Set- und Reset-Eingänge, die ebenfalls mithilfe der PLA-Struktur angesteuert werden.

In der Praxis stellt sich die Frage, welcher CPLD-Baustein zur Lösung eines konkret vorliegenden Problems geeignet ist. Neben der benötigten Anzahl an Ein- und Ausgängen spielt hierbei auch die Frage, wie viele Gatter durch ein bestimmtes CPLD ersetzt werden

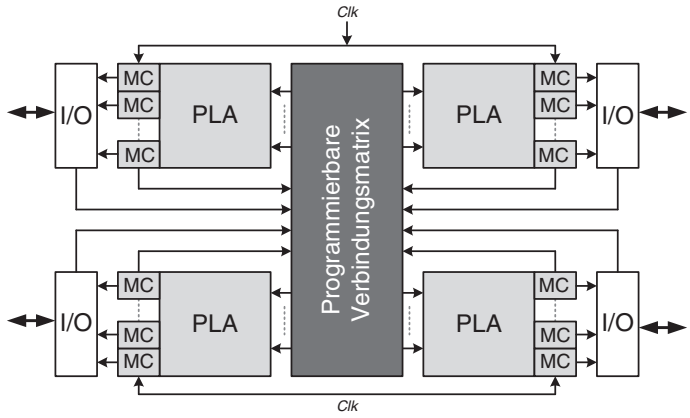


Abb. 9.8 Struktur eines CPLDs auf PLA-Basis

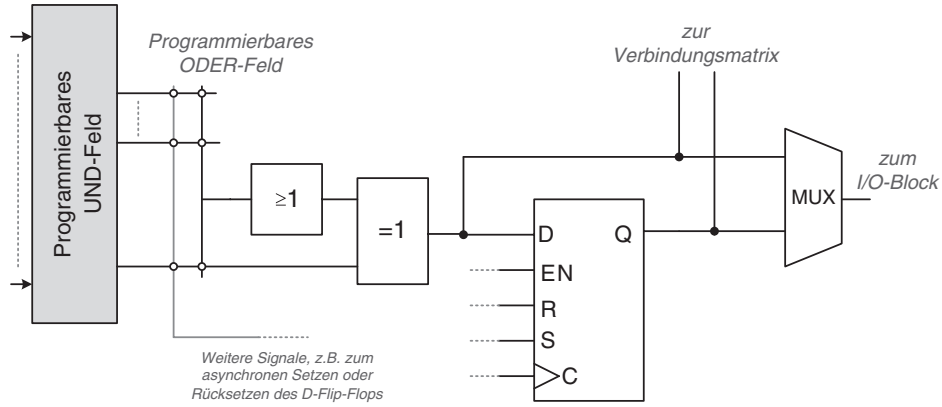


Abb. 9.9 Schematischer Aufbau einer Makrozelle

können, eine wichtige Rolle. Die Antwort auf diese Fragestellung lässt sich häufig nicht alleine durch den Blick auf die Architektur eines CPLDs beantworten. Passt die zu implementierende Funktion nur schlecht zu der im CPLD-Baustein vorgegebenen Struktur, wird die Realisierung ineffizient sein, so dass viele Teile der verfügbaren CPLD-Ressourcen nicht genutzt werden können. Daneben hat auch die Effizienz der Syntheseprogramme, die zum Umsetzen der in VHDL beschriebenen Funktion verwendet werden, einen nicht unerheblichen Einfluss auf das Ergebnis. In der Praxis wird man daher, sofern nicht auf Erfahrungswerte aus ähnlich gelagerten Fällen zurückgegriffen werden kann, vor der finalen Auswahl eines CPLD Bausteins mehrere Syntheseläufe ausführen, um so den Ressourcenverbrauch für unterschiedliche Bausteine abschätzen zu können.

Tab. 9.2 fasst exemplarisch einige wichtige Parameter der CPLD-Familie CoolRunner-II der Firma Xilinx zusammen.

CPLDs werden von mehreren Herstellern angeboten. Die wichtigsten sind Xilinx, Altera, Lattice, MicroSemi und Atmel. Einige Anbieter, wie die Firmen Altera oder Lattice, setzen als Alternative zu dem hier vorgestellten PLA-basierten Konzept eine LUT-basierte Realisierung ein, die bis vor einigen Jahren hauptsächlich im Bereich der im nachfolgenden Abschnitt beschriebenen FPGAs zu finden war.

Tab. 9.2 Parameter der CPLD-Familie CoolRunner-II (Xilinx)

	Baustein					
	XC2C32A	XC2C64A	XC2C128	XC2C256	XC2C384	XC2C512
Makrozellen	32	64	128	256	384	512
Max. I/Os	33	64	100	184	240	270
Max. Taktfrequenz F_{system} (MHz)	323	263	244	256	217	179

Tab. 9.3 Parameter der CPLD-Familie MAX V (Altera)

	Baustein						
	5M40Z	5M80Z	5M160Z	5M240Z	5M570Z	5M1270Z	5M2210Z
Logic Elements	40	80	160	240	570	1270	2210
Äquiv. Makrozellen	32	64	128	192	440	980	1700
Max. I/Os	54	79	79	114	159	271	271
Verzögerungszeit, pin-to-pin (ns)	7,5	7,5	7,5	7,5	9,0	6,2	7,0

Um einen Vergleich mit PLA-basierten CPLDs zu unterstützen geben die Hersteller zum Teil an, wie vielen Makrozellen ein CPLD entspricht. Als ein Beispiel hierfür sind in Tab. 9.3 einige Parameter der MAX V Serie der Firma Altera angegeben. Das Kernelement dieser CPLDs ist ein Logic Element (LE). Ein Logic Element enthält eine LUT mit 4 Eingängen, ein Flip-Flop sowie Logik zum Setzen oder Rücksetzen des Flip-Flops.

9.4 Field Programmable Gate Arrays

Der Begriff *Gate Array* bezeichnete ursprünglich Bausteine, die aus einem großen Feld vorgegebener Logikgatter bestand. Die Verdrahtung der Gatter, und damit die zu realisierende logische Funktion, konnte vom Kunden festgelegt werden. Die Verdrahtung der Gatter wurde dann im Auftrag des Kunden in einer Halbleiterfabrik realisiert. Auch die Funktion der heute üblichen Form der Gate-Arrays, die Field Programmable Gate Arrays, kann durch den Anwender festgelegt werden. Da die Programmierung elektrisch erfolgt, sind keine zeitaufwendigen Produktionsschritte in einer Halbleiterfabrik erforderlich: Mithilfe eines Programmiergerätes kann die gewünschte logische Funktion in wenigen Sekunden auf ein FPGA geladen werden. Da FPGAs zu attraktiven Preisen angeboten werden, haben Sie sich in vielen Bereichen durchgesetzt. Im Folgenden werden die Grundkonzepte dieser Bausteine näher vorgestellt.

9.4.1 Allgemeiner Aufbau eines FPGAs

Wie bei anderen programmierbaren Logikbausteinen lässt sich die digitale Funktion eines FPGAs im Feld programmieren. Ein wesentliches Merkmal von FPGAs ist es, dass sich deutlich komplexere Funktionen realisieren lassen, als dies mit PALs oder CPLDs möglich wäre. Auch im Hinblick auf die technische Realisierung der „Programmierbarkeit“ unterscheiden sich FPGAs von vielen CPLDs. Während einfache Logikbausteine (hierzu zählen wir auch CPLDs) im Kern eine zweistufige UND/ODER-Struktur einsetzen, basieren FPGAs auf einer tabellenbasierten Implementierung.

Die Grundidee eines FPGAs ist relativ einfach: Man realisiert einen Baustein, der viele kleine *Logikblöcke* enthält, in denen sich programmierbare Lookup-Tabellen (LUTs) befinden. Jede LUT besitzt beispielsweise vier Eingänge und einen Ausgang. Die spätere Programmierung der LUTs legt fest, nach welcher logischen Funktion der Ausgangswert aus den Eingängen berechnet werden soll. Da für die Implementierung eines digitalen Systems auch Flip-Flops benötigt werden, enthalten die Logikblöcke auch Flip-Flops. Meist sind die gleiche Anzahl an LUTs und Flip-Flops vorhanden, da dies dem Bedarf in praktischen Schaltungen entspricht. Dabei wird jeder LUT ein FF zugeordnet, so dass der Ausgangswert einer LUT auch innerhalb eines Logikblocks gespeichert werden kann.

Für die Verbindungen zwischen den Logikblöcken wird ein Verbindungsnetzwerk eingesetzt. Die Programmierbarkeit des Verbindungsnetzwerkes wird durch programmierbare Schalter erreicht *(Switch Matrix).* Die Funktionsweise des Verbindungsnetzwerkes kann man mit Gleisen einer Eisenbahn vergleichen: Sollen Daten von einem Logikblock an einen bestimmten anderen Logikblock gesendet werden, werden die „Weichen" innerhalb des Netzwerkes so programmiert, dass eine elektrische Verbindung zwischen den beiden Logikblöcken hergestellt wird. Im Gegensatz zu einer Eisenbahnverbindung werden die Weichen nicht dynamisch im Betrieb umgeschaltet, sondern sie werden nach dem Einschalten einmalig für die gewünschte logische Funktion konfiguriert.

Durch die Programmierbarkeit der Logik-Blöcke und des Verbindungsnetzwerkes, können komplexe logische Funktionen durch die Kombination mehrerer LUTs umgesetzt werden. Die maximale Komplexität der Gesamtfunktion ist natürlich durch die Anzahl der verfügbaren Logikblöcke begrenzt.

Darüber hinaus ist es natürlich auch denkbar, dass das Verdrahtungsnetzwerk der limitierende Faktor einer FPGA-basierten Systemimplementierung ist, wenn eine sehr aufwendige Signalverdrahtung benötigt wird. In diesem Fall können nicht alle vorhandenen LUTs genutzt werden.

Neben den Logikblöcken und dem Verbindungsnetzwerk enthalten FPGAs auch Ein-/Ausgabeblöcke *(IO-Blocks* oder kurz *IOBs).* Mithilfe dieser Blöcke kann unter anderem eine Anpassung von Logikpegeln erfolgen. Arbeitet ein FPGA beispielsweise mit einer internen Versorgungsspannung von 1,8 V, können die Pegel der internen Signale mithilfe der IOBs so angepasst werden, dass sie auch Bausteinen mit einer Versorgungsspannungsspannung von 3,3 V zugeführt werden können. Daneben stehen in den IOBs auch Funktionen zur Verfügung, die für eine besonders schnelle Ein-/Ausgabe hilfreich sein können. Ein Beispiel hierfür sind IOB-interne Parallel-Seriell-Wandler, die auf Schieberegistern basieren *(Serializer).* Ausgabedaten werden von den Logikblöcken parallel (zum Beispiel 4 oder 8 bit) an die IOBs herangeführt. Innerhalb des IOBs werden die Daten „serialisiert", das heißt Bit für Bit am äußeren Anschluss des FPGAs ausgegeben. Auf diese Weise kann eine hohe Datenrate am Ausgang des FPGAs realisiert werden, obwohl die Implementierung der logischen Funktion innerhalb des FPGAs vergleichsweise langsam ist. Für die Eingabe können *De-Serializer* eingesetzt werden, welche die Daten seriell einlesen und für die FPGA-interne Logik in paralleler Form zur Verfügung stellen.

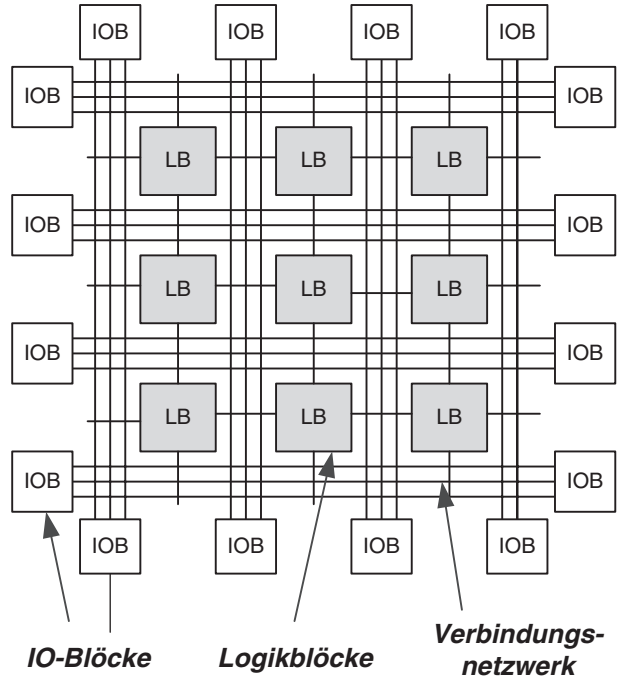


Abb. 9.10 Prinzipieller Aufbau eines FPGAs

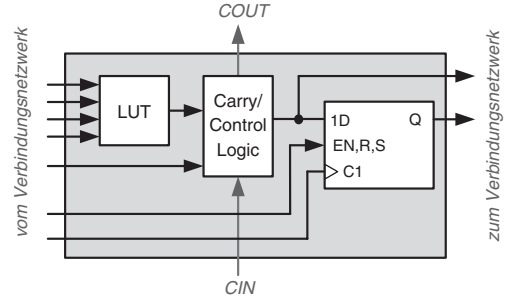


Abb. 9.11 Aufbau eines Logikblocks

Die Grundstruktur eines FPGAs, das aus Logik-Blöcken, IO-Blöcken und einem Verbindungsnetzwerk besteht, ist in Abb. 9.10 dargestellt.

Der Aufbau eines einfachen Logikblocks ist in Abb. 9.11 dargestellt. Die meisten Eingänge und Ausgänge sind mit dem Verbindungsnetzwerk verbunden. Darüber hinaus existieren die Anschlüsse *CIN* und *COUT,* die mit dem Block *Carry/Control-Logic* verbunden sind. Mithilfe dieser Anschlüsse und der zugehörigen Logik wird ein besonders schneller Durchlauf der Übertragsbits eines Addierers ermöglicht. Mithilfe dieser besonderen Carry-Logik kann die Verzögerungszeit der Addition deutlich reduziert werden.

9.4.2 Taktverteilung im FPGA

In digitalen Systemen werden die Datenausgabe und die Datenübernahme der Flip-Flops mithilfe von Taktsignalen gesteuert. Im Idealfall „sehen“ alle Flip-Flops eines Systems zum gleichen Zeitpunkt die steigende Flanke eines Taktsignals. In der Praxis lässt sich dieser Idealfall nicht realisieren, da Taktsignale, wie alle anderen Signale, über Leitungen des Chips an die Flip-Flops herangeführt werden müssen. Reale Leitungen besitzen eine Verzögerungszeit, so dass Flip-Flops, die nah an einer Taktquelle platziert sind, eine steigende Flanke etwas eher sehen als ein Flip-Flop, das am Ende einer Taktleitung liegt.

Dass dies ein potentielles Problem für die Realisierung eines Systems darstellen kann, macht folgendes Beispiel deutlich: Nehmen wir vereinfachend an, dass die verwendeten D-Flip-Flops eine Setup-Zeit und ein *Clock-to-Q-Delay* (also die Zeit, die benötigt wird, um nach der steigenden Taktflanke den im Flip-Flop gespeicherten Wert am Ausgang zur Verfügung zu stellen) von jeweils 1 ns besitzen. Nehmen wir weiterhin an, die Logik und die Verdrahtung zwischen zwei derartigen Flip-Flops habe eine Verzögerungszeit von 3 ns. Mit diesen Werten würde sich eine minimale Taktperiode von $1\text{ ns} + 1\text{ ns} + 3\text{ ns} = 5\text{ ns}$ ergeben. Dieses System kann also bei idealer Taktverteilung mit maximal 200 MHz betrieben werden.

Erhält das zweite Flip-Flop die steigende Flanke früher als das erste Flip-Flop, vergrößert sich die maximale Periodendauer entsprechend, da das empfangende (zweite) Flip-Flop bereits früher stabile Daten am Eingang erwartet. Nehmen wir an, die zeitliche Verschiebung des Taktsignals (im Fachjargon auch *Clock Skew* genannt) betrage 2 ns. Dann würde sich die minimale Taktperiode um diese 2 ns auf 7 ns erhöhen und damit die maximale Taktfrequenz des Systems auf etwa 140 MHz absinken.

Was kann man also tun? Nun, die Signalverzögerungen beruhen auf physikalischen Gesetzen und können daher nicht eliminiert oder umgangen werden. Aber ein erster Schritt zur Problemlösung ist es, die Verzögerungen des Taktsignals innerhalb des Chips zu kennen. Auf Basis dieser Kenntnis kann für jedes Flip-Flop, dessen Ausgang mit einem anderen Flip-Flop verbunden ist, die Verzögerung des Taktsignals abgeschätzt und bei der Logik-Synthese entsprechend berücksichtigt werden. Aber natürlich löst dies noch nicht das eigentliche Problem, dass große Verzögerungen des Taktsignals zu einer signifikanten Reduktion der Systemfrequenz und damit der Rechenleistung führen können. Um dieses zu Problem zu reduzieren, setzen FPGAs spezielle Verbindungsnetzwerke zur Verteilung der Taktsignale ein. Ein Beispiel für den Aufbau eines Taktnetzwerks mit zentralen Taktreibern ist in Abb. 9.12 dargestellt.

Die Taktsignale werden baumartig im System verteilt. Auf diese Weise wird erreicht, dass der Clock-Skew in einem akzeptablen Rahmen gehalten werden kann und es kann davon ausgegangen werden, dass Flip-Flops, die sich in örtlicher Nähe befinden, in etwa das gleiche zeitliche Verhalten des Taktes sehen. Werden zwei Flip-Flops, die weit voneinander entfernt liegen, miteinander verbunden, kann hierbei natürlich weiterhin ein signifikanter Clock-Skew auftreten. So sehen zum Beispiel Flip-Flops, die in der Nähe der Takttreiber liegen, die steigende Flanke deutlich eher als Flips-Flops, die in den Ecken des FPGAs platziert sind.

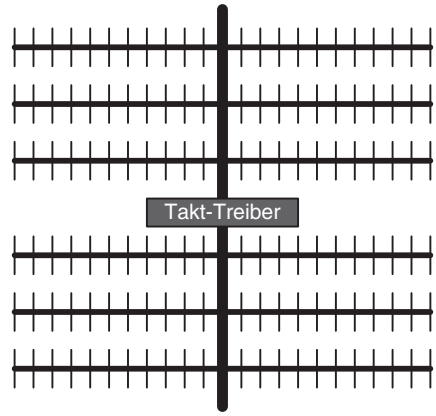


Abb. 9.12 Taktnetzwerk mit zentralen Takttreibern

Mit Fortschreiten der Halbleitertechnologie wird dieses Problem verschärft: Einerseits steigen die erzielbaren Taktfrequenzen kontinuierlich an, wodurch die Verzögerungen durch das Taktverteilungsnetzwerk immer deutlicher spürbar werden. Andererseits werden die geometrischen Abmessungen der Leitungen kleiner, was zu einem höheren Widerstand und damit zu langsameren Pegelwechseln führt. Um den Nachteil der zentralen Taktpufferung zu reduzieren, werden in modernen FPGAs Taktreiber eingesetzt, die über den Chip verteilt sind. Auf diese Weise wird die Leitungslänge zwischen Takttreiber und Takteingang der Flip-Flops reduziert und damit der Clock-Skew deutlich reduziert.

Aus diesen Erläuterungen zum Aufbau des Taktnetzwerkes ergibt sich auch, dass man niemals ein Taktsignal aus einer logischen Funktion heraus generieren sollte, da dieses Taktsignal nicht über das Taktnetzwerk geführt werden kann und somit signifikante Clock-Skew-Probleme die Folge sein können. Da das Taktsignal über Gatter geführt wird, wird dies in der Praxis auch als *Gated Clock* bezeichnet. Insbesondere Anfängern im FPGA-Design unterläuft nicht selten der Fehler, dass versehentlich Gated Clocks in einem VHDL-Entwurf realisiert werden, indem zum Beispiel ein Ausgangssignal eines Moduls einfach mit dem Takteingang eines anderen Moduls verbunden wird. Syntaktisch ist dies völlig korrekt und auch in der Simulation der Schaltung wird man häufig keine unerwarteten Ergebnisse sehen. Um das Risiko zu minimieren, dass versehentlich Gated Clocks in einem VHDL-Entwurf eingebaut werden, sollten die Takteingänge aller VHDL-Module mit einem (wenn im System nur ein Takt verwendet wird: mit dem gleichen) Taktsignal verbunden werden. Jegliche logische Verknüpfungen (und seien sie noch so simpel) eines Taktsignals mit anderen Signalen sollten vermieden werden.

9.4.3 Typische Spezialkomponenten

Um die Implementierung von logischen Funktionen besser zu unterstützen, enthalten heutige FPGAs vielfach Spezialkomponenten, die zusätzlich zu den Logikblöcken für

die Implementierung eines Systems verwendet werden können. In diesem Abschnitt werden die wichtigsten dieser Elemente kurz vorgestellt.

Mit Spezialkomponenten wird das Ziel verfolgt, eine bestimmte häufig genutzte Funktion möglichst effizient zur Verfügung zu stellen. Die Instanziierung dieser Module wird häufig von den Designtools unterstützt.

9.4.3.1 Speicherelememente

In vielen Fällen ist für die Realisierung eines Systems auch die Speicherung von Daten erforderlich. Wird eine sehr große Speicherkapazität benötigt, ist in der Regel ein externer Speicher außerhalb des FPGAs unvermeidbar. Ist der benötigte Speicherbedarf jedoch kleiner, ist eine Speicherung der Daten innerhalb des FPGAs wünschenswert, da so schneller und flexibler auf die Daten zugegriffen werden kann.

Da eine LUT letztlich auch ein kleiner Speicher ist, liegt die Idee nahe, die verfügbaren LUTs zu einem Speicher mit der benötigten Kapazität zu verbinden. Dieses Prinzip wird von FPGAs unterstützt und man spricht in diesem Fall von verteiltem Speicher *(Distributed Memory).* Der Nachteil dieses Ansatzes ist, dass die wertvollen Ressourcen der Logikblöcke für die Speicherung von Daten eingesetzt werden und nicht mehr für die Implementierung von logischen Funktionen zur Verfügung stehen.

Daher bieten heutige FPGAs auch Speicher in Form von sogenanntem *Block-RAM* an. Hierbei handelt es sich um mehrere kleine Speicher (meist in der Größe weniger kByte), die auf dem FPGA-Chip verteilt sind. Der Vorteil von Block-RAM ist, dass die erzielbare Speicherdichte, also Bits pro Siliziumfläche, um ein Vielfaches größer ist als bei der Verwendung von Distributed Memory. Daher bietet sich die Verwendung von Block-RAM immer dann an, wenn ein größerer Speicher benötigt wird beziehungsweise die Ressourcen zur Implementierung von Logik knapp sind.

Um den Speicher für verschiedene Anwendungen möglichst gut nutzen zu können, ist die Wortbreite der Block-RAMs konfigurierbar. Beispielsweise besitzen die Block-RAMs der Cyclone-V-FPGAs (Fa. Altera) eine Größe von 9 kbit. Der Speicher kann mit Wortbreiten zu 1, 2, 4, 8, 9, 16, 18, 32 oder 36 genutzt werden, wobei die maximale Anzahl der Worte immer eine Zweierpotenz ist. Da die Gesamtkapazität festliegt, nimmt die maximale Anzahl der Speicherworte mit der Wortbreite ab. So kann ein einzelner dieser Speicher zum Beispiel 8192 Worte mit einer Breite von 1 bit aufnehmen oder aber auch für die Speicherung von 512 16-Bit-Worten genutzt werden. Wortbreiten von 9, 18 und 36 bit werden unterstützt, da diese in der Kommunikationstechnik verwendet werden.

FPGA-internes Block-RAM wird meist als Dual-Port-Speicher implementiert, der zwei Schreib-/Leseanforderungen gleichzeitig bearbeiten kann. Diese Eigenschaft ist zum Beispiel dann vorteilhaft, wenn ein Modul Daten generiert, die vor der Verarbeitung durch ein zweites Modul zwischengespeichert werden müssen. Beide Module können dann unabhängig voneinander auf den Speicher zugreifen. Eine Arbitrierungslogik, die festlegt welches der beiden Module auf den Speicher zugreifen darf, kann dann entfallen.

9.4.3.2 Arithmetische Module

Eine häufig benötigte arithmetische Operation ist die Multiplikation. Daher beinhalten die meisten aktuellen FPGAs spezielle Multiplizierer-Module, die gegenüber einer LUT-basierten Implementierung der Multiplikation den Vorteil einer geringeren Verzögerungszeit bieten. Darüber hinaus kann durch die Verwendung der Hardware-Multiplizierer die Anzahl der benötigten Logikblöcke reduziert werden.

In modernen FPGAs wird das Konzept zur Unterstützung arithmetischer Funktionen häufig erweitert und es stehen nicht nur Multiplizierer zur Verfügung. Der FPGA-Hersteller Xilinx bietet beispielsweise sogenannte „DSP-Slices“ an. Hierbei handelt es sich um Module, die neben einem Multiplizierer auch einen Addierer und einen Akkumulator enthalten. Mithilfe dieser Module sollen insbesondere Anwendungen der digitalen Signalverarbeitung *(Digital Signal Processing, DSP)* unterstützt werden.

Die meisten angebotenen arithmetischen Module sind für die Verarbeitung von ganzen Zahlen ausgelegt. Einige FPGA-Serien, wie zum Beispiel Stratix-10 der Firma Altera, stellen auch Spezialhardware zur Verarbeitung von Gleitkommazahlen bereit.

9.4.3.3 Takterzeugung

FPGAs enthalten meist auch Komponenten zur Erzeugung von intern verwendeten Taktsignalen. Diese Komponenten beinhalten meist eine Phasenregelschleife (engl. *Phase-Locked Loop, PLL*), die es ermöglicht, aus einem Eingangstaktsignal Ausgangssignale zu erzeugen, deren Frequenz und Phasenlage aus dem Eingangssignal abgeleitet wird. Teilweise kommen auch *DLLs (Delay-Locked Loop)* zum Einsatz.

Die Quelle des Eingangstaktes einer PLL kann entweder ein von außen zugeführtes Signal oder ein bereits im FPGA (zum Beispiel durch eine weitere vorgeschaltete PLL) vorhandenes Taktsignal sein.

Die PLLs der meisten FPGAs erlauben die gleichzeitige Erzeugung mehrerer Taktsignale aus einem einzelnen Eingangstakt, wobei die Frequenzen der erzeugten Signale sowohl kleiner als auch größer als die Frequenz des Eingangstaktes sein können. Neben der einfachen Erzeugung unterschiedlicher Systemtaktsignale können die PLLs auch zur Synchronisierung des externen Taktsignals mit den intern verwendeten Takten verwendet werden. Dies ist insbesondere dann hilfreich, wenn die Eingangsdaten des FPGAs synchron zur Verfügung gestellt werden.

Das Blockschaltbild einer PLL zeigt Abb. 9.13. Die Phasenlage eines von außen zugeführten Taktes wird mit einem Referenztakt verglichen. Mithilfe einer Regelung wird ein analoges Signal erzeugt, welches einem spannungsgesteuerten Oszillator *(Voltage-Controlled-Oscillator, VCO)* zugeführt wird. Durch Teilung des Oszillatortaktes werden die Ausgangssignale der PLL sowie der zum Phasenvergleich zurückgeführte Referenztakt erzeugt.

9.4.3.4 Spezialisierte Peripheriemodule

Viele FPGAs bieten spezielle Peripheriemodule an, die Schnittstellen mit besonders kritischen Zeitanforderungen besitzen. Ein typisches Beispiel für derartige Module sind

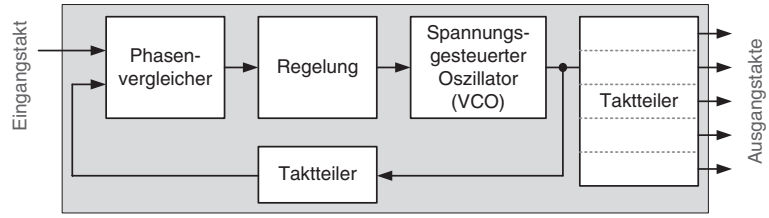


Abb. 9.13 Aufbau einer PLL

Speicher-Controller. In älteren FPGA-Generationen musste die Speicheranbindung noch mithilfe der Standard-FPGA-Ressourcen (Logikblöcke, IO-Blöcke) erfolgen. Dass hierbei wertvolle Ressourcen für eine standardisierte und häufig benötigte Funktion eingesetzt werden müssen, ist eher ein untergeordnetes Problem. Viel schwerwiegender ist häufig das Problem, dass die maximalen Taktfrequenzen, und damit die erzielbare Speicherbandbreite, bei einer Implementierung mit den üblichen FPGA-Ressourcen begrenzt sind. Um die hohen Datenraten wie sie von modernen Speicherbausteinen angeboten werden, auch für ein FPGA-basiertes Design nutzbar zu machen, werden spezialisierte Speichercontroller benötigt. Diese Komponenten sind für die Anbindung von externem Speicher optimiert und unterstützen Datenraten von mehreren GByte/s, die sich mithilfe von Logikblöcken nicht realisieren ließen. In einem beschränkten Umfang können diese Module konfiguriert werden. So sind zum Beispiel die Auswahl des Speichertyps oder auch einige Eigenschaften der FPGA-internen Schnittstelle wählbar.

Ein anderes Beispiel für ein spezielles Peripheriemodul ist eine PCI Express (PCIe) Schnittstelle. Der PCIe-Bus hat sich als wichtiger Standard für die Verbindung von Komponenten etabliert. Da die Implementierung einer PCIe-Schnittstelle besondere Anforderungen (insbesondere im Hinblick auf das Zeitverhalten) stellt, ist eine Implementierung mit Standard-FPGA-Ressourcen schwierig und aufwendig. Dieser Nachteil wird durch die Bereitstellung von PCIe-Hard-Macros vermieden und die entsprechende Funktion kann so effizienter und mit geringerem Aufwand implementiert werden.

9.4.3.5 Prozessor-Subsysteme

Häufig besteht der Wunsch, Teile eines Systems „in Hardware“ auf einem FPGA, andere Teile dagegen „in Software“ auf einem Mikroprozessor zu implementieren.

Natürlich kann ein Mikroprozessor auch mithilfe von Logikblöcken implementiert werden. Die FPGA-Hersteller bieten hierzu entsprechende Prozessordesigns (zum Beispiel *NIOS* der Firma Altera oder *Microblaze* der Firma Xilinx) mit den zugehörigen Werkzeugen zur Softwareentwicklung an. Da diese Prozessoren mithilfe der flexibel einsetzbaren programmierbaren Logikkomponenten implementiert werden, werden sie auch als *Soft-Prozessoren* bezeichnet. Allerdings gilt auch für diese Lösungen, dass ihre Effizienz eher als moderat anzusehen ist.

Wäre es da nicht logisch, in einem FPGA neben den spezialisierten Hard-Makros auch Prozessoren – oder am besten gleich ganze Prozessorsysteme – anzubieten?

Genau dieser Ansatz wird von einigen Herstellern verfolgt. So bieten zum Beispiel die Firmen Xilinx und Altera Chips an, die neben einem FPGA-Teil auch Multikern-Rechner-Systeme beinhalten. Die maximalen Taktfrequenzen, und damit die erzielbare Rechenleistung dieser Systeme erreichen ein Vielfaches von dem der Soft-Prozessoren. Da diese Chips nicht mehr als reine FPGAs anzusehen sind, werden sie von den Herstellern unter dem Begriff *System-on-Chip (SoC)* vermarktet. Dieser Begriff soll deutlich machen, dass es sich um komplette Systeme handelt, deren Funktion sich als Kombination von Software (auf dem CPU-Subsystem) und Hardware (auf dem FPGA-Teil) festlegen lässt.

9.5 FPGA-Familien

Der FPGA-Markt ist auf den ersten Blick relativ unübersichtlich. Es gibt unterschiedliche Anbieter, wobei die Firmen Xilinx und Altera (inzwischen von der Firma Intel übernommen) zusammen ca. 90 % des Marktes bedienen. Die Hersteller bringen schritthaltend mit der Weiterentwicklung der Halbleitertechnik etwa alle 2 Jahre eine neue Bausteingeneration heraus. Innerhalb dieser Generationen werden wiederum unterschiedliche Familien angeboten, die FPGAs mit ähnlichen Grundeigenschaften beinhalten, sich aber im Hinblick auf die Komplexität (Anzahl der Logikblöcke und Hard-Makros, Größe des internen Speichers usw.) unterscheiden.

Die Bausteine einer Generation werden häufig in einer besonders preisgünstigen „Low-Cost"-Familie und einer besonders leistungsstarken „High-Performance"-Familie angeboten. Daneben werden teilweise auch „Mid-Range"-Familien angeboten, die einen Mittelweg zwischen den beiden anderen Familien bieten (vgl. Tab. 9.4).

Durch die Fortschritte der Halbleitertechnologie steigt die Leistungsfähigkeit von Generation zu Generation an. So bieten aktuelle Low-Cost-Familien teilweise Eigenschaften an, die den High-Performance-Familien zurückliegender Generationen entsprechen. Tab. 9.5 fasst den Zeitpunkt der Einführung und die jeweils verwendete

Tab. 9.4 Auswahl einiger FPGA-Familien und ihre Marktpositionierung

Altera	Familie	Stratix	Arria	Cyclone
Altera	Positionierung	„High density, high performance"	„Balance of cost, power and performance"	„Low system cost plus performance"
Xilinx	**Familie**	**Virtex**	**Kintex**	**Artix**
Xilinx	Positionierung	„System performance"	„Price Performance with low power"	„System performance per Watt for cost sensitive applications"

Tab. 9.5 Zeitpunkt der Markteinführung und verwendete Halbleitertechnik der Stratix-Familie (Fa. Altera)

Generation/Name	Stratix	Stratix II	Stratix III	Stratix IV	Stratix V	Stratix 10
Jahr der Einführung	2002	2004	2006	2008	2010	2013
Halbleitertechnologie(nm)	130	90	65	40	28	14

Halbleitertechnologie für das Beispiel der High-Performance-Familie Stratix der Firma Altera zusammen. Die jeweils verwendeten Produktionstechnologien entsprechen in etwa denen, die auch bei der Produktion von anspruchsvollen Spitzenprodukten wie PC-Prozessoren, zum Einsatz kommen. Genauso wie bei PC-Prozessoren wird also auch bei der Produktion von FPGAs angestrebt, die neueste verfügbare Produktionstechnologie einzusetzen.

9.5.1 Vergleich ausgewählter FPGA-Familien

Innerhalb der Stratix-Familie werden unterschiedliche Bausteine angeboten. Eine Übersicht über die Eigenschaften dieser FPGAs ist in Tab. 9.6 zusammengefasst. Die Abkürzung *ALM (Adaptive Logic Module)* ist eine Hersteller-spezifische Abkürzung. Die wesentlichen Elemente eines ALM sind eine LUT mit 7 Eingängen, Logik für schnelle Addition und 4 Register.

Zum Vergleich zu der High-Performance-Familie Stratix 10 fasst Tab. 9.7 einige der Eigenschaften von Vertretern der Familie Cyclone V zusammen.

Die interne Speicherkapazität lässt sich relativ leicht, auch über FPGA-Generationen hinweg, vergleichen. Ein Vergleich der Logikelemente ist dagegen schwieriger, da der Aufbau der programmierbaren Grundelemente von Generation zu Generation wechseln kann. Ein einfacher Vergleich der Anzahl der ALMs ist nicht unbedingt zielführend, weil sich ALMs unterschiedlicher Generationen in ihrem Aufbau unterscheiden können. Für einen groben Vergleich unterschiedlicher Bausteine gibt die Firma Altera daher das Maß

Tab. 9.6 Übersicht über einige Eigenschaften von ausgewählten FPGAs der Stratix-10-Familie

Bezeichnung	GX500	GX1100	GX2500	GX5500
Anzahl ALMs	164.160	370.080	821.150	1.867.680
Anzahl Flip-Flops	656.640	1.480.320	3.284.600	7.470.720
Speicher (Mbit)	46	92	208	166
Arithmetik-Module für Signalverarbeitung	1.152	2.250	5.011	1.980
Multiplizierer (18 × 19 bit)	2.304	5.040	10.022	3.960
PCIe-Makros	1	2	6	3

Tab. 9.7 Übersicht über einige Eigenschaften von ausgewählten FPGAs der Cyclone-V-Familie

Bezeichnung	5CGXC3	5CGXC5	5CGXC7	5CGXC9
Anzahl ALMs	13.460	29.080	56.480	113.560
Anzahl Flip-Flops	53.840	116.320	225.920	454.240
Speicher (Mbit)	1,6	4,8	7,6	13,8
Arithmetik-Module für Signalverarbeitung	57	150	156	342
Multiplizierer (18 × 18 bit)	114	300	312	684
PCIe-Makros	1	2	2	2

Logic Elements (LE) an, das die verfügbaren ALMs in fiktive Grundelemente umrechnet. Einige der angebotenen FPGAs der Firma Altera sind auch als „SoC-Varianten“ verfügbar, die zusätzlich zum FPGA-Teil ein Multikern-CPU-Subsystem beinhalten.

Der Hersteller Xilinx verwendet zur Angabe der FPGA-Komplexität die Begriffe *Slice* beziehungsweise *Complex Logic Block (CLB).* Ein CLB der „Ultrascale“-FPGAs enthält beispielsweise 8 LUTs mit jeweils 6 Eingängen, Addiererlogik und 16 Flip-Flops. Einige Parameter von Bausteinen der Kintex- beziehungsweise Virtex-Ultrascale-Familie sind in Tab. 9.8 zusammengefasst.

Für besonders kostensensitive Systeme bietet Xilinx die Artix-7-Serie an. Diese ist ähnlich positioniert wie die Cyclone-Serie von Altera. Ebenso wie Altera bietet auch die Firma Xilinx Bausteine an, die CPU-Subsysteme enthalten. So enthält beispielsweise die Zynq-7000-Serie ein Subsystem, das auf einem Zweikern-System basiert, während mit der Zynq-Ultrascale + -Serie ein Prozessorsystem zum Einsatz kommt, das insgesamt 6 Prozessoren zur Verfügung stellt. Die CPUs dieser Serie werden durch Hard-Makros unterstützt, die für Beschleunigung von 3D-Grafik- oder Videofunktionen ausgelegt sind, so dass die anderen Ressourcen (FPGA-Teil oder Prozessoren) entlastet werden.

Obwohl in diesem Abschnitt bereits viele Zahlen präsentiert werden, welche die Eigenschaften kommerziell angebotener FPGA-Familien beschreiben, ist diese

Tab. 9.8 Übersicht über einige Eigenschaften von ausgewählten FPGAs der Kintex- und der Virtex-Ultrascale-Familie

	Kintex			Virtex		
Bezeichnung	KU035	KU060	KU115	XCVU065	XCVU125	XCVU440
Anzahl CLBs	25.391	41.460	82.920	44.760	89.520	316.620
Anzahl Flip-Flops	406.256	663.360	1.326.720	716.160	1.432.320	5.065.920
Block-RAM (Mbit)	19,0	38,0	75,9	44,3	88,6	88,6
Arithmetik-Module für Signalverarbeitung (DSP-Slices)	1.700	2.760	5.520	600	1.200	2.880
PCIe-Makros	2	3	6	2	4	6

Darstellung nur ein kleines Schlaglicht auf das umfangreiche Angebot der FPGA-Hersteller. Betrachtet man alleine die Anzahl der zur Verfügung gestellten Flip-Flops, so liegt beispielsweise zwischen dem kleinsten Baustein der Cyclone-V-Serie und dem größten Baustein der Stratix-10-Serie ein Faktor von etwa 140. Für den internen Speicher (Block-RAM) beträgt das Verhältnis etwa 100. Vergleichbare Faktoren ergeben sich auch für die Bausteine des Herstellers Xilinx.

Um die absoluten Zahlen einordnen zu können, kann folgendes Beispiel eines Systems zur Verarbeitung von Videosignalen dienen. Das System besitzt eine Kameraschnittstelle mit Anbindung zum externen Speicher, Module zur Verarbeitung der Bilder (zweidimensionale Filter) in Echtzeit sowie eine Ausgabeeinheit mit Speicheranbindung, die zur Anzeige der verarbeiteten Kamerabilder auf einem Monitor dient. Wird für die Implementierung dieses nicht ganz trivialen Systems ein Zynq-7000-SoC der Firma Xilinx eingesetzt, werden etwa jeweils 3000 LUTs und Flip-Flops benötigt. Selbst bei dem kleinsten in der Zynq-7000-Serie verfügbaren Baustein ist damit weniger als 20 % der FPGA-Ressourcen belegt.

Dieses Beispiel macht deutlich, dass viele der heutigen FPGAs nicht für den Ersatz von wenigen Gattern gedacht sind. Im Gegenteil: Sie ermöglichen die Realisierung hochkomplexer Systeme, für deren Realisierung noch vor wenigen Jahren ASICs erforderlich gewesen wären. Daher haben FPGAs den Einsatz von ASICs in vielen Bereichen ersetzt. Die seit einigen Jahren verfügbare Kombination von der „hardwareprogrammierbaren“ Logik mit leistungsfähigen „softwareprogrammierbaren“ Prozessor-Subsystemen eröffnet weitere Möglichkeiten für den Einsatz der FPGA-Technologie.

Die bisher betrachteten FPGAs zielen auf die Realisierung von wesentlichen Teilen eines Systems innerhalb der programmierbaren Logik. Ein anderer Ansatz wird mit den besonders kleinen, kostengünstigen und energieeffizienten FPGAs der Hersteller Lattice und Quicklogic verfolgt. So bietet beispielsweise Lattice die Serie Ice40 in den Varianten Ultra und UltraLite an. Diese Bausteine besitzen eine relativ geringe Anzahl von Logikblöcken und bieten nur wenig Speicherkapazität. Der entscheidende Vorteil dieser Bausteine ist die geringe statische Stromaufnahme, die im Bereich von 30 bis 70 µA liegt. Daher werden diese Bausteine bevorzugt in mobilen Geräten eingesetzt. Die FPGAs werden zum Teil als sogenannte *Glue Logic* verwendet, also zur Realisierung logischer Funktionen mit denen die Hauptkomponenten des Systems untereinander verbunden werden. Daneben kann mithilfe dieser FPGAs auch der Hauptprozessor des Systems, zum Beispiel bei Ein-/Ausgabe-Operationen, entlastet werden. Der Hauptprozessor kann so bereits in einen Stromsparmodus wechseln während das FPGA noch mit der Ein-/Ausgabe beschäftigt ist. Insgesamt wird so die Verlustleistung reduziert, da der relativ energiehungrige Hauptprozessor länger im Stromsparmodus verweilen kann.

Als ein exemplarischer Vertreter von besonders energieeffizienten FPGAs sind in Tab. 9.9 die wesentlichen Kennwerte der Ice40-Serie des Herstellers Lattice zusammengefasst.

Tab. 9.9 Eigenschaften von Low-Power FPGAs am Beispiel der Ice40-Serie

	UltraLite		Ultra		
Bezeichnung	UL640	UL1K	LP1K	LP2K	LP4K
Anzahl Logikblöcke	640	1248	1100	2048	3520
Block-RAM (kbit)	56	56	64	80	80
Multiplizierer	–	–	2	4	4
PLLs	1	1	1	1	1
Stat. Stromaufnahme (μA)	35	35	71	71	71

9.6 Hinweise zum Selbststudium

In vielen Fällen werden die Programme zum Entwurf von FPGA-Systemen in kostenlosen Varianten angeboten und können von Internetseiten der Hersteller heruntergeladen werden. Für die Bedienung der Software bieten die Hersteller Online-Tutorials, Trainingsvideos und eine umfangreiche Dokumentation an, die es ermöglichen, erste eigenständige Schritte im Bereich des VHDL-Entwurfs für programmierbare Logikbausteine durchzuführen.

Da der Entwurf einer FPGA-Platine eine herausfordernde Aufgabe ist, bieten sich für eigene Experimente fertige Boards an, die teilweise auch zu vergünstigten Preisen für Studierende und andere nicht-kommerzielle Nutzer angeboten werden. Für erste eigene Schritte bieten sich günstige Boards an, die bereits für deutlich unter 100 € zum Kauf angeboten werden.

Für die beiden Marktführer Xilinx und Altera bieten die Firmen Digilent (www.digilentinc.com) beziehungsweise Terasic (www.terasic.com) günstige Einsteigerboards an. Da diese Boards auch ein integriertes Programmiergerät besitzen, lassen sie sich ohne weitere Kosten für eigene Experimente verwenden.

Sehr interessant sind die Boards, die mit FPGAs ausgestattet sind, die auch ein CPU-Subsystem als Hardmacro beinhalten. Als ein Beispiel für ein solches Board ist das mit Xilinx-Baustein Zynq ausgestattete ZyBo-Board der Firma Digilent in Abb. 9.14 dargestellt. In der Mitte des Boards ist der FPGA-Baustein zu sehen. Darunter ist einer der beiden SDRAM-Speicher untergebracht. Dieses Board verfügt über viele Anschlussmöglichkeiten wie VGA, HDMI, Ethernet, USB, Audio, sowie Buchsen für die Anbindung eigener Hardware.

Mithilfe dieser Boards lassen sich auch erste Schritte im Bereich des FPGA-Enwurfs durchführen ohne das CPU-System zu nutzen. Später kann dann die Verwendung des CPU-Subsystems einbezogen werden. So können interessante Experimente bis hin zur Einbindung von eigener Hardware unter dem Betriebssystem Linux durchgeführt werden. Obwohl diese Boards mit einem Preis ab ca. 100 € etwas teurer sind als die einfachsten FPGA-Experimentierboards, kann sich die Anschaffung auf Grund der erweiterten Möglichkeiten lohnen.

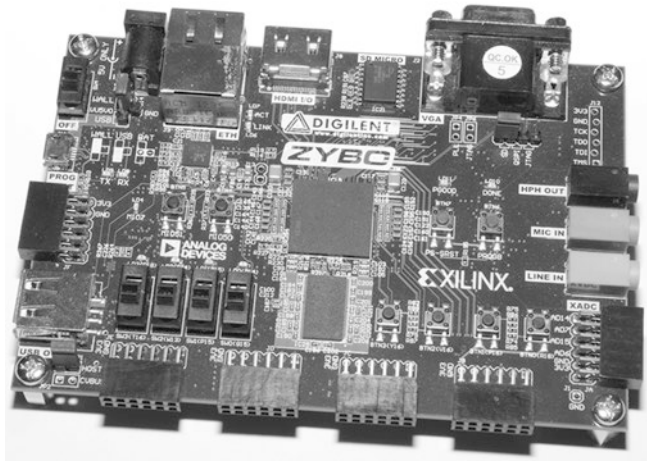


Abb. 9.14 Beispiel eines erschwinglichen FPGA-Boards für eigene Experimente: Das ZyBo-Board der Firma Digilent Inc.

9.7 Übungsaufgaben

Hier finden Sie Aufgaben, die einige Aspekte dieses Kapitels aufgreifen. Die Lösungen finden Sie am Ende des Buches.

Sofern nicht anders vermerkt, ist nur eine Antwort richtig.

Aufgabe 9.1
Welche Vorteile besitzen programmierbare Logikbausteine gegenüber logischen Standard-Komponenten beziehungsweise ASICs? *(Mehrere Antworten sind richtig)*

a) Die digitale Funktion programmierbarer Logikbausteine kann durch den Anwender festgelegt werden.
b) Designfehler lassen sich schneller korrigieren als dies bei dem Einsatz von ASICs möglich wäre.
c) Mithilfe Programmierbarer Logikbausteine können logische Funktionen kompakter realisiert werden als dies mit ASICs möglich wäre.
d) Mithilfe Programmierbarer Logikbausteine können logische Funktionen kompakter realisiert werden als dies mit Standardkomponenten (zum Beispiel 74er-Logikserie) möglich wäre.

Aufgabe 9.2
Wodurch zeichnen sich PAL-Bausteine aus?

a) Sie ermöglichen die Realisierung beliebig komplexer Funktionen.
b) Sie bieten eine höhere Komplexität als FPGAs.

c) Sie besitzen intern eine UND/ODER-Struktur.
d) Sie enthalten grundsätzlich keine Flip-Flops. Daher kann mit den Bausteinen immer nur eine Kombinatorik realisiert werden.

Aufgabe 9.3
Wodurch zeichnen sich CPLDs aus?

a) CPLDs sollten aus Kostengründen nur für Systeme eingesetzt werden, die in sehr hohen Stückzahlen gefertigt werden.
b) Im Vergleich zu PALs bieten CPLDs eine deutlich geringere Komplexität.
c) Sie enthalten grundsätzlich keine Flip-Flops. Daher kann mit den Bausteinen immer nur eine Kombinatorik realisiert werden.
d) Die Funktion der Schaltung kann mithilfe von VHDL beschrieben werden.

Aufgabe 9.4
Wodurch zeichnen sich FPGAs aus?

a) Typische FPGAs realisieren logische Funktionen auf Basis einer zweistufigen UND/ODER-Struktur.
b) Sie können nicht zur Realisierung von endlichen Automaten verwendet werden.
c) FPGAs realisieren logische Funktionen mithilfe von Lookup-Tabellen.
d) Alle FPGAs besitzen einen Mikroprozessor in Form eines Hardmacros.

Aufgabe 9.5
Wie viele unterschiedliche logische Funktionen können mit einer LUT mit 5 Eingängen realisiert werden?

a) 5
b) 25
c) 32
d) 64

Aufgabe 9.6
Welche der folgenden Komponenten sind in typischen FPGAs enthalten? *(Mehrere Antworten sind richtig)*

a) Spezialmodule für ausgewählte arithmetische Operationen, zum Beispiel Multiplizierer
b) Speicher
c) Spezialmodule zur Beschleunigung von 3D-Grafik-Anwendungen
d) Module zur Takterzeugung

Halbleitertechnik

10

Digitale Schaltungen werden als *Integrierte Schaltung* aufgebaut. Der Begriff Integrierte Schaltungen beschreibt, dass sich auf einem Stück Halbleiter nicht nur ein einzelner, sondern viele Transistoren befinden. Eine komplette Schaltung ist also auf dem Halbleiterkristall integriert. Ursprünglich umfasste eine Integrierte Schaltung einige tausend Transistoren; mittlerweile können über eine Milliarde Transistoren auf einer Fläche von etwa einem Quadratzentimeter zusammengefasst werden.

Für Integrierte Schaltungen sind verschiedene Begriffe gebräuchlich. Sie werden auch als Mikrochip, Chip, IC oder ASIC bezeichnet. IC steht für „Integrated Circuit", ASIC für „Application Specific Integrated Circuit" also Anwendungsspezifische Integrierte Schaltung.

Die wesentlichen Vorteile Integrierter Schaltungen sind insbesondere geringe Baugröße, geringe Kosten, hohe Geschwindigkeit und geringe Parameterabweichungen.

- Durch Verwendung integrierter Schaltungen kann die Baugröße eines Gerätes sehr gering sein. Statt mehrerer Bauelemente, die einzeln in Chipgehäusen verpackt sind, ist nur ein einzelnes Chipgehäuse erforderlich.
- Durch die Zusammenfassung mehrerer Bauelemente können fast immer die Kosten für ein elektronisches Gerät reduziert werden. Die wichtigsten Kostenvorteile sind dabei die geringere Anzahl an benötigten Bauelementen, kleinere und damit günstigere Platinen und Gerätegehäuse, sowie kostengünstigere Fertigung durch Verwendung von weniger Komponenten.
- In einer Schaltung mit geringerer Baugröße sind die Verbindungsleitungen zwischen den Transistoren wesentlich kürzer. Dadurch kann die Rechengeschwindigkeit der Schaltung erhöht werden, da wesentlich kleinere Kapazitäten umgeladen werden.
- Wenn sich die einzelnen Transistoren einer Schaltung auf demselben Halbleiterkristall befinden, haben die Transistoren nur sehr geringe Produktionsschwankungen zueinander.

W. Gehrke et al., *Digitaltechnik,* Springer-Lehrbuch,
DOI 10.1007/978-3-662-49731-9_10

10.1 CMOS-Technologie

Die für einen IC gewählte Schaltungstechnik wird als Chip-Technologie bezeichnet. Die zurzeit mit Abstand größte Marktbedeutung hat die *CMOS-Technologie*, die in diesem Kapitel erläutert wird.

Die CMOS-Technologie verwendet *Silizium* als Halbleitermaterial und das Hauptanwendungsgebiet sind digitale Schaltungen. Sie erlaubt eine sehr hohe Integrationsdichte. Das heißt, dass auf einem Chip sehr viele Transistoren untergebracht werden können. Auf Basis der CMOS-Technologie werden Computer-Prozessoren, Grafikkarten-ICs, Speicherbausteine, MP3-Decoder und viele andere ICs gefertigt.

Der Name CMOS steht für *Complementary Metal-Oxid-Semiconductor* und beschreibt das Grundprinzip. *Complementary,* also komplementär, meint zwei sich ergänzende Schaltungsteile, die zusammen einen digitalen Ausgangswert ergeben und *Metal-Oxid-Semiconductor* bezeichnet in diesem Zusammenhang einen bestimmten Typ von Feldeffekttransistoren.

Der Vorteil der CMOS-Technologie ist ihre relativ geringe Verlustleistung. Dies spart zum einen Energie, insbesondere bei mobilen Geräten wie Laptop oder Mobiltelefon. Ebenso wichtig ist aber zum anderen, dass die Schaltungen sich nicht zu stark erwärmen, denn die Verlustleistung muss vom Halbleiter auf das Chipgehäuse und von dort auf die Umgebung abgeführt werden.

Aktuelle Computer und ihre Grafikkarten werden durch große und manchmal störend laute Lüfter gekühlt. Die Aussage, CMOS-Schaltungen hätten eine geringe Verlustleistung, mag darum zunächst nicht offensichtlich sein. Allerdings enthält eine integrierte Schaltung etliche Millionen Transistoren, die mit hoher Geschwindigkeit Berechnungen durchführen. Nur durch die geringe Verlustleistung von CMOS-Schaltungen ist es überhaupt möglich, eine so hohe Integrationsdichte zu erreichen und die Verlustleistung in einem handhabbaren Rahmen zu halten.

10.1.1 Prinzipieller Aufbau

Der Aufbau und die Funktionsweise einer CMOS-Schaltung werden am Beispiel eines NAND-Gatters mit zwei Eingängen deutlich. Abb. 10.1 zeigt links den prinzipiellen Aufbau eines NAND-Gatters. Die zwei Eingänge *A* und *B* sind an insgesamt vier Schalter angeschlossen. Abhängig von dem Wert der Steuerleitung sind die Schalter geöffnet oder geschlossen. Dadurch verbinden sie den Ausgang *Y* entweder mit 0 oder 1.

Natürlich sind in Integrierten Schaltungen keine mechanischen Schalter, sondern Transistoren eingebaut. Es werden zwei Transistorarten verwendet. Der p-Kanal-Transistor leitet bei einer 0 (niedrige Spannung) am Eingang und sperrt bei einer 1 (hohe Spannung). Der n-Kanal-Transistor leitet dagegen bei einer 1 am Eingang und sperrt bei einer 0. Abb. 10.1 zeigt auf der rechten Seite das Schaltbild des NAND-Gatters. Die Masse wird als *Ground (GND)* bezeichnet. *VDD* ist die Versorgungsspannung (mit V für

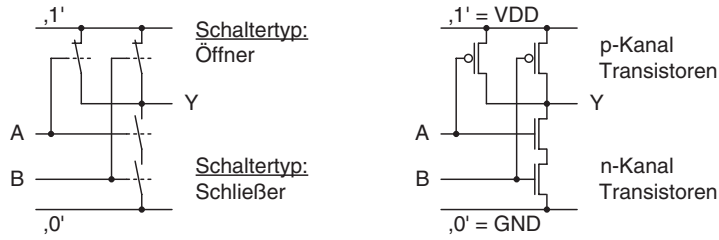


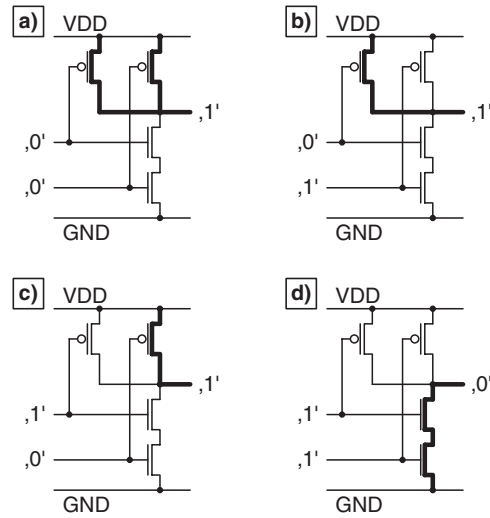
Abb. 10.1 Grundprinzip und reales Schaltbild eines NAND-Gatters

„Voltage“ und D für den Drain-Anschluss des Transistors). Typische Werte für die Versorgungsspannung sind zwischen 1,0 und 5,0 V.

Zur Erläuterung der Funktion zeigt Abb. 10.2 die möglichen Ansteuerungen der Eingänge. Die dick dargestellten Leitungen kennzeichnen welche Verbindungen leitend sind. Da beide Eingänge jeweils zwei Werte einnehmen können, existieren insgesamt vier Möglichkeiten der Ansteuerung.

- Im Fall a) sind beide Eingänge gleich 0. Dadurch leiten beide p-Kanal-Transistoren und der Ausgang wird niederohmig mit der Versorgungsspannung verbunden. Außerdem sperren die n-Kanal-Transistoren, so dass kein Kurzschluss von der Versorgungsspannung zur Masse entsteht.
- In den Fällen b) und c) ist ein Eingang 0, der andere 1 und einer der p-Kanal-Transistoren ist leitend, der andere sperrt. Durch die Parallelschaltung der p-Kanal-Transistoren ist auch hier eine Verbindung des Ausgangs zur Versorgungsspannung

Abb. 10.2 Vier Möglichkeiten der Ansteuerung eines NAND-Gatters



vorhanden; der Ausgang ist 1. Von den n-Kanal-Transistoren ist einer durch eine 1 am Eingang leitend. In der Reihenschaltung fließt jedoch kein Strom nach Masse.
- Im Fall d) sind beide Eingänge 1. Jetzt sind beide n-Kanal-Transistoren leitend und der Ausgang ist mit Masse verbunden, gibt also eine 0 aus. Die beiden p-Kanal-Transistoren sperren, so dass der Ausgang nicht mit Versorgungsspannung verbunden ist.

Die vier Eingangskombinationen ergeben somit die NAND-Funktion. In den vier möglichen Fällen zeigt sich die wichtige Eigenschaft der Schaltung, dass von den beiden Netzwerken aus p-Kanal und n-Kanal-Transistoren jeweils eins leitend, das andere gesperrt ist. Die Netzwerke verhalten sich also genau entgegengesetzt, was durch das ‚C' in CMOS, also den Begriff *komplementär,* ausgedrückt wird.

10.1.2 Feldeffekttransistoren

Feldeffekttransistoren werden sowohl nach n-Kanal und p-Kanal als auch nach selbstsperrend und selbstleitend unterschieden. Da in der CMOS-Technologie nur selbstsperrende Transistoren eingesetzt werden, sind nur diese im Folgenden erläutert. Sie werden auch als Anreicherungstyp oder Enhancement-Typ bezeichnet. Selbstleitende Transistoren (Verarmungstyp, Depletion-Typ) werden in der CMOS-Technologie nicht verwendet.

Das Grundmaterial, genannt *Substrat*, ist monokristallines Silizium, bei dem also die Silizium-Atome ein gleichmäßiges Gitter bilden. Dieses Material wird *dotiert*, das heißt, es werden kleine Mengen weiterer chemischer Elemente hinzugefügt. Je nach chemischem Element handelt es sich um eine *n-Dotierung* mit zusätzlichen Elektronen oder um eine *p-Dotierung* mit sogenannten Löchern, also Freistellen, so dass sich Elektronen bewegen können.

n-Kanal-Transistor

Der Aufbau eines Feldeffekttransistors ist in Abb. 10.3 als Schnittansicht von schräg oben dargestellt. Das Substrat ist leicht p-dotiert und in dieses Grundmaterial werden die beiden Anschlüsse Source und Drain durch n-Dotierung erzeugt. Zwischen den Anschlüssen liegt über einer Isolationsschicht der Gate-Anschluss. Die Isolationsschicht besteht meist aus Siliziumdioxid SiO_2 und der Gate-Anschluss aus polykristallinem Silizium, der durch eine hohe Dotierung gut leitet. *L* und *W* bezeichnen die Länge und Weite des Transistors. Sie sind wichtige Kenngrößen, denn aus ihnen ergeben sich die Größe und die Leitfähigkeit des Transistors.

Die Funktion des Feldeffekttransistors ist in Abb. 10.4 dargestellt. Zur einfacheren Darstellung ist die Seitenansicht gewählt. Im spannungslosen Zustand ist die Verbindung zwischen Source und Drain nicht leitend.

Bei Anlegen einer positiven Spannung an das Gate werden die p-Ladungsträger im Substrat, also die Löcher, verdrängt, denn gleichnamige Ladungen stoßen sich ab. Gleichzeitig werden n-Ladungsträger, also Elektronen, angezogen, denn ungleichnamige

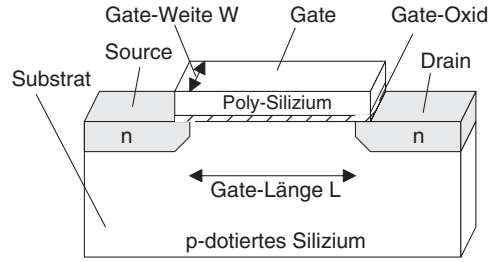


Abb. 10.3 Aufbau eines n-Kanal-Feldeffekttransistors

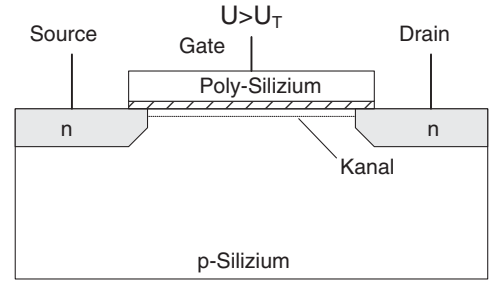


Abb. 10.4 Funktion des n-Kanal-Feldeffekttransistors

Ladungen ziehen sich an. Ab einer gewissen Spannung sind so viele Löcher verdrängt und Elektronen angezogen, dass sich ein Überhang von n-Ladungsträgern zwischen Source und Drain bildet. Dieser Bereich wird als *Kanal* bezeichnet. Mit dem Kanal bildet sich ein Gebiet, das zwischen Source und Drain durchgängig eine n-Dotierung besitzt, so dass der Transistor leitet.

Da die Leitfähigkeit durch einen *n-Kanal* entsteht, wird dieser Aufbau als *n-Kanal-Transistor* bezeichnet. Die Spannung, ab der ein Kanal entsteht, ist die Schwellenspannung U_T (T für „Threshold“, Schwelle). Der genaue Wert der Schwellenspannung ist unter anderem von der Dotierung abhängig.

p-Kanal-Transistor

Der Aufbau eines *p-Kanal-Transistors* ist im Prinzip der gleiche, allerdings sind die Dotierungen vertauscht (Abb. 10.5). Das Substrat ist n-dotiert und die Bereiche für Source und Drain haben eine p-Dotierung. Durch eine negative Spannung am Gate werden Elektronen abgestoßen und Löcher angezogen, so dass sich ab der Schwellenspannung ein *p-Kanal* bildet, der die p-Bereiche Source und Drain verbindet.

Die negative Gate-Spannung bedeutet dabei nicht, dass auf einem CMOS-Chip negative Spannungen verwendet werden. Die Gate-Spannung muss negativ gegenüber dem Bezugspotential des Substrats werden. Dies wird dadurch erreicht, dass beim p-Kanal-Transistor das Substrat an Versorgungsspannung gelegt wird. Eine Gate-Spannung von 0 Volt ist damit negativ gegenüber Substrat.

Ein Unterschied zum n-Kanal-Transistor besteht in den elektrischen Eigenschaften. Die Beweglichkeit der Löcher ist etwas geringer als die Beweglichkeit der Elektronen.

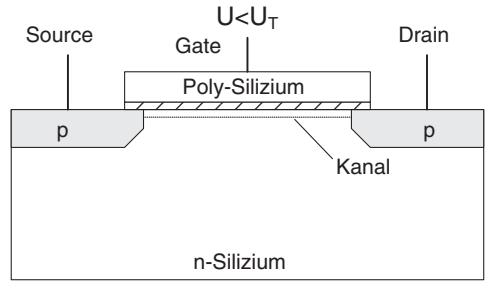


Abb. 10.5 Funktion des p-Kanal-Feldeffekttransistors

Deswegen ist der Widerstand eines p-Kanal-Transistors etwa 2- bis 3-mal so hoch wie bei einem p-Kanal-Transistor gleicher Größe. Als Ausgleich wird normalerweise ein p-Kanal-Transistor mit doppelter oder dreifacher Gate-Weite *W* (siehe Abb. 10.3) verwendet, wodurch beide Transistoren etwa gleichen elektrischen Widerstand haben.

10.1.3 Layout

Über den Transistoren befinden sich Verbindungsleitungen aus Metall. Für die vielen Verbindungen auf einem Chip sind mehrere Lagen an Verbindungsleitungen vorhanden. Moderne ICs haben etwa fünf bis zehn Lagen, wovon die unteren Lagen für lokale Verbindungen, die oberen Lagen für längere Verbindungen und die Spannungsversorgung verwendet werden. Zwischen den Verbindungsleitungen sowie zu den Transistoren sind Isolierschichten, die an vertikalen Verbindungsstellen durch Kontaktlöcher, sogenannte *Vias* unterbrochen sind. Abb. 10.6 zeigt die Transistorstruktur (gates) und die Verbindungslagen (M1 bis M4) im Elektronenmikroskop und gibt einen Eindruck von der realen Geometrie.

Der physikalische Aufbau einer CMOS-Schaltung wird als *Layout* bezeichnet. Das Layout beschreibt die Position der Transistoren sowie der Verbindungsleitungen. In Abb. 10.7 ist zunächst das Layout eines einzelnen Transistors gezeigt. Links sieht man die Seitenansicht, wie im vorherigen Abschnitt erläutert. Dabei sind Source und Drain durch Metalllage und Kontaktloch angeschlossen. Rechts ist die Draufsicht gezeigt, die für das Layout verwendet wird. Dabei wird in der Darstellung nicht zwischen Source und Drain unterschieden.

Das Layout eines kompletten Gatters ist in Abb. 10.8 dargestellt. Es handelt sich um das oben beschriebene NAND-Gatter. Zur Orientierung ist das Schaltbild noch einmal angegeben. Im Layout sind oben und unten Metallleitungen für die Anschlüsse von Versorgungsspannung *(VDD)* und Masse *(GND)* vorhanden.

Die beiden n-Kanal-Transistoren befinden sich im unteren Bereich des Layouts und sind in Reihe geschaltet. Zwischen den Transistoren ist keine zusätzliche Verbindung nötig. Das Drain-Gebiet des einen Transistors ist direkt das Source-Gebiet des anderen Transistors. Ein Anschluss dieser Reihenschaltung ist an *GND,* der andere am Ausgang *Y.* Die Gate-Anschlüsse sind mit den Eingängen des NAND-Gatters, *A* und *B* verbunden.

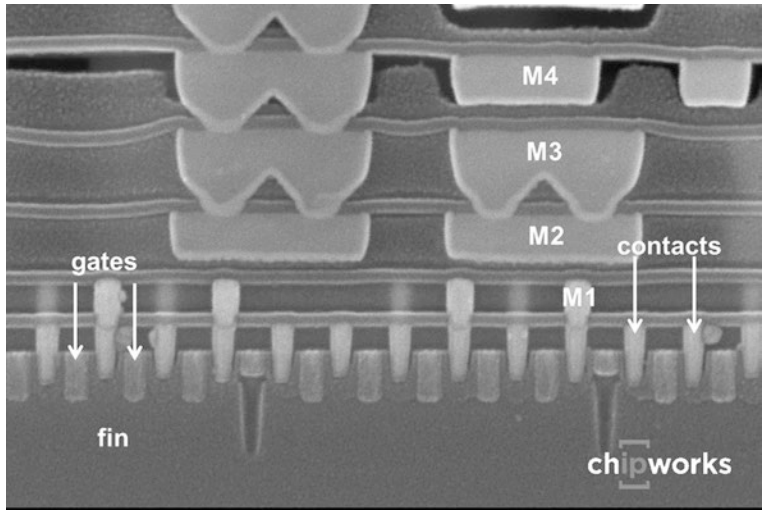
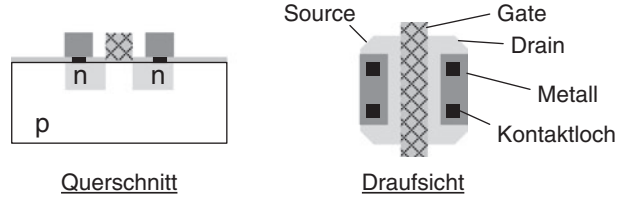


Abb. 10.6 Transistor im Elektronenmikroskop. (Foto: Chipworks)

Abb. 10.7 Layout eines Transistors



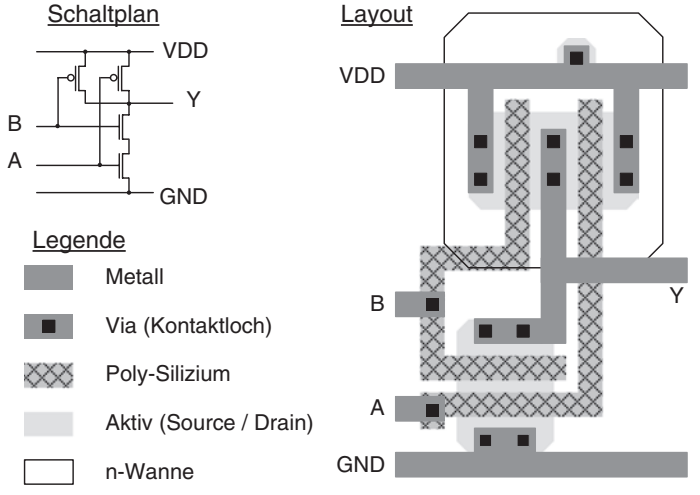


Abb. 10.8 Layout eines NAND-Gatters

Im oberen Bereich des Layouts sind die beiden p-Kanal-Transistoren. Sie sind parallel geschaltet und verbinden jeweils *VDD* mit dem Ausgang *Y*. Auch sie werden durch die Eingänge *A* und *B* angesteuert. Wie oben erläutert, benötigen die p-Kanal-Transistoren ein anderes Substrat und dies wird durch die sogenannte n-Wanne bereitgestellt. Die n-Wanne ist ein Bereich, in dem das eigentlich p-dotierte Grundmaterial durch Dotierung in einen n-Bereich umgewandelt wird. Durch das Kontaktloch ganz oben an der *VDD*-Leitung wird die n-Wanne mit dem Pegel der Versorgungsspannung verbunden.

Im Layout sind auch Länge und Weite des Gates dargestellt. Die *Gate-Länge* wird so kurz wie möglich gewählt, damit der Widerstand durch den Transistor nicht unnötig groß wird. Die Weite wird so gewählt, dass n-Kanal und p-Kanal-Netzwerke den gleichen Widerstand und damit symmetrisches Verhalten haben. Beim NAND-Gatter sind zwei n-Kanal-Transistoren in Reihe, was den doppelten Widerstand ergibt. Die p-Kanal-Transistoren haben aufgrund der geringeren Beweglichkeit der Löcher ebenfalls etwa doppelten Widerstand. Somit sind die Widerstände beider Transistornetzwerke etwa gleich groß.

10.2 Grundschaltungen in CMOS-Technik

In diesem Abschnitt wird für einige Grundschaltungen der Aufbau in CMOS-Technik erläutert. Das Ziel ist dabei, dass Sie sich vorstellen können, wie Digitalschaltungen aus Transistoren aufgebaut werden.

10.2.1 Inverter

Der Inverter ist noch einfacher aufgebaut als das NAND-Gatter und besteht aus nur zwei Transistoren. Ein Transistor verbindet den Ausgang mit *VDD,* ein anderer mit *GND.* Schaltbild und Layout sind in Abb. 10.9 dargestellt. Die Gate-Weite des p-Kanal-Transistors (oben) ist doppelt so groß wie beim n-Kanal-Transistor, um die geringere Beweglichkeit der Löcher auszugleichen.

10.2.2 Logikgatter

Andere Grundgatter können in ähnlicher Weise wie das NAND-Gatter mit n- und p-Kanal-Transistoren aufgebaut werden. Ein Netzwerk von n-Kanal-Transistoren verbindet den Ausgang mit Masse, ein zweites Netzwerk von p-Kanal-Transistoren verbindet den Ausgang mit der Versorgungsspannung. Dabei ist wichtig, dass die Netzwerke zueinander komplementär sind, also stets genau eins der Netzwerke leitet.

Das Beispiel in Abb. 10.10 hat die Funktion $Y = \overline{(A \vee B) \,\&\, C}$. Man erkennt, dass die Netzwerke auch in ihrer Topologie komplementär sind. Im p-Kanal-Netzwerk sind die

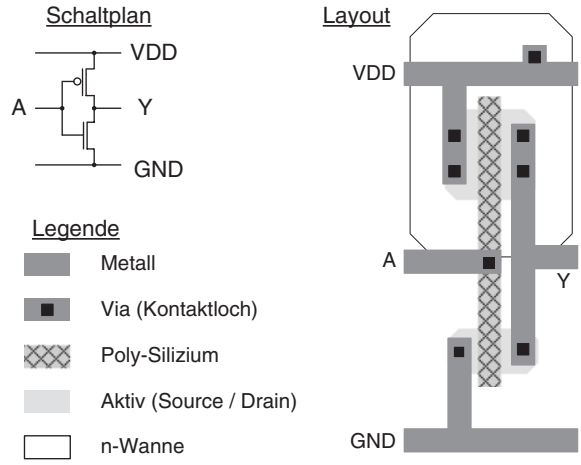


Abb. 10.9 Schaltbild und Layout eines Inverters

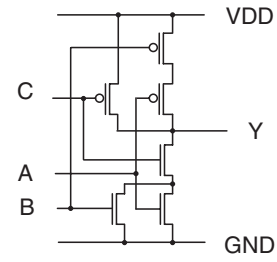


Abb. 10.10 Komplexgatter in CMOS-Technik

Transistoren für *A* und *B* in Reihe und *C* ist parallel dazu. Im n-Kanal-Netzwerk sind *A* und *B* parallel geschaltet und *C* liegt in Reihe dazu.

Nach dem gezeigten Grundprinzip lassen sich viele weitere Logikgatter entwerfen. Ein Kennzeichen von CMOS-Logikgattern ist, dass Funktionen mit einer Invertierung einfacher zu implementieren sind. Dies bedeutet, dass beispielsweise die NAND-Funktionen einfacher als eine UND-Funktion aufgebaut sein kann, denn die NAND-Funktion nutzt die Eigenschaft, dass eine 0 die Transistoren nach *VDD* öffnet.

Für ein Gatter ohne Invertierung wird ein Inverter angefügt. Ein UND-Gatter besteht beispielsweise aus dem NAND-Gatter (Abb. 10.1), ergänzt um den Inverter aus Abb. 10.9. Die Schaltung benötigt 6 Transistoren, vier für das NAND, zwei für den Inverter. Im Layout werden die beiden Schaltungsteile kombiniert, um wenig Fläche zu belegen.

10.2.3 Transmission-Gate

In den bisher gezeigten Grundgattern verbinden die Transistoren den Ausgang mit *VDD* oder *GND*. Es ist jedoch auch möglich, Signaleingänge durch die Transistoren zu

leiten oder zu sperren. Die entsprechende Schaltungsstruktur wird als *Transmission-Gate* bezeichnet und ist in Abb. 10.11, links dargestellt. Ein n-Kanal und ein p-Kanal-Transistor sind parallel geschaltet und geben abhängig vom Steuersignal *EN* den Eingang auf den Ausgang weiter. Da die Transistoren bei unterschiedlichem Pegel der Steuersignale leiten, ist das Signal *EN* in positiver und negativer Polarität erforderlich.

Der Vorteil dieser Schaltungsstruktur ist der geringe Schaltungsaufwand. Manche Funktionen lassen sich mit deutlich weniger Transistoren umsetzen, als bei der Struktur mit komplementären Transistornetzwerken nötig wäre. Der Nachteil der Struktur ist, dass ein Transmission-Gate keine Treiberfähigkeit besitzt. Dies ist jedoch meist kein Problem, denn der Treiber des Eingangssignals kann üblicherweise ein oder sogar mehrere Transmission-Gates treiben. Falls die Treiberfähigkeit nach dem Transmission-Gate zu gering ist, kann ein Inverter als Treiber eingefügt werden.

Vielleicht haben Sie beim Blick auf Abb. 10.11 überlegt, ob nicht ein Transistor als Transmission-Gate ausreichen würde. Dies ist ungünstig, denn der n-Kanal-Transistor schaltet eine 0 mit vollem Pegel, reduziert aber eine 1 um die Schwellenspannung. Umgekehrt schaltet der p-Kanal-Transistor die 1 mit vollem und die 0 mit reduziertem Pegel. Erst die Kombination beider Transistoren gibt ein gutes Schaltverhalten.

Ein Logikgatter, welches die Transmission-Gate-Struktur verwendet, ist in Abb. 10.11, rechts zu sehen. Es handelt sich um einen 1-aus-2-Multiplexer mit Steuereingang *S* und Dateneingängen *A* und *B*. Die Dateneingänge sind jeweils durch ein Transmission-Gate mit dem Ausgang *Y* verbunden. Da die Ansteuerung für die Transmission-Gates unterschiedliche Polarität hat, ist genau ein Gate geöffnet, das andere sperrt. Die Schaltung benötigt nur sechs Transistoren, zwei für den Inverter und vier in den Transmission-Gates, und ist damit sehr kompakt.

10.2.4 Flip-Flop

Neben kombinatorischen Elementen enthält eine Digitalschaltung natürlich auch Flip-Flops zur Speicherung von Informationen. Heutzutage werden praktisch immer taktflankengesteuerte D-Flip-Flops verwendet. Es gibt verschiedene Schaltungstechniken, um ein solches Flip-Flop zu realisieren. Die Varianten unterscheiden sich in Siliziumfläche, Schaltgeschwindigkeit und Stromverbrauch.

Als eine Flip-Flop-Schaltung ist in Abb. 10.12 exemplarisch das *Transmission Gate Pulsed Latch,* kurz TGPL, dargestellt. Zum Verständnis der Schaltung ist ein kleines

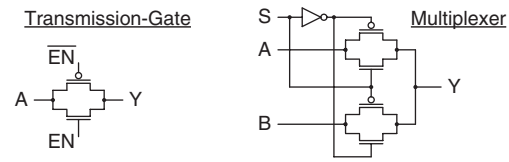


Abb. 10.11 Transmission-Gate und Anwendung in einem Multiplexer

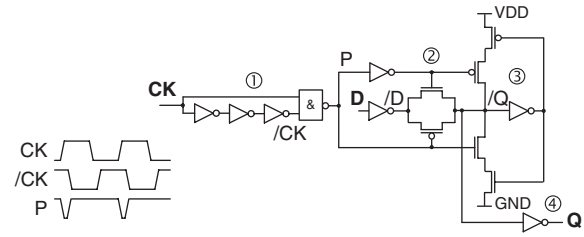


Abb. 10.12 Schaltbild des Transmission Gate Pulsed Latch. (Quelle: M. Alioto, IEEE Transactions on VLSI Systems, 2011)

Zeitdiagramm angegeben sowie einige interne Signalknoten mit Bezeichnung markiert. Dateneingang *D,* Takteingang *CK* sowie Datenausgang *Q* sind fett dargestellt.

Funktionsweise des Transmission Gate Pulsed Latch (TGPL):

1. Das TGPL enthält auf der linken Seite eine Taktaufbereitung. Der Takteingang *CK* wird durch drei Inverter verzögert und in der Polarität gedreht (Signal */CK*, siehe Zeitdiagramm). Durch ein NAND-Gatter werden dann *CK* und */CK* verknüpft und das Pulssignal *P* entsteht. *P* ist meist 1 und wird nur bei einer steigenden Taktflanke kurz 0. Damit steuert dieses Pulssignal die Datenübernahme an der Taktflanke.
2. Der Dateneingang *D* läuft zunächst durch einen Inverter, der als Treiber dient. */D* wird gespeichert, indem das Pulssignal *P* beim Pegel 0 ein Transmission-Gate öffnet. Das gespeicherte Datensignal liegt dann am internen Knoten */Q* an. Durch den Eingangsinverter hat es die umgekehrte Polarität. Kurz nach der Taktflanke wechselt *P* wieder auf den Wert 1 und schließt das Transmission-Gate.
3. Jetzt wird zur Datenspeicherung der interne Knoten */Q* durch den Inverter und die vier Transistoren auf der rechten Seite der Schaltung wieder nach */Q* gegeben. Dieser Schaltungsteil ist eine Rückkopplung, die den Wert an */Q* speichert. Nur wenn *P* auf 0 ist, also bei einer steigenden Taktflanke, unterbricht die Rückkopplung, damit ein neuer Eingangswert *D* gespeichert werden kann.
4. Durch den Inverter rechts unten wird der interne Knoten */Q* auf den Datenausgang *Q* gegeben. Die Invertierung am Eingang wird wieder durch den Ausgangsinverter aufgehoben, so dass der richtige Signalwert ausgegeben wird.

Für die sichere Funktion muss das Zeitverhalten der Schaltung genau abgestimmt werden. Die Laufzeit der drei Inverter bei ① muss ein Pulssignal P erzeugen, welches den Eingang *D* sicher übernimmt. Andererseits sollte das Pulssignal auch nicht zu lange 0 sein, denn während dieser Zeit darf sich *D* nicht ändern. Die Dauer des Pulssignals bestimmt also Setup- und Hold-Zeit.

Außerdem muss die Verzögerungszeit von Transmission-Gate bei ②, sowie Rückkopplung bei ③ zueinander passen, damit die Schaltungsteile sicher zusammenarbeiten.

Dieses Zeitverhalten muss bei allen Variationen der Arbeitsbedingungen sicher funktionieren. Als Variationen der Arbeitsbedingungen sind drei Einflussgrößen zu beachten, die unter der Abkürzung *PVT* zusammengefasst sind:

- **Process (P):** Die elektrischen Eigenschaften der Transistoren unterliegen den Toleranzen des verwendeten Halbleiterprozesses und können schwanken. Beispiel: Die Dotierung von Source und Drain kann gegenüber dem „Normalfall“ abweichen.
- **Voltage (V):** Die Versorgungsspannung kann, eventuell auch nur kurzzeitig, schwanken. Beispiel: Statt ideal 1,2 V kann die Spannung 1,15 oder 1,25 V betragen.
- **Temperature (T):** Die Temperatur kann schwanken. Beispiel: Der Chip kann bei Temperaturen im Bereich von −20°C bis 80°C arbeiten.

Bei der Entwicklung eines Flip-Flops wird die Schaltung darum unter verschiedenen Arbeitsbedingungen simuliert und es werden Testschaltungen hergestellt. Dabei kann auch überprüft werden, ob eventuell eine andere Flip-Flop-Schaltung für den jeweiligen Halbleiterprozess besser geeignet ist. Das oben beschriebene TGPL ist nur eine mögliche Schaltungsvariante.

10.3 Verlustleistung

Neben der Anzahl an Transistoren, welche die Größe einer Schaltung ausmacht, ist der Energieverbrauch einer Schaltung eine wichtige Kenngröße. Die CMOS-Technik ist prinzipiell sehr energieeffizient. Sie hat gegenüber anderen Halbleitertechniken den großen Vorteil, dass durch ein Gatter kein Ruhestrom fließt, denn entweder sperren die p-Kanal-Transistoren oder die n-Kanal-Transistoren. Vorgängertechnologien hingegen hatten einen ständigen Ruhestrom und wurden wegen dieser ständigen Verlustleistung durch CMOS abgelöst.

Durch immer leistungsfähigere Schaltungen ist allerdings auch die Verlustleistung von CMOS-Schaltungen in den letzten Jahren immer weiter gestiegen. Deutlich sichtbar ist dies bei High-End-Grafikkarten für PC-Spiele. Sie haben hohe Rechenleistung für die Berechnung der Grafik, aber auch große Kühlkörper und Lüfter zur Kühlung.

Es gibt verschiedene Gründe aus denen eine geringe Verlustleistung Integrierter Schaltungen sinnvoll ist.

- Höhere Leistungsaufnahme erhöht die Kosten für Chipgehäuse und Kühlkörper. Gegebenenfalls sind Lüfter erforderlich.
- Die Betriebskosten für Spannungsversorgung und Kühlung steigen. Dies ist insbesondere in Rechenzentren ein hoher Kostenanteil.
- Mobile Geräte wie Laptop, Tablet oder Smartphone sollen mit einer Akkuladung möglichst lange Betriebszeiten haben.
- Es werden autarke Sensoren eingesetzt, die mit einer Batterie mehrere Jahre betrieben werden sollen.

Die Verlustleistung entsteht durch einen statischen und einen dynamischen Anteil. Diese beiden Aspekte der Verlustleistung werden in den folgenden Abschnitten näher vorgestellt.

10.3.1 Statische Verlustleistung

CMOS-Schaltungen haben zwar keinen Ruhestrom, der durch einen geöffneten Transistor fließt. Dennoch fließen winzige sogenannte *Leckströme*, da der Transistor natürlich keine galvanische Trennung des Stromflusses vornimmt. Diese Leckströme addieren sich über die Milliarden Transistoren eines Chips und verursachen eine *statische Verlustleistung*.

Leckströme entstehen an verschiedenen Stellen des Transistoraufbaus. Insgesamt gibt es vier Anteile, die in Abb. 10.13 dargestellt sind (vergleiche Abb. 10.4 und 10.5). Der Anschluss B ist dargestellt, da auch über das Substrat (Bulk) Leckströme fließen können.

- *Subthreshold Leakage* I_{subth} entsteht, da der Kanal nicht vollständig ausgeschaltet werden kann.
- *Gate Leakage* I_{gate} ergibt sich auf Grund von Ladungsträgerübertragung durch sehr dünnes Gate-Oxyd.
- *Reverse Bias Junction Leakage* I_{rev} ist der Sperrstrom des pn-Übergangs zum Substrat.
- *Gate Induced Drain Leakage* I_{gidl} ist der Leckstrom vom Drain-Anschluss, verursacht durch die Feldstärke der Drain-Spannung.

Der Hauptanteil der statischen Verlustleistung entsteht durch die *Subthreshold Leakage* I_{subth}. Einen geringeren Anteil tragen *Gate Leakage* I_{gate} und *Reverse Bias Junction Leakage* I_{rev} bei. Die *Gate Induced Drain Leakage* I_{gidl} ist normalerweise vernachlässigbar. Allgemein führt eine erhöhte Temperatur zu steigenden Leckströmen.

Die *Subthreshold Leakage* ist exponentiell von der Schwellenspannung abhängig. Je höher die Schwellenspannung, umso geringer sind die Leckströme. Andererseits reduziert eine höhere Schwellenspannung auch die Verarbeitungsgeschwindigkeit, so dass ein Kompromiss gefunden werden muss.

10.3.2 Dynamische Verlustleistung

Die *dynamische Verlustleistung* entsteht bei Aktivität der Schaltung. Zum Verständnis wird die Inverter-Schaltung aus Abschn. 10.2.1 erneut betrachtet. Abb. 10.14 zeigt den Inverter sowie Spannungen und Ströme bei Schaltungsaktivität. Zusätzlich zu den beiden Transistoren ist ein Kondensator mit der Kapazität C_L abgebildet. Dieser stellt die

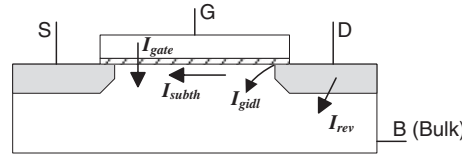


Abb. 10.13 Leckströme bei einem CMOS-Transistor

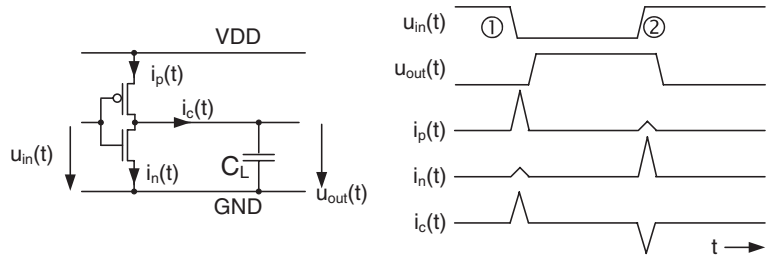


Abb. 10.14 CMOS-Inverter mit Zeitverlauf von Spannungen und Strömen

Lastkapazität dar, welche vom Inverter geschaltet wird. Die Lastkapazität setzt sich zusammen aus den Gate-Kapazitäten der nachfolgenden Gatter sowie der Leitungskapazität auf den Verbindungen dorthin. Der Zeitverlauf zeigt den prinzipiellen Verlauf der Spannungen am Eingang und Ausgang des Inverters sowie der Ströme im p-Kanal-Transistor $i_p(t)$, im n-Kanal-Transistor $i_n(t)$, sowie zum Kondensator $i_c(t)$.

Im Diagramm wechselt die Eingangsspannung zum Zeitpunkt ① von logisch 1 auf 0. Mit kurzer Zeitverzögerung wechselt darauf der Ausgang von 0 nach 1. Dabei wird der Kondensator über den p-Kanal-Transistor geladen, sichtbar an den Strömen i_p und i_c. Außerdem fließt ein kleinerer *Querstrom* über i_p und i_n, wenn beim Umschalten des Eingangs beide Transistoren für kurze Zeit teilweise leiten.

Zum Zeitpunkt ② wechselt der Eingang wieder von 0 auf 1 und der Ausgang kurz darauf von 1 nach 0. Jetzt wird der Kondensator über den n-Kanal-Transistor entladen, sichtbar an den Strömen i_n und einem negativen Wert für i_c. Wieder sind beim Umschalten kurzfristig beide Transistoren teilweise leitend, so dass erneut ein Querstrom über i_p und i_n fließt.

Die Verlustleistung des Inverters berechnet sich über das Integral des Stroms $i_p(t)$, multipliziert mit der Versorgungsspannung V_{DD}. Dabei hat das Umladen der Kapazität den größten Anteil. Einflussgrößen sind zum einen der Wert der Lastkapazität C_L sowie die Höhe der Versorgungsspannung V_{DD}. Zum anderen muss berücksichtigt werden, wie oft die Kapazität umgeladen wird. Dies wird durch die Taktfrequenz f der Schaltung angegeben, sowie die *Schaltaktivität* σ als Wahrscheinlichkeit einer 0-1-Flanke pro Taktzyklus.

Diese Einflussgrößen multiplizieren sich zur Verlustleistung P_C für das Umladen der Lastkapazität. Dabei hat die Versorgungsspannung einen quadratischen Einfluss:

$$P_{C,inv} = \sigma f \, V_{DD}^2 C_L$$

Die Einflussgrößen sind, ausgenommen die Schaltaktivität, bereits bekannt. Die Schaltaktivität drückt aus, wie häufig eine Leitung auf 1 wechselt und kann Werte zwischen 1 und 0 einnehmen.

- Das Taktsignal hat jeden Takt eine steigende Flanke und daher ist $\sigma = 1$
- Die unterste Stelle eines Zählers hat von Takt zu Takt abwechselnd die Werte 0 und 1. Es gibt also jeden zweiten Takt eine steigende Flanke: $\sigma = 0{,}5$

- Die oberste Stelle eines 8-Bit-Zählers hat nur eine steigende Flanke beim Übergang von 127 nach 128. Der nächste Wechsel tritt erst 256 Takte später auf: $\sigma = 1/256 \approx 0{,}004$
- Die Reset-Leitung einer CPU wird im normalen Betrieb nicht angesteuert, daher ist $\sigma = 0$
- Audio und Video-Signale haben, je nach Typ des Signals, einen Wert von $\sigma \approx 0{,}3$ bis 0,1

Für eine gesamte Integrierte Schaltung müssen die Anteile der einzelnen Schaltungsknoten addiert werden.

$$P_C = \sum_{i = alle\ Knoten} \sigma_i f\, V_{DD}^2\, C_{L,i}$$

Die Verlustleistung durch den Querstrom ist in dieser Gleichung noch nicht berücksichtigt. Allerdings ist der Anteil deutlich kleiner als P_C und ebenfalls proportional zur Schalthäufigkeit. Darum wird in der Praxis meist nur die Verlustleistung durch Umladen der Lastkapazitäten betrachtet. Der Einfluss des Querstroms kann beispielsweise berücksichtigt werden, indem die Lastkapazitäten C_L etwas höher angesetzt werden.

10.3.3 Entwurf energieeffizienter Schaltungen

Um Schaltungen mit geringer Verlustleistung zu entwerfen, werden möglichst alle Einflussgrößen optimiert. Ein Faktor ist die Versorgungsspannung, die früher bei 5 Volt lag und heute bis auf Werte von etwa 1 Volt reduziert wurde. Dies ist ohnehin erforderlich, damit die Feldstärken in den kleiner werdenden Transistoren nicht zu stark ansteigen. Durch die geringere Versorgungsspannung reduzieren sich statische und dynamische Verlustleistung.

Die statische Verlustleistung kann durch Wahl der Parameter des Halbleiterprozesses, also Transistorgeometrie und Dotierungsstärken reduziert werden. Weil dadurch auch die Geschwindigkeit einer Schaltung sinkt, kann ein Hersteller einen Halbleiterprozess in verschiedenen Varianten anbieten. Beispielsweise kann eine Version angeboten werden, die im Hinblick auf die Schaltgeschwindigkeit optimiert ist. Weitere Varianten des Halbleiterprozesses könnten eine Low-Power-Version oder eine „balancierte Version“, die einen Kompromiss aus Rechenleistung und Stromverbrauch darstellt, sein.

Die dynamische Verlustleistung kann reduziert werden, indem eine geringere Kapazität C_L umgeladen wird. Dies kann durch einen Prozess mit geringeren physikalischen Abmessungen erfolgen. Aber auch eine geringere Anzahl an Schaltungselementen reduziert die Anzahl an Schaltungsknoten und damit die Lastkapazität. Eine Möglichkeit ist beispielsweise, wenn eine Rechenoperation nur eine Genauigkeit von 16 bit anstatt 32 bit erfordert.

Als weitere Einflussgröße kann eine geringere Häufigkeit der Signalwechsel die dynamische Verlustleistung reduzieren. Eine Möglichkeit hierfür ist das Abschalten

ganzer Schaltungsteile, wenn sie nicht benötigt werden. Dies erfolgt beispielsweise in einer CPU mit mehreren Prozessoren. Bei geringer Rechenlast werden einzelne Prozessoren komplett ausgeschaltet und damit die dynamische Verlustleistung reduziert. Wenn ein Prozessor oder nicht benötigte Schnittstellenkomponenten vorübergehend von der Versorgungsspannung abgetrennt werden, reduziert sich zusätzlich auch die statische Verlustleistung.

10.4 Integrierte Schaltungen

Eine komplette Integrierte Schaltung setzt sich aus vielen einzelnen Gattern und Flip-Flops zusammen.

10.4.1 Logiksynthese und Layout

Standardzellbibliothek
Die in Abschn. 10.2 beschriebenen Grundschaltungen werden vom Hersteller eines Halbleiterprozesses in einer Bibliothek zur Verfügung gestellt. Diese Grundschaltungen werden als *Standardzellen* bezeichnet. Eine *Standardzellbibliothek* umfasst beispielsweise 100 bis 200 Zellen, darunter:

- Inverter und Treiber, also nacheinander geschaltete Inverter, für größere Lastkapazitäten
- Logikgatter, also UND-, ODER-, NAND-, NOR-, XOR-Gatter mit unterschiedlicher Anzahl an Eingängen
- Komplexgatter, für kombinierte Logikfunktionen, beispielsweise die Funktion $Y = \overline{(A \vee B) \,\&\, C}$ aus Abb. 10.10 oder der Multiplexer aus Abb. 10.11
- Arithmetische Schaltungen, beispielsweise Volladdierer
- Flip-Flops in verschiedener Konfiguration, beispielsweise mit Set oder Reset

Außerdem können für manche Zellen Varianten mit verschiedener Treiberstärke vorhanden sein. Ein Flip-Flop, das nur ein weiteres Gatter ansteuert, benötigt einfache Treiberstärke. Falls mehrere Gatter angesteuert werden, könnte die vierfache Treiberstärke sinnvoll sein.

Logiksynthese
Die Auswahl der passenden Standardzelle erfolgt normalerweise durch ein EDA-Programm. Dazu schreiben Sie VHDL-Code und das Programm sucht dann die passende Standardzelle für die beschriebene Funktion. Anhand der Verbindungen zu weiteren Standardzellen, entscheidet das Programm auch, welche Treiberstärke eingesetzt werden soll. Dieser Schritt wird als Logiksynthese bezeichnet.

Beispielsweise wurde im Kapitel 6 eine Flankenerkennung beschrieben, bei der folgender VHDL-Code verwendet wurde:

```
if (a_sync_old='0') and (a_sync='1') then
   q <= '1';      else
   q <= '0';      end if;
```

Die Logiksynthese interpretiert diesen Code und erkennt, dass eine Logikfunktion $\bar{A}\,\&\,B$ erforderlich ist. A ist dabei das VHDL-Signal *a_sync_old, B* ist *a_sync*. Für die Umsetzung in Standardzellen hat die Logiksynthese mehrere Möglichkeiten:

- Inverter für A gefolgt von einem UND-Gatter.
- Da Grundgatter in CMOS-Technologie stets eine Invertierung beinhalten, wäre ein NAND- oder NOR-Gatter vorteilhaft. Die Logikfunktion kann mit den Gesetzen von De Morgan umgewandelt werden in $\bar{A}\,\&\,B = \overline{\left(A \vee \bar{B}\right)}$. Damit ergibt sich ein Inverter für B gefolgt von einem NOR-Gatter.
- Eventuell steht in der Standardzellbibliothek ein passendes Komplexgatter mit der Funktion $\bar{A}\,\&\,B$ zur Verfügung.

Dieser Entwurfsschritt ist ähnlich zur in Kapitel 7 beschriebenen Synthese von FPGA-Schaltungen. Allerdings muss die Logiksynthese unter vielen Standardzellen wählen, während der FPGA-Synthese üblicherweise nur Look-Up-Tables und Flip-Flops zur Verfügung stehen.

Layout

Die Logiksynthese erzeugt eine Netzliste mit benötigten Standardzellen und ihren Verbindungsleitungen. Im nächsten Schritt werden die Position der Standardzellen und die Lage der Verbindungsleitungen ermittelt. Die physikalische Anordnung wird als Layout, die beiden Einzelschritte als Placement und Routing bezeichnet. Auch diese Schritte werden von einem EDA-Programm durchgeführt und sind ähnlich zur Platzierung und Verdrahtung des FPGA-Entwurfs.

Miteinander verbundene Standardzellen werden vom EDA-Programm möglichst nah aneinander platziert. Dazu probiert ein intelligenter Algorithmus verschiedene Anordnungen aus. Abb. 10.15 zeigt das Layout einer automatisch erzeugten Teilschaltung.

Aus den Teilschaltungen wird schließlich die gesamte integrierte Schaltung zusammengestellt. Abb. 10.16 zeigt als Beispiel das Chip-Foto eines System-on-Chip (SoC) für ein Smartphone. Es handelt sich um die zentrale Steuereinheit des Geräts mit zwei CPU-Kernen und der Grafikerzeugung (GPU) sowie lokalem Speicher (L1, L2, SRAM). Ebenso sind verschiedene Schnittstellen für externen Speicher (DRAM), Kamera, USB und das Display (LCD) vorhanden. Für die Taktaufbereitung dienen PLLs (Phase-Locked Loop). Der Chip enthält über 1 Mrd. Transistoren auf rund 1 Quadratzentimeter Fläche.

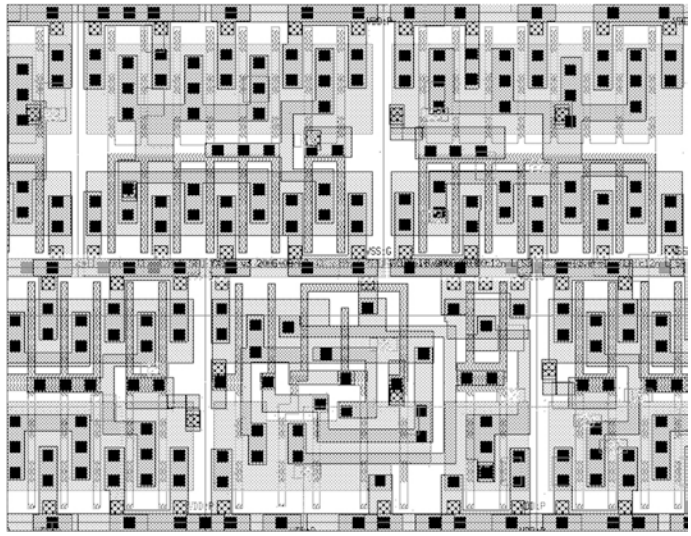


Abb. 10.15 Teil eines Chip-Layouts. (Quelle: Infineon)

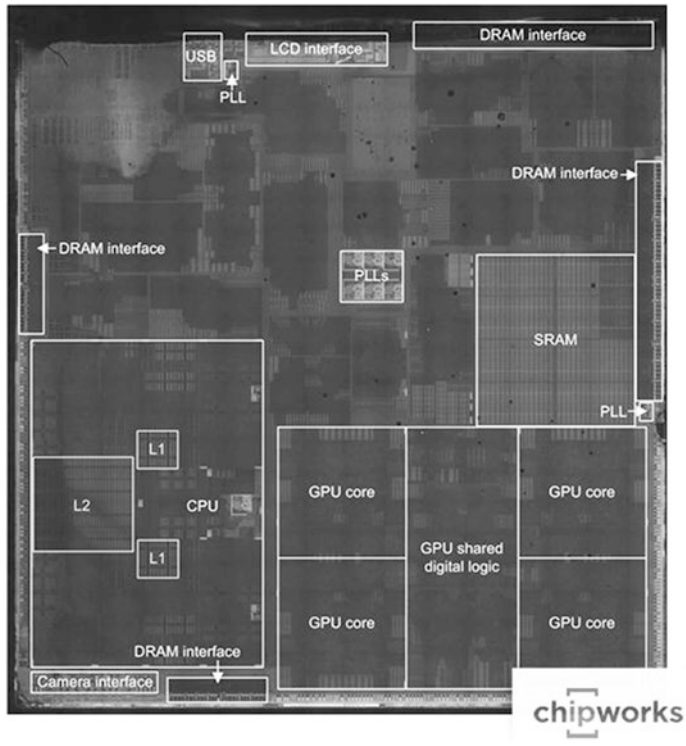


Abb. 10.16 Chip-Foto eines System-on-Chip für ein Smartphone. (Foto: Chipworks)

10.4.2 Herstellung

Als Grundmaterial für die Herstellung von CMOS-Schaltungen wird monokristallines Silizium verwendet. Die Herstellung erfolgt auf dünnen Siliziumscheiben, genannt *Wafer*. Ein Wafer ist etwa 1 mm dick und hat einen Durchmesser zwischen 15 und 30 cm (Abb. 10.17). Auf diesem Substrat werden durch aufwendige chemische und physikalische Prozesse die Strukturen für die Schaltung aufgebracht. Aus einem kompletten Wafer können mehrere hundert einzelne Chips gefertigt werden.

Die Anzahl an Chips je Wafer ergibt sich direkt aus der Fläche. Als Zahlenbeispiel betrachten wir einen Wafer mit 30 cm Durchmesser, auf dem sich Chips mit der Fläche von 2 cm^2 befinden. Die Kreisfläche ist $\pi \cdot r^2$, also $3{,}14 \cdot (15\ cm)^2 = 707\ cm^2$. Da jeder Chip 2 cm^2 benötigt, ergibt der Wafer theoretisch 353 Chips. An den Kanten, zum Sägen der Chips und für kleine Testflächen geht jedoch Fläche verloren. Praktisch können aus dem Wafer darum etwa 250 bis 300 Chips hergestellt werden.

Auf dem Wafer werden die Strukturen der Transistoren und Metallleitungen in mehreren Arbeitsschritten nacheinander gefertigt. Abb. 10.18 zeigt den Arbeitsschritt der Erzeugung von Source und Drain eines Transistors (vergleiche Abb. 10.4). Das Substrat ist p-dotiert und für Source und Drain sollen zwei n-dotierte Bereiche entstehen. Das Bild zeigt einen kleinen Ausschnitt des Wafers in Seitenansicht.

Zunächst wird die Oberfläche mit Fotolack versehen, mit einer Belichtungsmaske abgedeckt und belichtet. Dieser Verarbeitungsschritt wird als Lithographie (auch Belichtungstechnik) bezeichnet. Die nicht belichteten Stellen können entfernt werden und lassen das darunter liegende Substrat frei (Abb. 10.18, links). Dann wird der Halbleiter in eine Atmosphäre mit dem Dotierungsgas gebracht und erhitzt. Für eine n-Dotierung kann die Dotierung zum Beispiel Arsen sein. Die Dotierungsatome dringen in das Substrat ein und bilden Source und Drain (Abb. 10.18, rechts).

Auf diese Art werden Schritt für Schritt die einzelnen Ebenen einer Schaltung erzeugt. Die komplette Bearbeitung eines Wafers benötigt mehrere hundert Verarbeitungsschritte. Dazu gehört immer wieder das Auftragen von Fotolack, Belichten mit einer Fotomaske, Freiätzen unbelichteter Regionen, Dotieren nichtabgedeckter Bereiche und Entfernen des Fotolacks. Für die einzelnen Schaltungsebenen werden rund 20 bis 30 verschiedene Belichtungsmasken benötigt.

Aufgrund der sehr feinen Strukturen würde ein Staubkorn oder ein Haar auf dem Wafer die Fertigung stören und der Chip wäre an der Stelle des Staubkorns unbrauchbar. Darum findet die Fertigung in einem Reinraum statt. Dort trägt man spezielle Schutzkleidung und einen Mundschutz und es werden möglichst Industrieroboter eingesetzt. Dennoch bleibt trotz aller Sorgfalt eine geringe Staubkonzentration, so dass sich Fertigungsfehler nicht komplett vermeiden lassen.

Darum müssen sämtliche ICs nach der Fertigung einzeln getestet werden. Üblicherweise erfolgt dieser *Fertigungstest* zweimal, einmal noch auf dem Wafer, ein anderes Mal nach dem Verpacken. Durch den ersten Test werden Kosten beim Verpacken in die Gehäuse

Abb. 10.17 Silizium-Wafer. (Foto: imec)

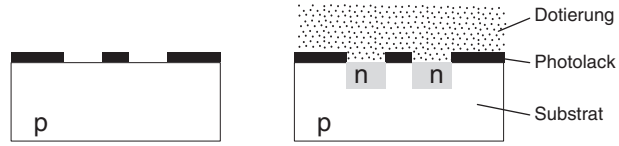


Abb. 10.18 Substrat vor und während der Dotierung von Source und Drain eines CMOS-Transistors

gespart, denn defekte Chips werden nicht weiter verarbeitet. Durch den zweiten Test wird überprüft, ob das Zersägen des Wafers und das Verpacken zu Fehlern geführt haben.

Der Anteil der korrekt gefertigten ICs wird als *Ausbeute* (engl. *Yield*) bezeichnet. Genaue Ausbeutewerte werden von den Halbleiterfirmen als Betriebsgeheimnis gehütet. Werte für eine eingefahrene Fertigung können bei 80 bis 90 % liegen. Für eine neue Halbleitertechnologie kann die Ausbeute jedoch auch bei nur 10 % oder noch darunter liegen. Dennoch kann solch eine Fertigung wirtschaftlich sein, wenn die Produkte aufgrund der Leistungsfähigkeit der neuen Technologie einen entsprechend hohen Preis erzielen.

10.4.3 Packaging

Nach Erstellen der Schaltungsstrukturen wird schließlich der Wafer in einzelne Chips zersägt und in Gehäuse verpackt. Diese unverpackten Chips werden auch als *Die* bezeichnet; der Plural ist *Dies* oder *Dice*. Mit dünnen Golddrähtchen werden *Die* und Gehäuse miteinander verbunden. Die Drähtchen werden als *Bond-Draht* bezeichnet, der Fertigungsschritt als *Bonding*. Abb. 10.19 zeigt, wie in einem geöffneten Gehäuse die Bond-Drähte eine Verbindung zum *Die* herstellen. Für die Bond-Drähte wird Gold als Material verwendet, weil es ein sehr guter elektrischer Leiter ist und sich für diese Anwendung gut verarbeiten lässt.

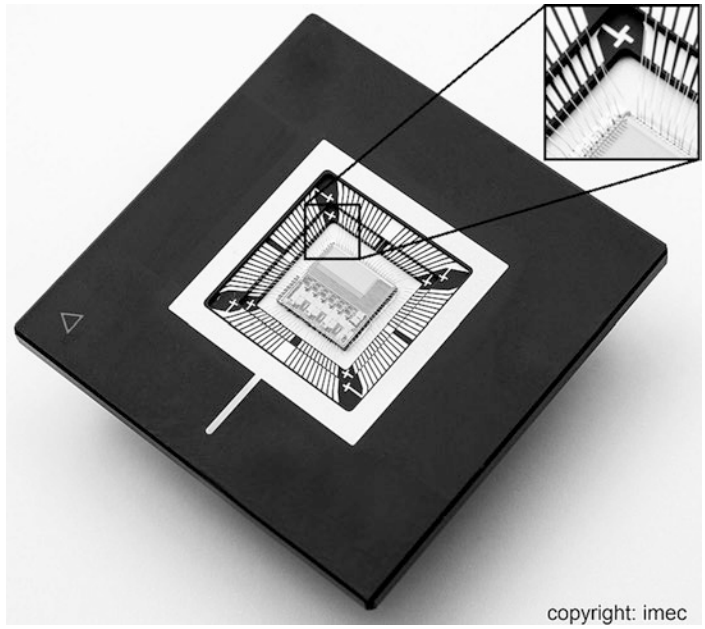


Abb. 10.19 Geöffneter Chip mit Bond-Drähten zwischen Die und Gehäuse. (Foto: imec, bearbeitet)

Die Anschlussflächen im Inneren des Gehäuses sind mit den Pins außen am Gehäuse verbunden. Mit den Pins erfolgt dann die elektrische Verbindung zur Platine.

10.4.4 Gehäuse

Es sind verschiedene Gehäuseformen gebräuchlich. Hauptkriterium für die Auswahl des Gehäuses durch den Hersteller ist die Anzahl der Anschlüsse. Weitere Kriterien sind auftretende Verlustleistung, Platzbedarf und Gehäusekosten. Um die Ausrichtung der ICs zu bestimmen, sind an den Gehäusen Orientierungsmarken angebracht, meist ein eingeprägter Punkt oder eine Kerbe im Gehäuse. Zusätzlich kann sich in einer Ecke ein fehlender oder zusätzlicher Pin befinden.

Abb. 10.20 zeigt beispielhaft einige Gehäuseformen. Von links nach rechts sind abgebildet:

- **DIL-Gehäuse** (Dual In-Line): Geeignet für kleine Anzahl an Pins. Die „Beinchen“ des Gehäuses sind für eine Durchsteckmontage gedacht, werden also durch Löcher in der Platine geführt.
- **PLCC-Gehäuse** (Plastic Leaded Chip Carrier): Für mittlere Anzahl an Pins geeignet. Die Pins erlauben die Oberflächenmontage und das Einstecken in Sockel.

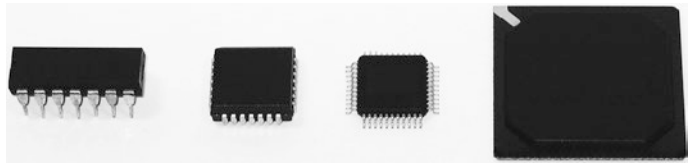


Abb. 10.20 Verschiedene Gehäuse für Integrierte Schaltungen

- **QFP-Gehäuse** (Quad Flat Pack): Ebenfalls für mittlere Anzahl an Pins und die Oberflächenmontage geeignet. Im Vergleich zu PLCC etwas kleinere Pins.
- **BGA-Gehäuse** (Ball Grid Array): Bis zu großer Anzahl an Pins verfügbar. Die Anschlüsse für die Oberflächenmontage befinden sich als Lötkugeln unterhalb des Bausteins.

10.5 Miniaturisierung der Halbleitertechnik

Die erste Integrierte Schaltung wurde 1958 von Jack Kilby entwickelt, der dafür den Nobelpreis für Physik erhielt. Seitdem hat sich die Halbleitertechnik kontinuierlich weiterentwickelt. Ein wesentlicher Fortschritt ist, dass es durch geschickte Fertigungstechnik gelungen ist, die Größe eines Transistors immer weiter zu reduzieren.

Als Angabe wie klein die Strukturen einer Halbleitertechnologie sind, wird die sogenannte *Strukturgröße* als Größenangabe verwendet. Früher entsprach die Strukturgröße der Gate-Länge L des Transistors (vergleiche Abb. 10.3). Durch verschiedene Möglichkeiten für die Gestaltung der Transistorgeometrie hat die Strukturgröße heute jedoch keinen direkten Bezug zu einer bestimmten Geometrie. Eine kleinere Strukturgröße kennzeichnet einen moderneren Prozess, der mehr Transistoren enthalten kann. Durch die kleineren Abmessungen arbeitet er schneller und mit weniger Verlustleistung. Die Strukturgröße beträgt aktuell 10 Nanometer (Stand 2016). Diese Angabe finden Sie oft in Zeitschriftenartikeln, beispielsweise als „neue CPU in 10 nm Technologie“. Ein menschliches Haar hat übrigens einen Durchmesser von rund 80 µm, ist also 8000mal so dick.

10.5.1 Moore’sches Gesetz

Durch die Miniaturisierung passen immer mehr Transistoren auf einen einzelnen Chip. Diese Entwicklung wird als *Moore’sche Gesetz* bezeichnet.

> Das Moore’sche Gesetz besagt: Die Anzahl der Transistoren pro Integrierter Schaltung verdoppelt sich alle zwei Jahre.

Abb. 10.21 zeigt den Anstieg der Integration. Die vertikale Achse hat eine logarithmische Skala, das heißt, ein Teilstrich der Skala entspricht einem Multiplikationsfaktor von 10 gegenüber dem vorherigen Teilstrich. Die Punkte stellen Einführungsjahr und

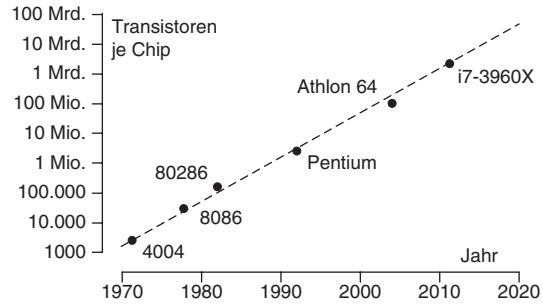


Abb. 10.21 Das Moore'sche Gesetz beschreibt die stetige Zunahme an Transistoren je integrierter Schaltung

Transistoranzahl für einige Computer-Prozessoren dar, angefangen beim Intel 4004, dem ersten in Serie produzierten Mikroprozessor.

Gordon Moore, ein Mitbegründer der Firma Intel, hat die nach ihm benannte Aussage, die natürlich kein Naturgesetz, sondern eine Prognose ist, bereits 1965, also am Anfang der „Geschichte" integrierter Schaltkreise formuliert. Ursprünglich wurde sogar eine jährliche Verdopplung prognostiziert, 1975 dann auf den Zeitraum von zwei Jahren zurückgenommen. Das „Moore's Law" ist oft zitiertes Synonym für das stürmische Wachstum der Halbleiterindustrie. Ein Ende dieser Entwicklung wurde zwar oft vorausgesagt, scheint aber für die nächsten Jahre noch nicht in Sicht.

10.5.2 FinFET-Transistoren

Bei der Miniaturisierung von Halbleitern gibt es eine natürliche Grenze: Die Größe der Atome. Der Atomdurchmesser eines Siliziumatoms beträgt etwa 0,25 nm, so dass die Gate-Länge heute bereits unter hundert Atomen liegt. Als Folge müssen für die Schalteigenschaften der Transistoren quantenphysikalische Einflüsse einzelner Atome beachtet werden. Durch die kleinen Abmessungen verschlechtern sich die elektrischen Eigenschaften der Transistoren.

Darum werden neue Transistorgeometrien entwickelt, die für sehr kleine Strukturen besser geeignet sind, als die in Abschn. 10.1.2 beschriebenen, sogenannten Planar-Transistoren. Eine erfolgreich eingesetzte Struktur sind *FinFET-Transistoren*. Dabei liegt das Gate nicht oberhalb des Kanals, sondern um einen Steg herum, der wie eine Finne oder

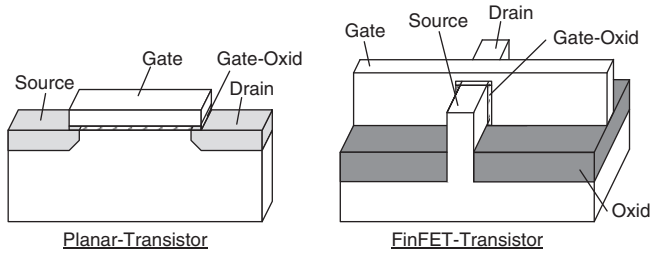


Abb. 10.22 Dreidimensionaler Aufbau eines FinFET-Transistors

Rückenflosse aussieht. Aus dieser Finne und der Abkürzung FET für Feldeffekttransistor ergibt sich der Name FinFET. Der physikalische Aufbau eines FinFET-Transistors ist in Abb. 10.22 dargestellt. Das Gate umschließt den Kanal von drei Seiten und hat daher auf kleinem Raum eine hohe Schaltwirkung. Auch Abb. 10.6 zeigt FinFET-Transistoren.

10.5.3 Weitere Technologieentwicklung

In den nächsten Jahren werden Fortschritte in der Fertigungstechnik für eine weitere Miniaturisierung sorgen. Techniken in der Erprobung sind unter anderem dreidimensionaler Aufbau von Schaltungen und Verbindungen mit *Kohlenstoffnanoröhren* (CNT, englisch *Carbon Nanotubes*). Das Grundprinzip digitaler Schaltungen, also das Schalten von Nullen und Einsen bleibt auch für die vorgeschlagenen neuen Fertigungstechniken erhalten.

Das Problem für eine neue Fertigungstechnik ist oft die Zuverlässigkeit in der industriellen Fertigung. Wenn im Labor ein Aufbau funktioniert, ist dies nur der erste Schritt. Eine neue Technik muss auch in der Massenfertigung zu vertretbaren Kosten eine hohe Fertigungsausbeute ergeben.

10.6 Übungsaufgaben

Haben Sie den Inhalt des Kapitels verstanden? Prüfen Sie sich mit den Aufgaben und Fragen am Kapitelende. Die Lösungen und Antworten finden Sie am Ende des Buches.

Bei den Auswahlfragen ist immer genau eine Antwort korrekt.

Aufgabe 10.1
Was für Schaltelemente werden für CMOS-Schaltungen benutzt?

a) Feldeffekttransistoren
b) Mechanische Schalter
c) Feldeffekt- und Bipolartransistoren
d) Bipolartransistoren

Aufgabe 10.2
Ein CMOS-Inverter besteht aus zwei Transistoren. Wie heißt der mit Versorgungsspannung (VDD) verbundene Transistor?

a) Depletion-Transistor
b) p-Kanal Transistor
c) n-Kanal Transistor
d) Verarmungstransistor

Aufgabe 10.3
Ein CMOS-Inverter besteht aus zwei Transistoren. Wie heißt der mit Masse (GND) verbundene Transistor?

a) n-Kanal Transistor
b) Depletion-Transistor
c) p-Kanal Transistor
d) Verarmungstransistor

Aufgabe 10.4
Wenn bei CMOS das Substrat p-dotiert ist, muss der p-Kanal-Transistor in einem speziellen, umdotierten Gebiet liegen. Wie wird dieses Gebiet bezeichnet?

a) Sillicon Region
b) Silicon Valley
c) n-Wanne
d) Raumladungszone
e) Verarmungszone

Aufgabe 10.5
Was bedeutet der Begriff Complementary (komplementär) bei CMOS-Gattern?

a) Es ist stets entweder n-Kanal- oder p-Kanal-Netzwerk leitend
b) p-Kanal-Transistoren haben eine größere Kanalweite
c) CMOS-Gatter beinhalten normalerweise eine Invertierung
d) p-Kanal- und n-Kanal-Transistoren haben entgegengesetztes Verhalten

Aufgabe 10.6
Warum hat im CMOS-Inverter der p-Kanal-Transistor eine 2–3fache Kanalweite?

a) Löcher haben eine höhere Beweglichkeit als Elektronen
b) Löcher haben eine geringere Beweglichkeit als Elektronen
c) Die Schaltzeiten 0 nach 1 sowie 1 nach 0 sollen unterschiedlich sein
d) Die Reihenschaltung mehrerer Transistoren wird ausgeglichen
e) Die Parallelschaltung mehrerer Transistoren wird ausgeglichen

Aufgabe 10.7
Welchen Aufbau hat ein Transmission-Gate?

a) Zwei unterschiedliche Inverter sind parallel geschaltet
b) Es werden nur p-Kanal-Transistoren verwendet

c) Zwei Inverter sind in Reihe geschaltet
d) n-Kanal und p-Kanal-Transistor sind parallel geschaltet
e) Es werden nur n-Kanal-Transistoren verwendet

Aufgabe 10.8
Was besagt das Moore'sche Gesetz?

a) Die Fläche von Integrierten Schaltungen verdoppelt sich alle zwei Jahre
b) Der Stromverbrauch Integrierter Schaltungen ist proportional zur Anzahl an Transistoren
c) Die Fläche von Integrierten Schaltungen halbiert sich alle zwei Jahre
d) Die Anzahl der Transistoren pro Integrierter Schaltung verdoppelt sich alle zwei Jahre
e) Der Stromverbrauch Integrierter Schaltungen ist proportional zur Fläche

Aufgabe 10.9
Was kennzeichnet einen FinFET-Transistor?

a) Die Dotierung wird besonders schwach gewählt
b) Der Kanal ist oberhalb des Gatters
c) Die Dotierung wird besonders stark gewählt
d) Es handelt sich um einen Bipolartransistor
e) Das Gate liegt um den Kanal herum

Aufgabe 10.10
Welche Funktion hat die Schaltung in Abb. 10.23?

Hinweis: Bei einer 0 am Eingang leiten die p-Kanal-Transistoren (oberes Netzwerk), bei einer 1 am Eingang leiten die n-Kanal-Transistoren (unteres Netzwerk). Stellen Sie eine Funktionstabelle für die vier möglichen Eingangskombinationen auf und ermitteln Sie, welcher Spannungswert am Ausgang anliegt. Aus der Funktionstabelle können Sie die Logikfunktion erkennen.

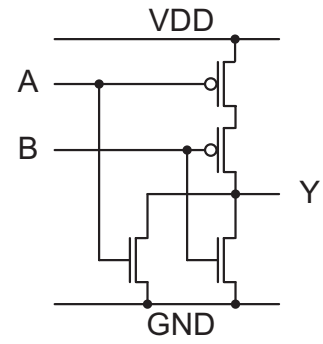


Abb. 10.23 Schaltung für Aufgabe 10.10

Speicher

11

Die Speicherung von Informationen ist eine wichtige Funktion innerhalb von Digitalschaltungen. Für kleine Speichergrößen werden Flip-Flops eingesetzt, die bereits aus vorherigen Kapiteln bekannt sind. Für mittlere und größere Datenmengen sind spezielle Speicherstrukturen effizienter, die in diesem Kapitel vorgestellt werden. Für mittlere Datengrößen werden die Speicher auf einem Chip integriert. Für sehr große Datenmengen sind spezielle Speicherbausteine verfügbar.

Es gibt verschiedene Technologien für den Aufbau von Speichern, die sich in ihren Eigenschaften deutlich unterscheiden und daher jeweils eigene Anwendungsbereiche haben. Die wichtigste Unterscheidung bei den Speichertechnologien ist die Speicherfähigkeit ohne Betriebsspannung.

- **Flüchtige Speicher** benötigen eine Versorgungsspannung zum Erhalt der Informationen. Zu diesen Speichern gehören SRAM und DRAM. Auch Flip-Flops benötigen die Versorgungsspannung zur Informationsspeicherung.
- **Nichtflüchtige Speicher** behalten ihren Inhalt auch ohne Versorgungsspannung. Zu diesen Speichern gehören EEPROM, FRAM, MRAM, PCRAM und RRAM.

Die englischen Begriffe sind *Volatile Memory* und *Non-Volatile Memory*.

Im Folgenden werden zunächst die verschiedenen Technologien zur Speicherung erläutert und danach aktuelle Speicherbausteine betrachtet.

W. Gehrke et al., *Digitaltechnik,* Springer-Lehrbuch,
DOI 10.1007/978-3-662-49731-9_11

11.1 Übersicht

11.1.1 Begriffe und Abkürzungen

Für die verschiedenen Speichertypen und Speicherorganisationen werden eine Reihe von Begriffen und Abkürzungen verwendet. Für Ihren Überblick klären wir für zunächst die wichtigsten Bezeichnungen.

- **SRAM** steht für *Static Random Access Memory,* also ein statischer Speicher mit wahlfreiem Zugriff.
- **DRAM** steht für *Dynamic Random Access Memory,* also ein dynamischer Speicher mit wahlfreiem Zugriff.

Der Unterschied zwischen statisch und dynamisch bedeutet, dass ein SRAM seine Daten unbegrenzt hält, solange die Versorgungsspannung anliegt. Das DRAM hingegen würde Daten nach einiger Zeit verlieren und darum muss die gespeicherte Information in regelmäßigen Abständen aufgefrischt werden. Der Fachbegriff für diesen Vorgang ist *Refresh.*

- **ROM** ist ein *Read-Only-Memory,* also ein Speicher, der nur gelesen werden kann. Er enthält feste Werte, die nicht verändert werden können.
- **EEPROM** ist ein nicht-flüchtiger Speicher, der mehrfach neu beschrieben werden kann. Die Abkürzung steht für *Electrically Erasable Programmable Read-Only Memory.*
- **FRAM, MRAM, PCRAM** und **RRAM** sind innovative nichtflüchtige Speicher. Die Abkürzungen stehen für *Ferroelectric RAM, Magnetoresistive RAM, Phase-Change RAM* und *Resistive RAM.*
- **NVRAM** steht für *Non-Volatile RAM* und ist der Oberbegriff für nichtflüchtige Speicher.

In dem Begriff EEPROM ist eine längere Geschichte der Speichertechniken verborgen.

- **ROM** ist der Ausgangspunkt. Sie werden mit festem Speicherinhalt hergestellt, der vor der Fertigung festgelegt wurde.
- **PROM** steht für *Programmable ROM,* also programmierbares ROM. Damit werden Speicherbausteine bezeichnet, bei denen der Speicherinhalt programmiert werden kann. Zunächst war aber nur ein einziger Programmiervorgang möglich.
- **EPROM** steht für *Erasable PROM,* also löschbares PROM. Der Löschvorgang erfolgte durch Belichtung mit UV-Licht. Das EPROM wurde aus der Platine entnommen und für circa 15 min in ein spezielles Belichtungsgerät gelegt. Danach konnte es neu programmiert werden.
- **EEPROM** steht für *Electrically Erasable PROM,* also ein PROM, welches elektrisch löschbar ist und nicht mehr belichtet werden muss.

- **Flash-EEPROM** bezeichnet eine häufig genutzte Variante des EEPROMs. Dabei können Speicherzellen nicht einzeln geändert werden, sondern beim Ändern des Speicherinhalts werden ganze Speicherblöcke zurückgesetzt („geflasht“).

Auch der Begriff RAM, also Random Access Memory, hat historischen Hintergrund. Heutige Speicher haben fast immer einen wahlfreien Zugriff auf die gespeicherten Informationen. Früher wurden auch *FIFO-Speicher* verwendet, die Daten in der gleichen Reihenfolge ausgeben, mit der sie geschrieben werden. Der Begriff FIFO steht für *First-In-First-Out* und diese Speicher schieben intern die Daten wie in einem Fließband schrittweise weiter.

Auch heute werden noch FIFOs verwendet, beispielsweise in Computer-Netzwerken, wenn Datenpakete empfangen und in der gleichen Reihenfolge weitergegeben werden. In diesen FIFOs ist jedoch mittlerweile ein SRAM-Speicher enthalten, welcher in fester Reihenfolge angesteuert wird.

11.1.2 Grundstruktur

Die prinzipielle Grundstruktur ist für alle Speichertechnologien ähnlich und in Abb. 11.1 dargestellt. Die Speicherzellen sind in einer Matrixform in Zeilen und Spalten angeordnet. Auf die einzelnen Speicherzellen wird über eine *Adresse* zugegriffen. Anhand eines Teils der Speicheradresse wird eine Zeile ausgewählt. Der Rest der Speicheradresse wählt eine Spalte aus. Steuerleitungen geben an, ob Daten gelesen oder geschrieben werden sollen.

Die Daten werden über Lese- und Schreibverstärker aus der Speicherzelle gelesen beziehungsweise in die Zelle geschrieben. Über den Lese-/Schreibverstärker erfolgt der Datenaustausch mit der weiteren Schaltung. Normalerweise enthält ein Speicher

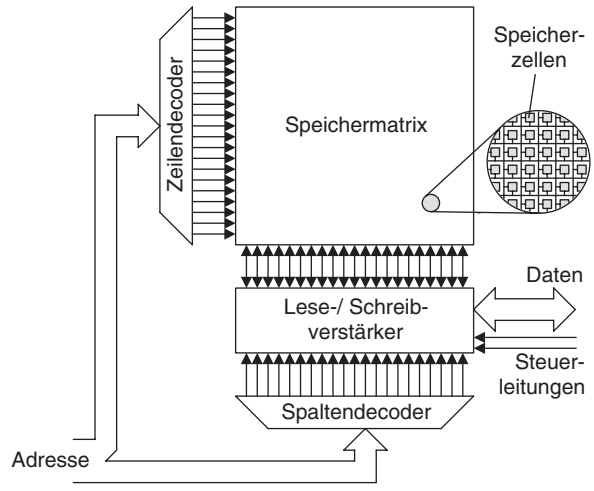


Abb. 11.1 Grundstruktur eines Halbleiterspeichers

Datenworte mit mehreren Bits, das heißt unter einer Adresse sind 8 Bit, 16 Bit oder 32 Bit gespeichert. Die einzelnen Speichertechnologien unterscheiden sich durch die Art der verwendeten Speicherzellen in der Matrix.

Durch die Matrixanordnung ergibt sich eine Zweiteilung der Adresse, welche die interne Ansteuerung des Speichers erleichtert. Anstelle eines großen Adressdecoders sind zwei kleine Decoder nötig. Die Aufteilung wird meist so gewählt, dass die Speichermatrix quadratisch ist oder ein Verhältnis von 2-zu-1 oder 4-zu-1 hat.

Als Beispiel wird ein Speicher für 2^{20} Datenworte zu 16 Bit betrachtet. Dies sind exakt 1.048.576 Datenworte, also rund eine Million. Dafür sind etwa 16 Mio. Speicherzellen erforderlich, die bei einer quadratischen Aufteilung eine Speichermatrix aus 4096 Zeilen und 4096 Spalten bilden. Jeweils 16 Zellen einer Zeile bilden ein Datenwort und haben die gleiche Adresse. Es müssen also 4096 Zeilen und $4096/16 = 256$ Spalten angesteuert werden.

Aus der Speichergröße ergibt sich die benötigte Wortbreite für die Adresse. Mit n Adressleitungen können 2^n Adressen angesteuert werden.

Der Speicher mit 2^{20} Datenworten benötigt somit 20 Adressleitungen. In der internen Struktur werden 12 Adressleitungen verwendet, um die Zeilenadresse zu bestimmen. Dies berechnet sich aus den 4096 Adressen, die dem Wert 2^{12} entsprechen. Die restlichen 8 Adressleitungen bestimmen die Spaltenadresse, denn 256 ist 2^8.

11.1.3 Physikalisches Interface

Die Geschwindigkeit eines Datenzugriffs ist natürlich wichtig für die Leistungsfähigkeit eines Speichers. Dabei unterscheidet man zwischen *Latenzzeit* und *Datentransferrate*. Die Latenzzeit ist die Reaktionszeit auf einen Datenzugriff und hängt von der Organisation des Speichers ab. Die Datentransferrate ist die Geschwindigkeit mit der Daten zwischen Speicher und System übertragen werden.

Die höchste Datentransferrate ist möglich, wenn der Speicher sich auf demselben Chip wie das restliche System befindet. Dies wird als interner Speicher oder *Embedded Memory* bezeichnet. Für separate Speicherbausteine, also externen Speicher, ist die Verbindung, das physikalische Interface zwischen Speicher und System, entscheidend für die Datentransferrate.

Zur Beschleunigung des Datentransfers werden verschiedene Schaltungstechniken eingesetzt.

Reduzierter Spannungshub mit Referenzspannung

Die Leitungen zwischen System und Speicher haben Kapazitäten, die bei Signalwechseln umgeladen werden müssen. Um dies zu beschleunigen, wird der Spannungshub auf den Leitungen reduziert. Allerdings sinkt dadurch auch der Störabstand, denn der Übergangsbereich zwischen Low- und High-Pegel wird sehr klein. Als Ausgleich wird eine Referenzspannung eingeführt. Wenn der Signalpegel höher als die Referenzspannung ist,

wird bei positiver Logik eine 1 erkannt. Spannungen unterhalb des Referenzpegels werden als eine logische 0 interpretiert. Störungen wirken sich auf Signale und Referenzspannung gleichermaßen aus, so dass die Information nicht verfälscht wird.

Terminierung von Leitungen
Auf elektrischen Leitungen können Reflektionen von Signalwechseln auftreten. Wenn diese die eigentlichen Signale überlagern, sind Fehler in der Datenübertragung möglich. Für die Signalleitungen zu externen Speichern gibt es daher Layout-Regeln, damit die Leitungen einen passenden Wellenwiderstand haben. Außerdem können auf der Platine oder direkt auf den Chips Abschlusswiderstände für eine Terminierung der Leitungen sorgen.

Double-Data-Rate
Schnelle Speicher verwenden ein synchrones Interface, bei denen die Abfolge der Daten durch einen Takt angezeigt wird. Allerdings kann die hohe Frequenz des Taktsignals problematisch sein. Grund ist, dass der Takt schnellere Signalwechsel als die Datenleitungen hat. Der Takt wechselt in jedem Zyklus von 0 nach 1 und wieder von 1 nach 0. Ein Datensignal hat jedoch pro Taktzyklus maximal einen Signalwechsel und damit die halbe Frequenz.

Zur Verringerung der Frequenz für das Taktsignal wird eine Datenübertragung mit *Double-Data-Rate,* abgekürzt *DDR,* verwendet. Dabei signalisieren steigende *und* fallende Taktflanken die übertragenen Daten. Pro Taktzyklus werden also zwei Datenworte übertragen, was zu der Bezeichnung „doppelte Datenrate“ führt.

11.2 Speichertechnologien

11.2.1 SRAM

Im SRAM erfolgt die Datenspeicherung durch Rückkopplung zweier Inverter. Abb. 11.2 zeigt einen Ausschnitt aus der Speichermatrix. Die Inverter sind wechselseitig mit ihren Ein- und Ausgängen verbunden, so dass eine gespeicherte 0 oder 1 doppelt invertiert und verstärkt wird. Damit bleibt die Information erhalten. Beim Abschalten der Versorgungsspannung entfällt die Rückkopplung, die Daten gehen verloren, der Speicher ist flüchtig.

Angesteuert werden die SRAM-Zellen über eine Zeilenadresse sowie Datenleitungen. Für jede Spalte sind zwei Datenleitungen vorhanden, die Daten und invertierte Daten verbinden.

- Zum Lesen von Daten wird eine Zeile ausgewählt und die Zeilenadresse auf 1 gesetzt. Dadurch werden alle Speicherzellen einer Zeile mit den Datenleitungen verbunden. Der Leseverstärker wählt dann die richtigen Spalten aus und gibt die Daten an den Ausgang.

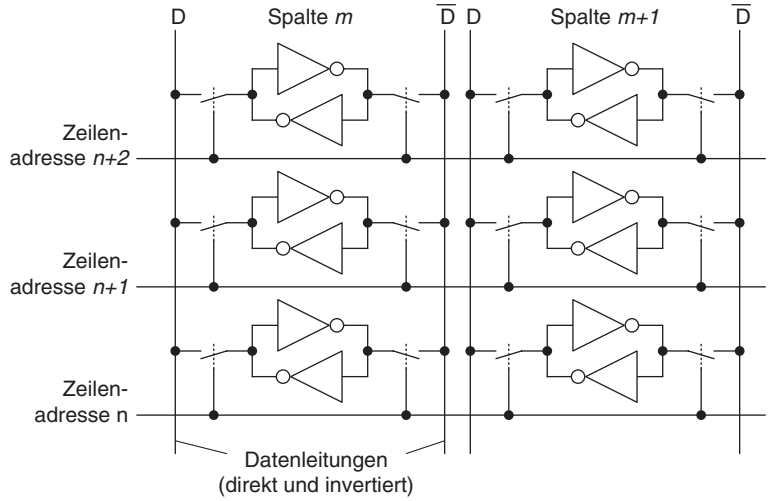


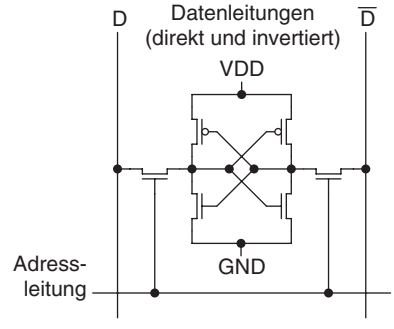
Abb. 11.2 Speicherzellen eines SRAMs

- Zum Schreiben von Daten wird ebenfalls eine Zeile durch Zeilenadresse auf 1 ausgewählt. Wiederum werden die Speicherzellen mit den Datenleitungen verbunden. Dort wo Daten geschrieben werden, müssen die Datenleitungen die neuen Werte enthalten. Außerdem muss der Schreibverstärker so stark sein, dass er die Rückkopplung der Speicherzelle überschreibt.

Die Speicherzelle selbst ist in Abb. 11.3 dargestellt. Die Inverter haben jeweils zwei Transistoren, die Schalter sind durch jeweils einen einzelnen Transistor aufgebaut. Anders als beim Transmission-Gate (vgl. Kapitel 10) wird nur ein n-Kanal-Transistor verwendet, um Transistoren zu sparen. Insgesamt benötigt die SRAM-Zelle 6 Transistoren. Sie wird daher auch als *6T-Zelle* bezeichnet.

Wir betrachten wieder den Speicher mit 2^{20} Datenworten zu 16 Bit. Horizontal verlaufen 4096 Zeilenadressen und vertikal für jede Zelle zwei Datenleitungen also

Abb. 11.3 Transistoraufbau einer SRAM-Speicherzelle



insgesamt 8192. Für die rund 16 Mio. Speicherzellen werden 6 · 16 Mio., also 96 Mio. Transistoren benötigt. Bei der Adressierung eines 16-Bit-Wortes werden 32 nebeneinanderliegende Datenleitungen angesprochen, je Bit zwei Leitungen.

11.2.2 DRAM

Ein DRAM verwendet eine andere Art der Speicherung. Eine Information wird als Ladung auf einem kleinen Kondensator gespeichert. Ein Transistor dient als Schalter zur Datenleitung. Die Adressleitung öffnet den Transistor, so dass die Ladung gespeichert oder abgefragt werden kann (Abb. 11.4).

Der wesentliche Vorteil der DRAM-Speicherung ist der geringere Platzbedarf gegenüber einem SRAM. Zunächst werden weniger Komponenten benötigt, und zwar nur ein Transistor und ein Kondensator, verglichen mit den sechs Transistoren des SRAMs. Ein weiterer Platzvorteil entsteht dadurch, dass keine p-Kanal-Transistoren verwendet werden und darum keine n-Wanne mit einem Mindestabstand zu den n-Kanal-Transistoren erforderlich ist. Der Masseanschluss des Kondensators verbindet zum Substrat. Darum wird keine Masseleitung benötigt und auch Versorgungsspannung sowie eine zweite Datenleitung sind nicht erforderlich, was weiterhin Platz einspart. Die Speicherkapazität eines DRAMS ist dadurch wesentlich höher als bei einem SRAM.

Das Speicherprinzip des DRAMs hat jedoch auch Nachteile, insbesondere die Notwendigkeit einer speziellen Halbleitertechnologie sowie die begrenzte Datenerhaltung.

Spezielle Halbleitertechnologie

Wichtig für die Informationsspeicherung ist ein Kondensator mit ausreichender Kapazität. Dieser ist in einem Standard-CMOS-Prozess nicht vorhanden, so dass eine spezielle Halbleitertechnologie erforderlich ist. Ein SRAM-Speicher hingegen lässt sich auf einem Standard-CMOS-Prozess fertigen.

Es gibt verschiedene Möglichkeiten, einen Kondensator aufzubauen. Zwei Grundprinzipien sind Capacitor over Bitline (COB) und Trench-Transistoren.

- Bei Capacitor over Bitline befindet sich der Kondensator oberhalb der Datenleitung (Bitline) und wird beim Aufbau der verschiedenen Schichten eines Chips erzeugt.
- Als Trench-Kondensator wird in das Substrat ein Graben (engl. *Trench*) oder Loch geätzt und mit leitfähigem Material aufgefüllt. Grundprinzip und Chipfoto einer

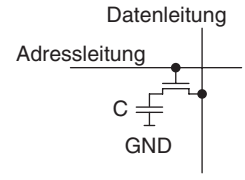


Abb. 11.4 Speicherzelle eines DRAMs

DRAM-Zelle mit Trench-Kondensator sind in Abb. 11.5 dargestellt. WL (Write Line) bezeichnet die Adressleitung.

Begrenzte Datenerhaltung

Die Ladung des Kondensators wird nicht, wie beim SRAM, durch eine Rückkopplung automatisch erhalten. Dies muss für die Speicherung und für den Lesevorgang berücksichtigt werden.

Bei der Speicherung wird der Kondensator durch Leckströme langsam entladen. Die Daten werden also nur für einen kurzen Zeitraum gespeichert und müssen durch einen *Refresh* periodisch erneuert werden. Die garantierte Speicherzeit zwischen zwei Refreshvorgängen ist abhängig von der Halbleitertechnologie und liegt in der Größenordnung von 100 ms.

Beim Lesevorgang wird der Transistor am Kondensator geöffnet und die Ladung über die Datenleitung gelesen. Dies erfordert einen sehr empfindlichen Leseverstärker, der erkennen muss, ob ein kleiner Kondensator am Ende einer langen Datenleitung geladen oder nicht geladen war. Außerdem wird durch das Lesen des Kondensators die Information gelöscht. Nach dem Lesen einer Zelle muss also immer die Information wieder in die Kondensatoren zurückgeschrieben werden.

Dies hört sich zunächst nach einem sehr hohen Aufwand an. Gemildert wird der Aufwand dadurch, dass beim Lesen eine ganze Zeile in den Leseverstärker geladen wird. Weitere Datenzugriffe in die gleiche DRAM-Zeile können darum sehr schnell erfolgen, da die Daten bereits im Leseverstärker vorhanden sind.

Als Zahlenbeispiel nehmen wir wieder den oben betrachteten Speicher mit 2^{20} Datenworten zu 16 Bit. Wenn er als DRAM implementiert ist, wird zunächst eine der 4096 Zeilenadressen angesprochen und in den Leseverstärker geladen. Dort stehen dann 256 Worte zu 16 Bit für den schnellen Datenzugriff bereit.

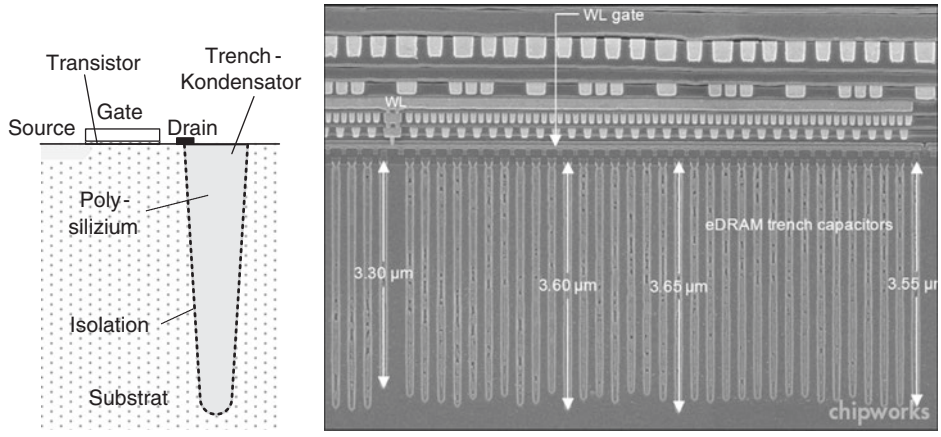


Abb. 11.5 DRAM-Speicherzelle mit Trench-Kondensator als physikalischer Aufbau und im Elektronenmikroskop. (Foto: Chipworks)

Aufgrund des geringeren Platzbedarfs für die Speicherzellen wird für die Speicherung großer Datenmengen oft ein DRAM eingesetzt. Beispielsweise wird der Hauptspeicher eines PCs durch DRAM-Speicher implementiert.

11.2.3 ROM

Wenn in einem System unveränderliche Werte gespeichert werden sollen, wird ein Read-Only-Memory (ROM) eingesetzt. An den Kreuzungspunkten von Adress- und Datenleitungen befinden sich Kontaktmöglichkeiten, die verbunden oder nicht verbunden sind und damit eine 0 oder 1 darstellen. Um einen Kurzschluss über andere Speicherstellen zu vermeiden, befindet sich an der Kontaktstelle eine Diode.

Abb. 11.6 zeigt den Aufbau eines ROMs mit verbundenen und unverbundenen Kontaktstellen. Zum Lesen einer Information wird eine Adressleitung auf High-Pegel gelegt und vom Leseverstärker überprüft, ob auf der Datenleitung ein Strom fließt. Die unbenutzten Adressleitungen liegen auf Low-Pegel und sind durch die Dioden abgetrennt.

11.2.4 OTP-Speicher

Eine besondere Art eines nichtflüchtigen Speichers stellt der Einmalprogrammierbare Speicher dar. Er wird als *OTP,* also *One-Time-Programmable* bezeichnet. Ein OTP-Speicher kann nach der Programmierung nicht mehr verändert werden und gegebenenfalls muss ein kompletter Baustein ausgetauscht und weggeworfen werden. In der Anfangszeit der Mikroelektronik war eine Programmierung nicht anders möglich. Heute ist diese Einschränkung für viele Anwendungen nicht mehr akzeptabel.

Für programmierbare Schaltungen (FPGAs) wird eine Einmalprogrammierung jedoch weiterhin eingesetzt. Sie hat den Vorteil, dass sie Sicherheit gegen unbeabsichtigte Änderung oder Manipulation einer Schaltung bietet. Ein Anwendungsbeispiel sind FPGAs für Satelliten und Raumfahrt, bei denen die Programmierung durch kosmische Strahlung nicht gestört werden darf. Bei der Entwicklung werden eventuell einige wenige Bausteine mit Testversionen programmiert und ausgetauscht. Danach kann eine Kleinserie mit dem gewünschten Speicherinhalt programmiert und in Geräte eingebaut werden.

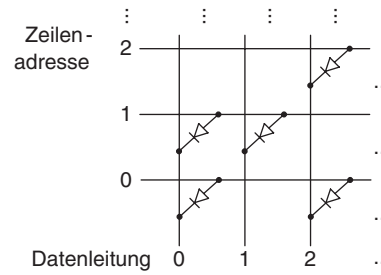


Abb. 11.6 Struktur eines ROMs

Implementiert werden Einmalprogrammierbare Speicher durch Sicherungen und Antisicherungen. Eine Sicherung brennt bei zu hohem Strom durch, während eine Anti-Sicherung *(Anti-Fuse)* bei Anlegen einer Programmierspannung eine elektrische Verbindung herstellt. In der Praxis sind heutzutage Anti-Fuses gebräuchlich, da diese zuverlässiger programmiert werden können.

Das Grundprinzip eines PROM zeigt Abb. 11.7. An jeder Verbindung von Adressleitung und Datenleitung ist eine Sicherung oder Anti-Fuse in Reihe zu einer Diode geschaltet. Bei der Programmierung wird festgelegt, welche Verbindungen benötigt werden.

11.2.5 EEPROM

Für viele Anwendungen sollen Daten nichtflüchtig gespeichert, aber auch leicht veränderbar sein. Das hierfür am weitesten verbreitete Halbleiterelement ist das EEPROM. Hierbei erfolgt die Datenspeicherung durch spezielle Transistoren mit einem zusätzlichen isolierten Gate (engl. *Floating-Gate*). Wie Abb. 11.8 zeigt, liegt das Floating-Gate zwischen dem regulären Steuer-Gate und dem Kanal. Auf dem Floating-Gate kann durch Tunneleffekte und sogenanntes Hot-Electron-Injection eine Ladung gespeichert und wieder gelöscht werden. Das Floating-Gate ist jedoch elektrisch isoliert und speichert die Ladung daher sehr lange. Die garantierte Speicherzeit beträgt je nach Baustein bis zu 20 Jahre.

Zum Lesen der Daten muss die Ladung nicht abgerufen werden. Der Transistor wird über das Steuer-Gate angesprochen. Falls keine Ladung auf dem Floating-Gate vorhanden ist, leitet der Transistor wie in der normalen CMOS-Technik. Falls eine Ladung

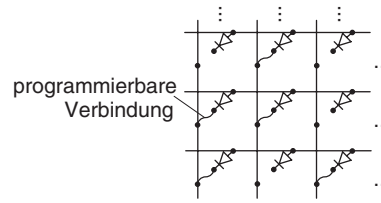


Abb. 11.7 Struktur eines OTP-Speichers

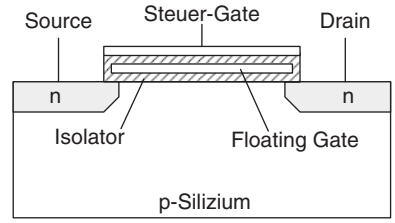


Abb. 11.8 Floating-Gate Transistor für EEPROMs

gespeichert ist, verschiebt sich die Schwellenspannung und der Transistor bleibt auch bei Ansteuerung durch das Steuer-Gate nichtleitend. So ist eine Unterscheidung des Speicherinhalts möglich.

Häufig wird die als Flash-EEPROM bezeichnete Schaltungsform eingesetzt. Hierbei hat der Schreibvorgang die Besonderheit, dass für eine einzelne Zelle nur die Änderung von einer 1 in eine 0 möglich ist. Falls eine 0 in eine 1 geändert werden soll, muss ein ganzer Block komplett auf 1 gesetzt werden und erneut die benötigten 0-Werte geschrieben werden. Typische Blockgrößen sind zwischen 8 kByte und 256 kByte. Dieses Löschen ganzer Speicherblöcke hat zu dem Namen *Flash* geführt. Ein Vorteil der Flash-Technik ist der geringere Schaltungsaufwand, u.a. weil beim Löschen nicht jede Zelle einzeln angesprochen werden muss.

Die Anzahl der möglichen Löschzyklen ist begrenzt und beträgt beispielsweise 100.000 Zyklen. Bei der Ansteuerung des Flash-EEPROMs wird meist versucht, die Blöcke möglichst gleich häufig zu benutzen, um die Lebensdauer des Bausteins zu verlängern. Diese Strategie bezeichnet man als *Wear Leveling,* also frei übersetzt „Ausgleichen der Abnutzung".

Es gibt zwei Strukturen für die Anordnung von Floating-Gate Transistoren zu einem Speicher, und zwar die NOR- und die NAND-Struktur, dargestellt in Abb. 11.9. Beiden Technologien gemeinsam ist, dass wieder eine Zeile durch einen Zeilendecoder ausgewählt wird.

- In der *NOR-Struktur* schalten die Speichertransistoren die Datenleitung parallel nach Masse. Die nicht aktiven Transistoren sind nicht leitend und stellen somit keine Verbindung nach Masse dar. Zum Lesen wird ein Transistor über die Adressleitung angesprochen. Abhängig von seinem Speicherzustand kann er daraufhin leitend werden und die Datenleitung nach Masse ziehen. Dies wird vom Leseverstärker erkannt.
- In der *NAND-Struktur* sind die Speichertransistoren in der Datenleitung in Reihe angeordnet. Die nicht aktiven Transistoren sind leitend geschaltet. Der Transistor, der

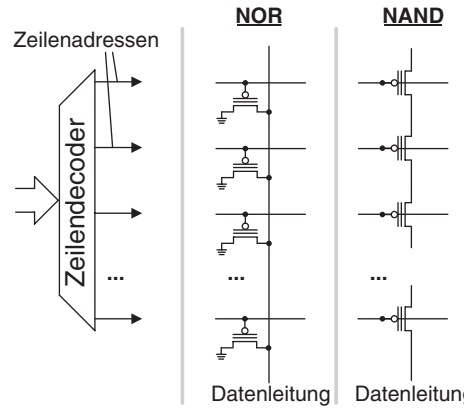


Abb. 11.9 Interne EEPROM-Speicherzellenstruktur in NOR- und NAND-Technik

gelesen werden soll, wird über die Adressleitung angesprochen und schaltet die Reihenschaltung leitend oder nicht leitend. Auch dies wird vom Leseverstärker erkannt.

Beide Strukturen werden in der Praxis eingesetzt.

- Der Vorteil der NOR-Struktur ist ein geringer Widerstand auf der Datenleitung, welcher eine gute Lesbarkeit der Daten ermöglicht. Der Nachteil ist ein höherer Flächenbedarf, da jeder Transistor einen Kontakt zu Masse benötigt.
- Der Vorteil der NAND-Struktur ist ein geringerer Flächenbedarf, da die Speichertransistoren direkt aneinander geschaltet werden. Dadurch ist die Speicherkapazität höher. Der Nachteil ist, dass die nicht aktiven Transistoren auch im leitenden Zustand noch einen gewissen Widerstand haben, der sich in der Reihenschaltung addiert. Dadurch ist das Auslesen schwieriger und es können Lesefehler auftreten.

Für die meisten Anwendungen wird heutzutage die NAND-Struktur verwendet, da die Speicherdichte deutlich höher ist. Beim Lesen können jedoch einzelne Datenworte fehlerhaft sein, sogenanntes *Bit-Flipping*. Darum wird die Information mit einem fehlerkorrigierenden Code gespeichert, englisch *Error Correcting Code (ECC)*. Durch Zusatzinformationen kann ein Controller einzelne Fehler erkennen und direkt korrigieren. Wenn zu viele Fehler in einem Speicherblock auftreten, können diese jedoch nicht mehr korrigiert werden. Ein problematischer Speicherblock muss rechtzeitig erkannt und als unbrauchbar markiert werden. Ein NAND-Speicher kann einige solcher *Bad Blocks* haben, wodurch sich seine Speicherkapazität leicht reduziert.

Eine Erhöhung der Speicherdichte ist möglich, indem verschiedene Ladungsmengen auf das Floating-Gate gespeichert werden. Je nach Ladung verschiebt sich die Schwellenspannung des Speichertransistors und kann durch den Leseverstärker unterschieden werden. Aktuell werden zwei bis vier Bit auf einem Transistor gespeichert, was die Unterscheidung von bis zu 16 verschiedenen elektrischen Ladungen erfordert. Diese Technik wird nur für NAND-Speicher eingesetzt und allgemein als *Multi-Level-Cell (MLC)* bezeichnet; bei Speicherung von 3 oder 4 Bit auch als *Triple-* oder *Quad-Level-Cell (TLC, QLC)*. Die mit diesen Techniken verbundene höhere Fehlerwahrscheinlichkeit erfordert einen Controller mit leistungsfähiger Fehlerkorrektur.

11.2.6 Innovative Speichertechniken

In den letzten Jahren ist der Markt für nicht-flüchtige Halbleiterspeicher (NVRAM) kontinuierlich gewachsen. Grund dafür ist, dass diese Speicher in immer mehr Geräten eingesetzt werden und dabei auch die Speichergrößen steigen. NVRAMs finden sich in USB-Speicher-Sticks, Digitalkameras, Mobiltelefonen, Tablets, Solid-State-Festplatten und weiteren Elektronikgeräten.

Darum werden weitere Speichertechniken entwickelt, die höhere Speicherkapazitäten, geringere Kosten oder einfachere Ansteuerung verglichen mit EEPROMs ermöglichen.

Einige dieser Techniken sind bereits im praktischen Einsatz, allerdings sind ihre Marktanteile noch recht klein. Es ist gegenwärtig nicht absehbar, welche der neuen Techniken zu einer Konkurrenz von EEPROMs werden oder diese sogar ersetzen können. Das Prinzip einiger innovativer Speichertechniken wird in diesem Unterkapitel vorgestellt.

Für die Speicherung wird ein Material gesucht, welches

- zwei verschiedene Zustände hat, die sich in ihren elektrischen Eigenschaften unterscheiden,
- einen einfachen Wechsel zwischen diesen Zuständen ermöglicht,
- beide Zustände stabil über Jahre hinweg behält,
- sehr oft zwischen diesen Zuständen wechseln kann, also mindestens hunderttausend, möglichst eine Milliarde Mal,
- platzsparend und kostengünstig zu einem CMOS-Prozess ergänzt werden kann.

Die vorgeschlagenen Speichertechniken nutzen jeweils andere Materialien zur Datenspeicherung. Die folgende Übersicht nennt aktuell verwendete Materialien für die Speichertechniken.

FRAM

FRAM, also Ferroelectric RAM, verwendet einen Kondensator mit einem ferroelektrischen Dielektrikum. Dieses Material hat eine Kristallstruktur, welche zwei stabile Zustände mit unterschiedlichem elektrischen Feld aufweist. Für das Material Blei-Zirkonat-Titanat (PZT) ist die Struktur in Abb. 11.10 dargestellt. In der Mitte der Kristallstruktur aus Blei (Pb) und Sauerstoff (O) ist ein Atom aus Zirconium oder Titan, welches sich in der unteren oder oberen Position der kubischen Struktur befinden kann. Durch ein elektrisches Feld lässt sich dieses zentrale Atom verschieben und so eine Information speichern.

MRAM

MRAM, also Magnetoresistive RAM, speichert Informationen in einer ferromagnetischen Schicht. Diese befindet sich getrennt durch ein dünnes Dielektrikum aus

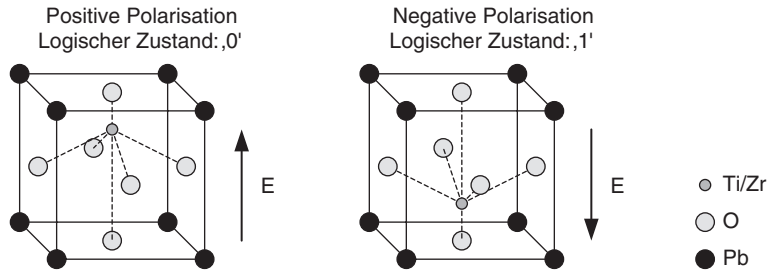


Abb. 11.10 Kristallstruktur eines FRAM-Speichermaterials

Aluminiumdioxid (Al_2O_3) gegenüber einer weiteren magnetischen Schicht (siehe Abb. 11.11). Die obere Magnetschicht ist magnetisch weich und kann in ihrer magnetischen Orientierung gedreht werden. Die untere Magnetschicht ist magnetisch hart und hat eine feste Orientierung. Der Strom durch das Dielektrikum ist durch einen Tunneleffekt abhängig davon, ob die magnetische Orientierung parallel oder antiparallel ist.

PCRAM

PCRAM, also Phase-Change-RAM, Phasenwechselspeicher, nutzt ein Material, welches eine kristalline oder amorphe Struktur einnehmen kann. Je nach Struktur ist der elektrische Widerstand unterschiedlich und zeigt so eine 0 oder 1 an. Der Wechsel zwischen den Strukturen erfolgt über Aufheizen durch elektrischen Strom. Je nach Geschwindigkeit der Abkühlung wird das Material kristallin oder amorph (Abb. 11.12).

RRAM

RRAM, auch ReRAM, für Resitive RAM, verändert ähnlich wie PCRAM den Widerstand eines Speichermaterials. Dabei befindet sich ein Metalloxid zwischen zwei Elektroden. Durch einen Strom kann der Widerstand des Metalloxids zwischen hohem und niedrigem Widerstand wechseln. Dafür ist allerdings keine Erwärmung und Abkühlung des Materials nötig, so dass ein Speichervorgang prinzipiell einfacher erfolgen kann. Ein Ausschnitt aus der Speichermatrix ist in Abb. 11.13 dargestellt.

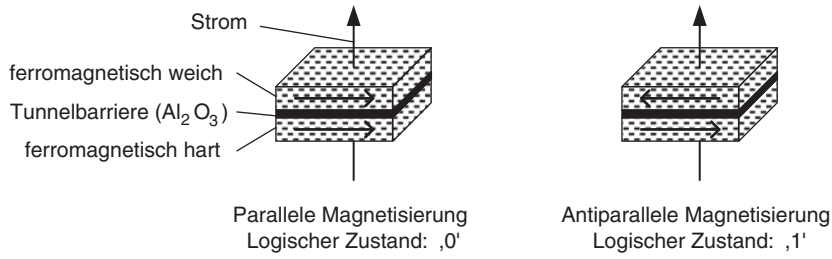
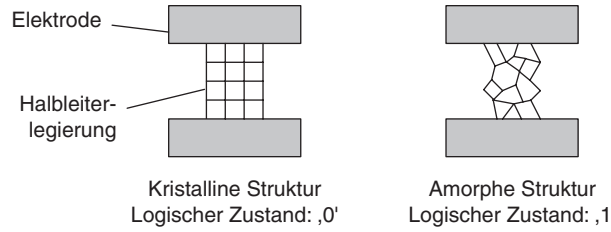


Abb. 11.11 Aufbau eines MRAM-Speicherelements

Abb. 11.12 Speicherprinzip eines Phase-Change-RAM



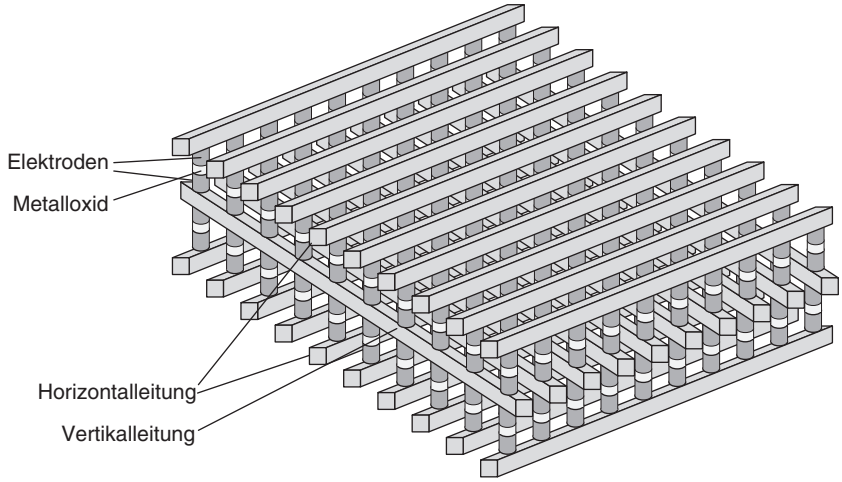


Abb. 11.13 Dreidimensionale Struktur eines RRAMs

Ansteuerung innovativer NVRAMs
Die Ansteuerung erfolgt für alle Speichertechnologien wieder in Matrixstruktur mit Adress- und Datenleitungen. Die Einbindung des Speichermaterials ist abhängig davon, welche elektrische Eigenschaft sich für die Datenspeicherung ändert. Teilweise wird ein Transistor benötigt, der die Speicherzelle freischaltet.

Eine besonders kompakte Anordnung ist für bestimmte RRAMs möglich. Durch horizontale und vertikale Leitungen kann eine einzelne Speicherzelle direkt angesprochen werden (Abb. 11.13). Durch eine Diode in der Speicherzelle, wie beim ROM, haben andere Zellen keinen Einfluss auf die Leseelektronik. Mehrere Lagen an Zellen sollen gestapelt werden, um die Speicherkapazität zu erhöhen. Dabei kann eine Leitung gemeinsam für zwei Ebenen an Speicherzellen genutzt werden (Vertikalleitung in Abb. 11.13).

11.3 Eingebetteter Speicher

Als *eingebetteter Speicher,* engl. *Embedded Memory,* werden Speicherblöcke bezeichnet, die sich gemeinsam mit einer größeren Schaltung auf einem Chip befinden.

11.3.1 SRAM

In fast jedem größeren digitalen Chip befinden sich SRAM-Speicherblöcke. Ein SRAM ist mit der normalen CMOS-Fertigungstechnik herzustellen und erfordert daher keinen zusätzlichen Fertigungsaufwand. Eingesetzt werden SRAM-Speicherblöcke beispielsweise als interner Speicher einer CPU, für die Zwischenspeicherung von Audio- und Videodaten oder bei der Zwischenspeicherung von Netzwerkdaten.

Die Ansteuerung eines SRAMs erfolgt durch Adresse, Datenleitungen und Steuerleitungen. Oft sind Flip-Flops an Eingängen und Ausgängen integriert, so dass auch ein Takteingang vorhanden ist.

- Die Adressleitungen entsprechen der Anzahl an Speicherworten. Ein Speicher mit 2^n Adressen benötigt n Adressleitungen, die parallel anliegen. So hat ein Speicher mit 1024 Speicherworten einen Adressbus mit 10 Leitungen, denn $2^{10}=1024$.
- Die Datenleitungen entsprechen der Wortbreite der Speicherworte. Ein Speicher für 16-Bit-Worte hat Datenleitungen mit 16 Stellen. Dateneingang und Datenausgang sind getrennte Leitungen. Bidirektionale Leitungen sind bei Embedded Memory nicht nötig, da die Anzahl der Verbindungsleitungen innerhalb eines Chips kaum begrenzt ist.
- Als Steuerleitung ist eine Schreibsteuerung erforderlich, die angibt, ob die Daten am Eingang in den Speicher geschrieben werden sollen. Optional ist ein Enable-Signal möglich, mit dem das SRAM zur Verringerung der Verlustleistung inaktiv geschaltet werden kann.

Ein Speicher für 1024 Worte der Wortbreite 16 bit hat damit die in Abb. 11.14 dargestellten Eingangs- und Ausgangssignale. Anstelle eines besonderen Symbols wird ein Block mit der Angabe der Speichergröße verwendet.

Embedded-SRAM werden in der Schaltungsentwicklung als Bibliothekselement bereitgestellt, ähnlich wie die Logikgatter oder Flip-Flops. Je nach Technologie sind bestimmte Speichergrößen vorgegeben oder können, in gewissen Grenzen, frei mit einem Generator erzeugt werden.

Ein Embedded-SRAM kann auch mehr als ein Speicher-Interface haben. Häufig werden Dual-Port-Speicher eingesetzt, die zwei unabhängige Zugriffe unterstützen. Beide Anschlüsse können verschiedene Takteingänge besitzen und somit auch Daten aus einem Taktbereich in einen anderen Taktbereich überführen. Durch die Adressierung muss sichergestellt werden, dass keine Konflikte durch gleichzeitigen Schreibzugriff auf die gleiche Speicherstelle auftreten.

Die Anschlüsse haben jeweils eigene Adresseingänge. Als Datenleitungen sind entweder für beide Anschlüsse Dateneingang und -ausgang vorhanden oder ein Anschluss ist ein Eingang, der andere Anschluss ein Ausgang. Auch mehr als zwei Anschlüsse sind prinzipiell für ein Embedded-SRAM möglich, werden aber selten verwendet.

Als Anwendungsbeispiel soll ein Audiosignal mit einem Halleffekt digital verfremdet werden. Dazu wird das Signal verzögert und mit reduziertem Pegel zum Eingangssignal addiert. Für die Verzögerung kann ein SRAM eingesetzt werden, in das permanent die aktuellen Signalwerte gespeichert und von anderer Adresse frühere Signalwerte gelesen werden.

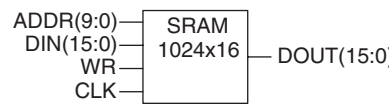


Abb. 11.14 Eingangs- und Ausgangssignale eines Embedded-SRAM

11.3.2 DRAM

Ein DRAM bietet eine deutlich höhere Speicherkapazität als SRAM, erfordert jedoch einen speziellen CMOS-Prozess. Embedded-DRAM wird in der Praxis eingesetzt, wenn große Datenmengen gespeichert werden sollen. Durch eine Kombination von Speicher und Signalverarbeitung sind sehr kompakte Systeme möglich.

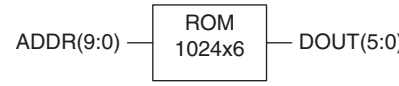
Embedded-DRAM lohnt sich meist nur in Einzelfällen. Sehr große Datenmengen übersteigen die Speicherkapazität und erfordern mehrere externe Speicherchips. Bei kleineren bis mittleren Datenmengen wird Embedded-SRAM verwendet. Dies erfordert zwar mehr Chipfläche, ist aber kostengünstiger, da kein spezieller CMOS-Prozess verwendet werden muss.

Ein Beispiel ist der Grafik-Prozessor SM768 von Silicon Motions mit 256 MByte Embedded-DRAM. Er erzeugt eine Grafik für einen Monitor und kann direkt an ein LCD-Panel angeschlossen werden. Der Baustein wird über USB 3.0 angesteuert, ohne dass eine Grafikkarte nötig ist. Auch komprimierte Videodaten können decodiert werden. Dadurch dass sich Speicher und Signalverarbeitung auf einem einzigen Baustein befinden, ermöglicht dieser einzelne Chip den kostengünstigen Aufbau eines intelligenten Monitors.

11.3.3 ROM

Festwertspeicher können, genau wie SRAMs, mit der normalen CMOS-Fertigungstechnik hergestellt werden. Damit eignen sie sich, wenn in einer Schaltung vorab festgelegte Informationen abgespeichert werden sollen. Eingesetzt werden ROMs beispielsweise für den *Boot-Code* einer CPU, also die fest vorgegebenen Anweisungen beim Starten eines Rechnersystems.

Ein weiteres Einsatzgebiet für ROMs ist die Verwendung als Tabelle für arithmetische Operationen. Als Beispiel hierfür nehmen wir an, dass in einer Digitalschaltung die Wurzel von einer Dualzahl mit der Wortbreite 10 bit benötigt wird. Der Ausgabewert soll auf ganze Zahlen gerundet werden. Die Ergebnisse dieser Rechenoperation können



ADDR (in hex)	Zahlenwert	Wurzel	Wurzel gerundet	DOUT (in hex)
000	0	0	0	00
001	1	1	1	01
002	2	1,41	1	01
003	3	1,73	2	02
...	...	...	...	...
123	291	17,06	17	11
...	...	...	...	...
3FF	1023	31,98	32	20

Abb. 11.15 Symbol und Ausschnitt der Wertetabelle für ein ROM zur Wurzel-Berechnung

vorab berechnet und in einem ROM gespeichert werden. Die Eingangswerte betragen 0 bis 1023, die Wurzel hiervon ist 0 bis 31,98, gerundet 0 bis 32. Für den Ausgabewert sind also 6 Bit erforderlich. Das ROM umfasst 1024 Worte mit jeweils 6 bit Wortbreite. Der Eingangswert wird als Adresse an das ROM angelegt. Am Ausgang wird das Ergebnis der Wurzelberechnung anzeigt. Die Schnittstelle des ROMs und ein Ausschnitt der Wertetabelle sind in Abb. 11.15 gezeigt.

11.3.4 NVRAM

Ein nichtflüchtiger Speicher (NVRAM) erfordert, genau wie ein DRAM, einen speziellen CMOS-Prozess. Anders als beim DRAM gibt es jedoch keine Alternative, wenn in einem Chip Daten auch ohne Versorgungsspannung gespeichert werden sollen. In diesem Fall muss ein CMOS-Prozess mit Erweiterung für NVRAM eingesetzt werden.

Ein häufig eingesetztes Anwendungsbeispiel sind Mikrocontroller. Auf einem einzigen Chip sind eine CPU, Peripherie und der Programmspeicher integriert. Damit der Mikrocontroller durch die Anwender programmiert werden kann, ist der Programmspeicher als NVRAM implementiert. Während der Programmentwicklung kann der Programmspeicher immer wieder umprogrammiert werden. Ebenfalls gibt es FPGAs, die programmierbare Logik und die Speicherung der Konfiguration in einem NVRAM kombinieren.

Alternativ kann das System auch auf zwei Chips aufgeteilt werden. Ein Chip in Standard-CMOS enthält den Mikrocontroller oder das FPGA und ein zweiter Speicher-Chip enthält den Programmspeicher oder die Konfiguration.

Ein Anwendungsbeispiel ist der ATmega328-Controller der Firma Atmel, welcher auf der populären Mikrocontroller-Platine Arduino Uno verwendet wird. Der ATmega328 enthält zwei Blöcke NVRAM.

- Ein Programmspeicher von 32 kByte.
- Ein Datenspeicher von 1 kByte, der vom Programm gelesen und beschrieben werden kann.

11.4 Diskrete Speicherbausteine

Wenn in einem digitalen System größere Datenmengen gespeichert werden müssen, werden hierzu häufig *diskrete Speicherbausteine* eingesetzt. Das System besteht dann aus mehreren Chips, also zum einen aus Signalverarbeitungschips, gefertigt in einem Standard-CMOS-Prozess, zum anderen aus einem oder mehreren Speicher-Chips, gefertigt in speziellen CMOS-Varianten.

11.4.1 Praktischer Einsatz

Ein Beispiel hierfür ist ein PC. Er enthält auf dem Motherboard unter anderem die Chips für CPU und Chipset, gefertigt in Standard-CMOS. Als Hauptspeicher wird DRAM eingesetzt, der sich auf steckbaren Speichermodulen befindet. Jedes Speichermodul enthält mehrere, beispielsweise acht, DRAM-Chips. Der Boot-Code für das PC-System, bekannt als BIOS (Basic Input Output System), sowie Grundeinstellungen befinden sich in einem NVRAM.

11.4.1.1 Systemaufbau

Eine Aufteilung des Systems unter Nutzung diskreter Speicherbausteine hat mehrere Vorteile.

- Die Kapazität externer Speicherbausteine ist höher, als bei gemeinsamer Nutzung der Chipfläche für Speicher und Signalverarbeitung.
- Höhere Flexibilität des Systems, weil je nach Bedarf mehr oder weniger externer Speicher angebunden werden kann.
  - Im oben genannten PC-System können DRAM-Riegel, je nach Bedarf eingesetzt werden.
  - Einige Smartphones werden mit unterschiedlicher Speicherkapazität verkauft. Auf den Geräten sind dann unterschiedliche NVRAMs verbaut.
- Externe Speicherbausteine sind gut verfügbar. Sie können, auch in kleinen Stückzahlen, kurz nach Markteinführung bei Distributoren gekauft werden. Dies ist nicht der Fall bei Chips mit Embedded-DRAM, die nur von wenigen Chipherstellern angeboten werden und häufig Großkunden vorbehalten sind. Auch für Embedded-NVRAM ist die Anzahl an Chipherstellern geringer als für Standard-CMOS-Speichertechnologien.
- Neue Speichertechnologien werden zunächst für den Massenmarkt der diskreten Speicherbausteine angeboten. Meist sind sie nur mit einer signifikanten Verzögerung von einem Jahr oder mehr als Embedded-Speicher verfügbar.
- Die Kosten für einen Chip mit Standard-CMOS-Technologie sind geringer als für einen Chip, der einen speziellen Herstellungsprozess mit Embedded-Speicher-Unterstützung benötigt. Die Einsparung ist in der Regel so hoch, dass sie auch die Kosten für die diskreten Bauelemente deckt.

Der Einsatz von diskreten Speicherbausteinen kann jedoch auch Nachteile haben.

- Je mehr Bauelemente ein System hat, umso größer ist der Platzbedarf. Dies ist insbesondere für mobile Geräte ungünstig.
- Ein Speicherzugriff auf externe Bauelemente hat eine geringere Bandbreite, da die Anzahl der Leitungen begrenzt und die Geschwindigkeit externer Signalleitungen geringer ist. Außerdem ist die Verlustleistung höher, da größere Leitungskapazitäten umgeladen werden müssen.

- Es muss sichergestellt werden, dass die verwendeten Speicherbausteine für die Produktlebensdauer verfügbar sind. Im PC-Bereich werden Bauteile oft nach wenigen Jahren durch leistungsfähigere Neuentwicklungen ersetzt. Für einen PKW müssen hingegen jahrzehntelang Ersatzteile verfügbar sein.

11.4.1.2 Aktuelle Speicherbausteine

Für flüchtige Datenspeicherung werden in der Praxis am häufigsten DRAM-Speicherbausteine eingesetzt. Der Grund dafür ist die höhere Speicherdichte eines DRAM, also Bits pro Siliziumfläche, verglichen mit einem SRAM. An diesen Marktverhältnissen wird sich auch in Zukunft wenig ändern.

Für nicht-flüchtige Datenspeicherung werden hauptsächlich EEPROMs in der Ausführung als NAND-Flash eingesetzt. Die NOR-Flash-Technologie hat den Nachteil der geringeren Speicherkapazität und darum nur einen kleinen Marktanteil. Innovative Speichertechnologien sind noch nicht so weit entwickelt, dass sie den Marktanteil von NAND-Flash-EEPROMs erreichen. Dies kann sich jedoch in den nächsten Jahren ändern.

Im Folgenden sind exemplarisch vier Speicherbausteine beschrieben, die in der Praxis weite Verbreitung haben oder exemplarisch für ähnliche Bausteine sind. Dazu wurden ein SRAM, ein DRAM, ein EEPROM und ein innovatives NVRAM ausgewählt. Sie werden in kompatibler Form von mehreren Herstellern angeboten und bieten dadurch höhere Sicherheit der Verfügbarkeit.

Die Entwicklung neuer Speicherbauelemente baut üblicherweise auf den Vorgängern auf. Das heißt, die Eigenschaften, die in den folgenden Abschnitten beschrieben sind, finden sich in ähnlicher Weise in den Vorgängern und sind Grundlage für die Spezifikation der nächsten Speichertechnologie.

11.4.2 QDR-II-SRAM

11.4.2.1 Übersicht

QDR bezeichnet eine Familie von Dual-Port-SRAMs, die also zwei Anschlüsse haben. Ein Anschluss ist ein Schreib-Interface, der andere ein Lese-Interface. Beide Anschlüsse übertragen Daten bei steigender und fallender Taktflanke (Double-Data-Rate), so dass als Bezeichnung *Quad-Data-Rate (QDR)* gewählt wurde. Es gibt verschiedene Geschwindigkeitsstufen der QDR-Familie. Hier soll QDR-II betrachtet werden, mit ‚II' im Sinne der römischen Zahl Zwei.

Das Einsatzgebiet dieser Speicherbausteine sind insbesondere Anwendungen, die eine sehr hohe Datenrate benötigen und bei denen Lese- und Schreiboperationen etwa gleich häufig vorkommen. Ein Anwendungsbeispiel sind Netzwerkanwendungen, bei denen Datenpakete zwischengespeichert werden müssen.

Die SRAMs werden mit unterschiedlichen Speichergrößen im Bereich von 18 bis 144 Mbit und Datenwortbreiten von 9, 18 und 36 bit angeboten. Ein typischer Baustein

ist der CY7C1514KV18 von Cypress, mit einer Speicherkapazität von 72 Mbit und 36 bit Datenwortbreite. Die Taktgeschwindigkeit darf 350 MHz betragen. Vergleichbare Bausteine werden unter anderem von IDT und Renesas angeboten. Der Speicher arbeitet mit Vielfachen von 9 bit, nicht 8 bit, da in der Telekommunikation häufig zusätzliche Bits zur Fehlererkennung verwendet werden.

Der Speicherbaustein hat folgende Anschlüsse:

- *A,* 20 Bit, Adresse, gemeinsame für Schreib- und Lese-Interface
- *D,* 36 Bit, Dateneingang
- *Q,* 36 Bit, Datenausgang
- */WPS,* Write-Port-Select aktiviert einen Schreibzugriff
- */RPS,* Read-Port-Select aktiviert einen Lesezugriff
- *K* und */K,* Takt für Schreib-Interface in positiver und negativer Polarität
- *C* und */C,* Takt für Lese-Interface in positiver und negativer Polarität
- *CQ* und */CQ,* Ausgabe des Takts *C* für Anpassung an Laufzeiten
- *VREF,* Referenzspannung für Datenleitungen
- weitere Pins für Steuerfunktionen, Stromversorgung und Fertigungstest

Insgesamt hat das Chipgehäuse 165 Pins. Der Schrägstrich (/) kennzeichnet Low-aktive Signale.

Auffällig ist die hohe Anzahl an Taktanschlüssen. Die Takte für Lese-Interface und Schreib-Interface sind in beiden Polaritäten vorhanden. Außerdem wird der Lesetakt in beiden Polaritäten wieder aus dem Speicherbaustein ausgegeben. Die Takte sind nicht unabhängig voneinander, sondern es handelt sich um den gleichen Takt mit unterschiedlichen Verzögerungen. Dieser Aufwand ist nötig, da bei den verwendeten hohen Taktfrequenzen die Laufzeit der Signale auf der Platine beachtet werden muss. In der Konfiguration mit 36 bit Wortbreite sind 333 MHz möglich, die einer Periodendauer von 3 ns entsprechen. Aufgrund der Anwendung der Double-Data-Rate-Technik hat jedes Datenwort nur eine Dauer von 1,5 ns.

11.4.2.2 Logisches Interface

Die Adressierung des SRAMs erfolgt stets abwechselnd für Lese- und Schreib-Interface. Abb. 11.16 gibt ein Beispiel für den Zeitablauf. Im oberen Bereich sind sechs Eingänge des SRAMs, im unteren Bereich drei Ausgänge dargestellt. Für das Taktinterface sind verschiedene Konfigurationen möglich. *K* als primärer Takt ist stets erforderlich, die Verwendung von C und CQ ist optional. In diesem Beispiel wird kein separater Lesetakt C, aber die Taktausgabe CQ verwendet.

Der Zeitablauf zeigt drei Lesezugriffe auf die Adressen a0 bis a2, sowie vier Schreibzugriffe auf die Adressen a4 bis a7. Die Zugriffe erfolgen immer als *Burst* (Sequenz) von zwei Datenworten, das heißt, pro Adresse werden immer zwei 36-Bit-Worte angesprochen. Damit sind für die Speicherkapazität von 72 Mbit 20 Adressleitungen nötig.

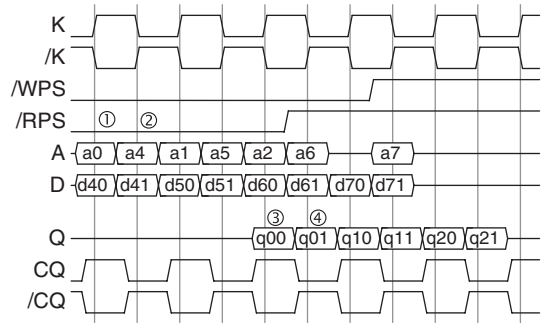


Abb. 11.16 Zeitablauf der Ansteuerung eines QDR-II-SRAMs

Zunächst wird die Leseoperation betrachtet. Die Adresse bei der steigenden Taktflanke von K ist immer die Leseadresse. Zum Zeitpunkt ① wird die Adresse a0 angegeben und durch /RPS auf 0 (Low-aktiv) ein Lesevorgang angezeigt. Der Zugriff auf das SRAM benötigt etwas Zeit, deswegen werden die Daten nach einer Latenzzeit von (hier) zwei Takten ausgegeben. Zum Zeitpunkt ③ wird das erste Datenwort mit der Bezeichnung q00 ausgegeben; einen halben Takt später bei ④ folgt das zweite Datenwort des Burst q01. Durch /RPS auf 0 folgen noch zwei weitere Datenzugriffe auf die Adressen a1 und a2, die Daten folgen unmittelbar auf den ersten Burst. Danach wird /RPS zu 1 und es folgen keine weiteren Leseoperationen.

Das Lese-Interface gibt auch CQ und /CQ als Hilfssignale für die Datenübernahme aus. CQ und /CQ haben ihre Taktflanken an der gleichen Position wie der Datenausgang. Das System, welches die Daten empfängt, kann hieraus den Takt für die Datenübernahme erzeugen.

Die Schreiboperation beginnt auch bei der steigenden Taktflanke von K, verwendet aber die Adresse einen halben Taktzyklus später an der steigenden Taktflanke von /K. Die erste Schreiboperation beginnt also zum Zeitpunkt ① mit dem ersten Datenwort d40 und dem Steuersignal /WPS. Dann folgt zum Zeitpunkt ② die Adresse a4, und das Datenwort d41. Auf eine Adresse werden mit den Datenworten d40 und d41 also insgesamt 72 Bit geschrieben. Im Diagramm werden vier Bursts von jeweils zwei Datenworten geschrieben. Danach wird /WPS zu 1 und das Schreib-Interface ist inaktiv.

Übrigens müssen mit einem Schreibzugriff nicht immer 72 Bit geschrieben werden. Über das Steuersignal Write-Byte-Select (in Abb. 11.16 nicht dargestellt) können Teile des Datenwortes ausgewählt werden.

11.4.2.3 Physikalisches Interface

Zusätzlich zur logischen Ansteuerung sind Zeitanforderungen und Spannungspegel zu beachten. Bei den Zeitanforderungen sind dies Setup- und Hold-Zeiten der Eingangssignale, sowie Vorgaben zum Duty Cycle der Takte und deren Zeitversatz.

Wird das oben genannte SRAM mit 333 MHz Takt betrieben, beträgt die Zykluszeit 3 ns und für Daten und Adresse steht die halbe Zykluszeit von 1,5 ns zur Verfügung. Die Zeitvorgaben sind in diesem Fall:

- Setup- und Hold-Zeit jeweils 0,3 ns
- Duty Cycle des Takts zwischen 40 % und 60 %
- Abstand der steigenden Flanken von K und /K mindestens 1,35 ns, also 45 % der halben Zykluszeit.
- Initialisierungszeit, also Zeit zwischen Anlegen der Spannungsversorgung und erstem Datenzugriff, 1 ms

Als Spannungspegel sind drei verschiedene Versorgungsspannungen definiert, und zwar

- Core-Spannung von 1,8 V für die komplette interne Logik
- I/O-Spannung von 1,5 V für die Ein- und Ausgangspins
- Referenzspannung von 0,75 V für die Erkennung der Datenpegel

Die Logikpegel der Signaleingänge sind in Relation zur Referenzspannung definiert. Der Low-Pegel muss 0,1 V kleiner, der High-Pegel 0,1 V größer als die Referenzspannung sein. Damit reicht also ein Spannungshub von 0,2 V aus.

Darüber hinaus gibt es weitere Vorgaben, unter anderem die maximal erlaubten Spannungen, die Stromaufnahme und weiteren Zeitanforderungen. Diese sind in den Datenblättern der QDR-II-SRAMs angegeben.

11.4.3 DDR3-SDRAM

11.4.3.1 Übersicht

DRAM-Speicherbausteine haben eine deutlich höhere Speicherkapazität als SRAMs und sind damit kostengünstiger. Allerdings ist die mögliche Datenrate geringer und die Ansteuerung deutlich komplexer, da nach jedem Lesevorgang die Information in den Speicherzellen wiederhergestellt werden muss (vgl. Abschn. 11.2.2). Außerdem ist ein regelmäßiger Refresh erforderlich.

DDR3-SDRAM ist eine moderne Familie von DRAM-Speichern mit einer Kapazität von bis zu 4 Gbit, also rund 30mal so viel wie ein QDR-II-SRAM. Das ‚S' in SDRAM steht für *synchron* und gibt an, dass der Baustein mit einem Takt arbeitet. In diesem Abschnitt wird exemplarisch der Baustein MT41J512M8 von Micron Technology betrachtet, ein 4 Gbit DDR3-DRAM mit einer Datenwortbreite von 8 bit. Vergleichbare Bausteine werden unter anderem von Samsung und Hynix angeboten. Verwendet wird der Baustein beispielsweise in DRAM-Modulen für PCs.

Der Baustein wird mit verschiedenen Geschwindigkeiten angeboten. Die Taktfrequenz darf knapp über 1 GHz betragen. Es gibt nur ein Speicherinterface mit bidirektionalen Datenleitungen. Die wesentlichen Anschlüsse sind:

- *A,* 16 Bit, Adresse
- *BA,* 3 Bit, Bankadresse, wählt eine von acht internen Speicherbänken aus
- *DQ,* 8 Bit, Datenbus, bidirektional als Dateneingang und Datenausgang
- *DQS* und */DQS,* Referenzsignal für das Ausgangstiming
- */RAS, /CAS, /WE,* Steuersignale für Lese- und Schreiboperationen
- *CK* und */CK,* Takt in positiver und negativer Polarität
- *VREF_DQ,* Referenzspannung für Datenleitungen
- *VREF_CA,* Referenzspannung für Steuerleitungen
- weitere Pins für Steuerfunktionen, Stromversorgung und Fertigungstest

Das Gehäuse hat 78 Anschlüsse, also weniger als die Hälfte, verglichen mit dem QDR-II-SRAM.

11.4.3.2 Logisches Interface

Das DRAM muss beim Start zunächst initialisiert werden. Für die Ansteuerung muss dann beim Lesen, Schreiben und Refresh der innere Aufbau beachtet werden. Das Arbeitsprinzip wird am besten deutlich, wenn der Lesevorgang betrachtet wird.

Beim Lesen wird eine komplette Zeile in den Schreib/Lese-Verstärker geladen. Dabei wird die Ladung in den Speicherzellen gelöscht und muss wieder „zurückgeschrieben" werden. Dieses Lesen und Zurückschreiben benötigt mehrere Taktzyklen. Während dieser Zeit ist das DRAM blockiert. Darum sind in einem DRAM-Chip acht unabhängige Speicherbänke verfügbar. Während eine Bank noch durch Zurückschreiben von Daten belegt ist, kann bereits auf eine andere Bank zugegriffen werden.

Der Lesezugriff auf den Speicher erfolgt in drei Schritten.

- **Activate:** Hierdurch wird eine Zeile in den Leseverstärker geladen.
- **Read:** Aus der Zeile werden Datenworte gelesen. Mehrere Leseoperationen für die aktivierte Zeile sind möglich und jede Leseoperation liest einen Burst von vier oder acht Worten.
- **Precharge:** Der Zugriff auf die Zeile wird beendet und die Daten wieder in die Speicherzellen zurückgeschrieben.

Die Schritte werden durch die Steuersignale /RAS, /CAS und /WE aufgerufen. Zwischen den Schritten gibt es Wartezeiten von mehreren Takten, die eingehalten werden müssen. Nach Activate können ebenfalls Schreiboperationen in die Zeile erfolgen, auch abwechselnd mit Leseoperationen.

Abb. 11.17 zeigt den Zeitablauf für zwei Leseoperationen auf zwei verschiedene Bänke. Als Burst sind 8 Worte gewählt. Die invertierten Signale /CK und /DQS sind zur

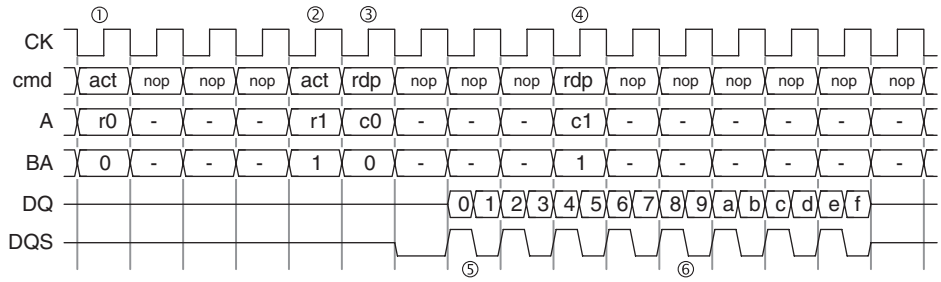


Abb. 11.17 Zeitablauf zweier Leseoperationen eines DDR3-SDRAMs

besseren Übersicht weggelassen. Die Steuersignale /RAS, /CAS, /WE sind zum Kommandowort ‚cmd' (Command) zusammengefasst. Die eingezeichneten Zeitpunkte haben folgende Bedeutung:

1. Aktivierung der Zeile r0 (r wie Row) in der Bank 0 mit dem Kommando ‚act' (Activate). Bevor die Zeile verwendet werden kann, muss mehrere Takte gewartet werden.
2. Aktivierung der Zeile r1 in der Bank 1.
3. Lesezugriff auf Spalte c0 (c wie Column) in der Bank 0. Nach Ausführen der Leseoperation soll die Zeile durch Precharge zurückgeschrieben werden. Als Kommando wird darum ‚rdp' (Read with Precharge) aufgerufen.
4. Lesezugriff auf Spalte c1 in der Bank 1, ebenfalls mit Precharge.
5. Nach einer Latenzzeit werden die Daten des Lesezugriffs ③ ausgegeben. Entsprechend der Burst-Länge werden acht Daten von 0 bis 7 ausgegeben. Als Hilfssignale für die Datenübernahme wird DQS ausgegeben. Die Taktflanken sind an der gleichen Position wie der Datenausgang und das System, welches die Daten empfängt, kann hieraus den Übernahmetakt erzeugen.
6. Direkt nach dem ersten Datenburst werden die Daten des Lesezugriffs ④ ausgegeben. Dies sind die Daten 8 bis f.

Die Bezeichnung *nop* (No Operation) gibt an, dass kein Kommando übertragen wird. Bitte beachten Sie, dass in Abb. 11.17 die Abstände zwischen den Kommandos etwas verkürzt dargestellt sind. Die internen Vorgänge benötigen bestimmte Zeiten, die einer Anzahl an Taktzyklen entsprechen. Deswegen werden mit steigender Taktfrequenz mehr Taktzyklen für bestimmte Abläufe benötigt.

Die maximale Taktfrequenz und die Wartezeiten werden als Kennziffern des DRAMs angegeben und sind Ihnen vielleicht schon begegnet, wenn Sie Speicherriegel für den PC gekauft oder die Werte im BIOS eingeben haben. Die Bezeichnung DDR3-1866 CL13 13-13-32 bedeutet beispielsweise:

- DDR3-1866: DDR3-SDRAM mit 1866 Mio. Transfers je Sekunde, also einer maximalen Taktfrequenz von 933 MHz.

- CL13 ist die Anzahl der Taktzyklen zwischen Read und Ausgabe der Daten. CL steht für Column Access Latency oder CAS Latency.
- Die folgenden drei Zahlen bezeichnen weitere Zeiten
  - 13 Taktzyklen zwischen dem Activate-Befehl einer Zeile und erstem Read-Zugriff
  - 13 Taktzyklen für den Precharge-Vorgang
  - 32 Taktzyklen zwischen zwei Activate-Befehlen auf dieselbe Bank

Der Zugriff auf ein DRAM erfordert also das Beachten der internen Speicherorganisation. Eine hohe Datenrate kann erreicht werden, wenn mehrere Daten aus der gleichen Zeile gelesen werden (nur ein Activate-Befehl nötig) und die Zugriffe ansonsten auf verschiedene Speicherbänke verteilt werden (Wartezeit zwischen Activate-Befehlen auf dieselbe Bank).

Diese Zugriffsmuster werden beispielsweise von den CPUs in einem PC berücksichtigt. Der Speichercontroller einer CPU liest größere Datenblöcke aus dem DRAM und speichert sie auf einem internen SRAM, dem sogenannten *Cache*. Die Daten sind so im DRAM abgelegt, dass ein Zugriff möglichst effizient erfolgen kann.

11.4.3.3 Physikalisches Interface

Das physikalische Interface des DDR3-SDRAMs nutzt ähnliche Prinzipien wie das QDR-II-SRAM. Da noch höhere Frequenzen auftreten können, sind die Anforderungen entsprechend höher.

Für ein DDR3-1866-SDRAM beträgt die Taktfrequenz 933 MHz Takt und somit ist die Zykluszeit 1,07 ns. Der Duty Cycle des Takts muss zwischen 47 % und 53 % liegen. Anstelle fester Setup- und Hold-Zeit für die Signaleingänge werden Grenzen für den Zeitverlauf der Spannung definiert. Darin ist auch festgelegt, wie stark ein Überschwingen der Signale erfolgen darf. Die Adress- und Steuerleitungen werden nur einmal pro Taktzyklus ausgewertet, während Datenleitungen zweimal pro Taktzyklus gültig sind. Daher wird zwischen diesen Signalen unterschieden.

Die Spannungsversorgung für Core und I/O beträgt 1,5 V, die Referenzspannung zur Erkennung der Datenpegel ist 0,75 V.

Spezifische Angaben zum physikalischen und logischen Interface finden Sie im Datenblatt eines DDR3-SDRAMs, beispielsweise dem MT41J512M8 von Micron.

11.4.4 EEPROM

11.4.4.1 Übersicht

Im Bereich der EEPROMs gibt es eine große Vielfalt an unterschiedlichen diskreten Speicherbausteinen. Es gibt kleine, mittlere und große Speichergrößen, sowie langsamen und schnellen Speicherzugriff.

- Kleine Speichergrößen im Bereich von einigen kByte, werden beispielsweise verwendet, um Geräteeinstellungen zu speichern, wie Netzwerkname, WLAN-Passwort und IP-Adresse eines Netzwerkgeräts.
- Mittlere Speichergrößen, im Bereich von MByte, werden beispielsweise zum Speichern von Messdaten oder von Programmcode für größere Prozessoren verwendet.
- Große Speichergrößen, im Bereich von GByte, werden als Massenspeicher verwendet, beispielsweise im Smartphone oder als Solid-State-Disk (SSD).

Bei kleineren Speichergrößen kann teilweise jedes Datenwort einzeln gelöscht werden. Mittlere und große Speichergrößen werden als Flash-EEPROM implementiert.

Der Speicherzugriff kann seriell über eine Datenleitung oder parallel über mehrere Leitungen erfolgen.

- Der serielle Zugriff ist langsamer, aber ausreichend, wenn nur wenige Daten benötigt werden oder wenn die Daten einmalig gelesen und dann auf dem System zwischengespeichert werden.
- Der parallele Zugriff ist schneller und für größere Datenmengen sinnvoll.

Aus den unterschiedlichen Anforderungen ergibt sich eine Vielfalt an diskreten EEPROM Speicherbausteinen. SRAM und DRAM Bausteine werden nur eingesetzt, wenn die Speicherkapazität auf einem Chip nicht ausreicht. Ein EEPROM Baustein ist jedoch bereits erforderlich, wenn nur wenige Byte nichtflüchtig gespeichert werden sollen, da ein Chip in Standard-CMOS-Technologie dies nicht bietet.

11.4.4.2 8 Gbit Flash-Memory

Als ein Beispiel für ein EEPROM mit großer Speicherkapazität wird der Baustein TH58NVG3S0HTA00 von Toshiba mit einer Speichergröße von 8 Gbit, also 1 GByte betrachtet. Es handelt sich dabei um ein NAND-Flash-EEPROM. Andere Anbieter von NAND-Flash-EEPROMs sind beispielsweise Cypress, Micron, Samsung und Winbond.

Der Baustein ist in 4096 Blöcken organisiert und jeder Block hat 64 „Speicherseiten“ mit jeweils 4352 Bytes. Dieser Inhalt einer Seite umfasst 4096 Bytes Nutzdaten sowie 256 Bytes für Speicherverwaltung und die bei der NAND-Struktur nötige Fehlerkorrektur. Ein Flash-Löschvorgang bezieht sich immer auf einen Block von 64 Seiten, also 256 kByte.

Der Baustein ist darauf ausgelegt mit einem fehlerkorrigierenden Controller zusammenzuarbeiten. Das Speicherinterface arbeitet ohne Takt und hat die folgenden Anschlüsse:

- *IO,* 8 Bit, I/O-Port
- *CLE,* Command Latch Enable, Übernahmesignal für Befehle
- *ALE,* Address Latch Enable, Übernahmesignal für Adresse
- */CE,* Chip Enable

- */WE,* Write Enable
- */RE,* Read Enable
- *RY/BY,* Ready/Busy, zeigt an, ob der Baustein noch einen Befehl ausführt
- */WP,* Write Protect, für einen Schreibschutz
- Pins für Stromversorgung

Das Gehäuse hat 48 Anschlüsse, von denen jedoch ein größerer Teil nicht verwendet wird. RY/BY ist ein gleichzeitig High-aktives Ready- und Low-aktives Busy-Signal.

11.4.4.3 Logisches Interface

Der 8-Bit-Port *IO* wird gemeinsam für Kommandos, Adressen und Daten verwendet. Kommandos werden durch bestimmte 8-Bit-Werte übermittelt. *CLE* und *ALE* zeigen an, um welche Information es sich jeweils handelt.

Die drei Grundoperationen des Bausteins sind Löschen eines Blocks, Schreiben von Daten und Lesen von Daten.

Löschen eines Blocks

Abb. 11.18 zeigt den Zeitablauf beim Löschen eines EEPROM-Blocks.

1. Der Löschvorgang wird durch ein spezielles Kommando gestartet. Dazu liegt der Wert 60_{16} auf den acht Datenleitungen und CLE zeigt an, dass es sich bei dieser Information um ein Kommando handelt. Die Datenübernahme erfolgt durch die steigende Flanke an /WE.
2. Die Adresse des zu löschenden Blocks wird auf der Datenleitung übertragen. Da die Datenleitung kleiner als die Adresswortbreite ist, wird die Adresse in drei Teile A-0, A-1, A-2 aufgeteilt.
3. Das Kommando $D0_{16}$ löst den Löschvorgang aus. Durch RY/BY wird angezeigt, dass der Baustein beschäftigt ist. Das Löschen eines Blocks benötigt etwa 2,5 bis 5 ms.
4. Nach Ende des Löschvorgangs muss überprüft werden, ob das Löschen erfolgreich war. Dazu wird das Kommando 70_{16} angegeben.
5. Der Baustein antwortet mit einem Statuswort (sts). Für diese Leseoperation wird /RE angesteuert.

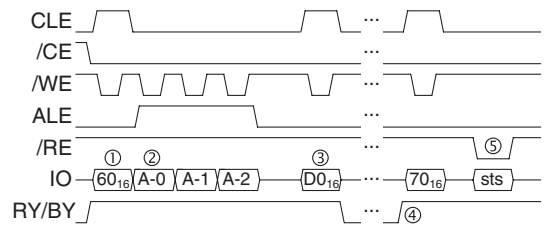


Abb. 11.18 Zeitablauf beim Löschen eines EEPROM-Blocks

Bei einem NAND-EEPROM kann es vorkommen, dass Speicherblöcke fehlerhaft sind oder im Gebrauch fehlerhaft werden. Dies würde durch das Statuswort angezeigt und der Controller verwendet einen solchen Block dann nicht mehr. Für den hier betrachteten Baustein werden 60.000 Löschzyklen angegeben, wobei mit zunehmender Anzahl an Löschzyklen einige Blöcke unbrauchbar werden können. Laut Datenblatt bleiben über die spezifizierte Lebensdauer mindestens 4016 der 4096 Blöcke funktionsfähig.

Schreiben von Daten

Das Schreiben von Daten erfolgt vorzugsweise für einzelne Seiten mit 4096 plus 256 Bytes. Der Zeitablauf ist in Abb. 11.19 dargestellt. Die Steuersignale werden ähnlich wie beim Löschen angesteuert und für bekannte Schritte nicht einzeln erläutert.

1. Das Kommando 80_{16} gibt an, dass ein Schreibvorgang ausgeführt werden soll.
2. Die Adresse von Block und Seite wird in fünf Teilen von A-0 bis A-4 übertragen.
3. Jetzt werden nacheinander die Daten jeweils mit der steigenden Flanke von /WE übertragen. Bis zu 4352 Byte sind möglich, das heißt D-x wäre dann D-4351. CLE und ALE auf 0 zeigen an, dass es sich weder um ein Kommando (CLE) noch um Adressen (ALE) handelt.
4. Das Kommando 10_{16} startet den Schreibvorgang. Die Daten sind bisher in einem internen Zwischenspeicher und werden jetzt in die Speichermatrix geschrieben. Das EEPROM überprüft den Schreibvorgang und versucht eventuell mehrfach zu schreiben. Durch RY/BY wird die Aktivität angezeigt. Die Programmierung einer Seite dauert 300 bis 700 µs.
5. Nach Ende des Schreibvorgangs muss mit dem Kommando 70_{16} überprüft werden, ob das Schreiben erfolgreich war.
6. Der Baustein antwortet mit einem Statuswort (sts).

Für die Verwendung des EEPROMs ist die interne Organisation zu beachten. Der Controller schreibt Daten auf freie Seiten des Speichers. Nicht mehr benötigte Seiten werden nicht sofort gelöscht, sondern zunächst als ungültig gekennzeichnet. Erst wenn alle Seiten eines Blocks ungültig sind, wird ein ganzer Block gelöscht. Hierfür kann es eventuell nötig sein, noch gültige Seiten in andere Blöcke zu kopieren.

Damit der Controller Schreibzugriffe und das Löschen von Blöcken optimieren kann, wird empfohlen, den Speicher nicht komplett zu füllen.

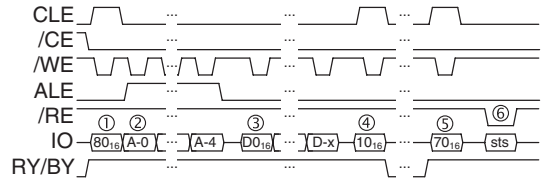


Abb. 11.19 Zeitablauf beim Schreiben in ein EEPROM

Lesen von Daten
Das Lesen von Daten erfolgt ähnlich wie das Schreiben von Daten. Zunächst wird ein Lesekommando gegeben, dann die Leseadresse in fünf Teilen und ein weiteres Kommando. Daraufhin lädt der Baustein die Daten aus der Speichermatrix in den Leseverstärker und gibt nacheinander die Daten ab der angeforderten Adresse aus. Das Lesen der Daten aus der Speichermatrix benötigt 25 µs.

11.4.4.4 Physikalisches Interface

Die Datenrate bei Schreib- und Lese-Zugriffen ist deutlich geringer als bei SRAM und DRAM, denn das EEPROM ist als Massenspeicher und nicht als schneller Arbeitsspeicher vorgesehen.

Die mögliche Datenrate beim Schreiben und Lesen von Daten beträgt 40 MHz. Hinzu kommen die oben genannten Zeiten für den Zugriff auf die Speichermatrix.

Die Spannungsversorgung des Bausteins beträgt 3,3 V. Die Daten werden durch Spannungspegel dargestellt. Der Low-Pegel wird bis 0,66 V erkannt, der High-Pegel ab 2,64 V, dazwischen befindet sich der Übergangsbereich, in dem keine eindeutige Zuordnung der Spannung zu einem logischen Wert möglich ist.

Spezifische Angaben zum physikalischen und logischen Interface finden Sie im Datenblatt.

11.4.5 FRAM mit seriellem Interface

11.4.5.1 Übersicht

Als ein Beispiel für ein NVRAM mit einer innovativen Speichertechnik wird der Baustein MB85RS64V von Fujitsu betrachtet. Es handelt sich um ein Ferroelektrisches RAM mit 8192 Worten zu 8 bit und seriellem Interface. Jede Zelle kann einzeln beschrieben werden. Mit der Speicherkapazität von 8 kByte handelt es sich um eine kleine Speichergröße. Dafür ist der Baustein allerdings auch sehr kompakt und hat ein Gehäuse mit nur 8 Pins. Bausteine mit ähnlichem Interface und Speichergröße sind auch als EEPROM von verschiedenen Herstellern verfügbar.

Als besonderer Vorteil sind für das FRAM 10^{12} mögliche Zugriffe pro Zelle spezifiziert. Für EEPROMs werden üblicherweise 10^5 bis 10^6 Schreibvorgänge angegeben.

Die Anschlüsse des Bausteins sind:

- *SCK,* Serial Clock, Takt für den seriellen Zugriff
- *SI,* Serial Data Input, serieller Dateneingang
- *SO,* Serial, Data Output, serieller Datenausgang
- */CS,* Chip Select, zum Aktivieren des Bausteins
- */WP,* Write Protect, Schreibschutz
- */HOLD,* pausiert einen Zugriff
- Versorgungsspannung und Masse

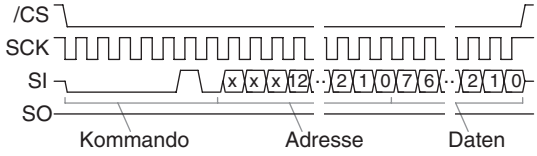


Abb. 11.20 Serielles Schreiben in ein FRAM mit SPI-Protokoll

11.4.5.2 Logisches Interface

Die Kommunikation mit dem Baustein erfolgt über das *Serial Peripheral Interface (SPI).* Bei diesem Protokoll sind getrennte Leitungen für Dateneingang und -ausgang vorhanden, das heißt die Datenleitung wird nicht bidirektional betrieben. Ein Zugriff auf den Baustein erfolgt über Kommandos. Bei Schreib- und Leseoperationen folgt nach dem Kommando eine Adresse und bei einem Schreibzugriff die Daten. Bei einem Lesezugriff antwortet der Baustein nach der Adressübertragung mit den angeforderten Daten.

Abb. 11.20 zeigt den Zeitablauf eines Schreibvorgangs. /CS aktiviert zunächst den Baustein. Dann werden insgesamt 32 Bits durch die steigende Flanke von SCK übertragen, die Most Significant Bits (MSB) jeweils zuerst. Die ersten 8 Bit sind das Schreibkommando 02_{16}. Dann folgt die Adresse mit 16 Bit. Da für 8 kByte nur 13 Bit benötigt werden, sind die obersten drei Adressbits unbelegt. Schließlich werden die Daten übertragen und durch /CS die Übertragung beendet.

Es ist auch möglich, mehrere Bytes an Daten zu übertragen, die dann in aufeinander folgende Adressen geschrieben werden (nicht in Abb. 11.20 dargestellt). Damit ist keine wiederholte Übertragung von Kommando und Adresse nötig.

11.4.5.3 Physikalisches Interface

Die maximale Taktfrequenz für SCK beträgt 20 MHz. Wartezeiten für die Programmierung sind nicht erforderlich. Die Spannungsversorgung des Bausteins beträgt 3,3 bis 5 V. Bei 3,3 Spannungsversorgung wird der Low-Pegel bis 0,66 V, der High-Pegel ab 2,64 V erkannt.

11.5 Speichersysteme

Ein *Speichersystem* ist die Kombination aus mehreren Speichern. Dabei sind verschiedene Konfigurationen möglich. Mehrere gleiche Speicher können kombiniert werden, um die Speicherkapazität zu erhöhen. Verschiedene Speicher können kombiniert werden, wenn unterschiedliche Eigenschaften benötigt werden. Dies kann SRAM- und DRAM-Speicher oder flüchtiger und nicht-flüchtiger Speicher sein. Außerdem können die Speicher sowohl Embedded Speicher auf der Integrierten Schaltung als auch diskrete Speicherbausteine außerhalb sein.

11.5.1 Adressdecodierung

Oft ist es gewünscht, dass die Speicher gemeinsam von einer zentralen Steuerlogik, zum Beispiel der CPU eines Rechnersystems, angesprochen werden sollen. Die Unterscheidung der Speicher erfolgt dann anhand der Speicheradresse. Der *Adressraum* enthält Adressbereiche für die unterschiedlichen Speicher. Je nach Größe von Adressraum und Speicherkomponenten können Adressbereiche auch unbelegt sein.

Der prinzipielle Aufbau eines Speichersystems ist in Abb. 11.21 dargestellt. Die zentrale Steuerlogik gibt eine Adresse sowie Steuersignale an das Speichersystem. Je nach Speichermodul können unterschiedliche Steuersignale sinnvoll und erforderlich sein. Hier sind dargestellt:

- *CS* für Chip Select: Ein Zugriff findet statt
- *WR* für Write: Ein Schreibzugriff findet statt
- *RD* für Read: Ein Lesezugriff findet statt

Ein Adressdecoder ermittelt dann aus der Adresse, welches Speichermodul adressiert ist und gibt an dieses Modul ein individuelles Chip-Select-Signal weiter. Über den Datenbus gibt die Steuerlogik entweder Daten an das Speichermodul oder es werden Daten empfangen.

Aus der Organisation des Adressraums ergibt sich die Adressierung. Zur Verdeutlichung wird das Speichersystem eines fiktiven Rechnersystems in Abb. 11.22 dargestellt. Die Adresse hat eine Wortbreite von 16 bit und kann damit 64 kByte adressieren. Speicherzugriffe erfolgen jeweils auf ein Byte. Es sind drei Speicher vorhanden, und zwar ein ROM von 4 kByte für den Boot-Code, ein SRAM von 8 kByte als Datenspeicher und ein EEPROM von 32 kByte für den Programmcode. Der Adressraum ist links in Abb. 11.22 angegeben. Das Präfix „0x“ kennzeichnet hexadezimale Zahlen. Die Adressbelegung ist wie folgt:

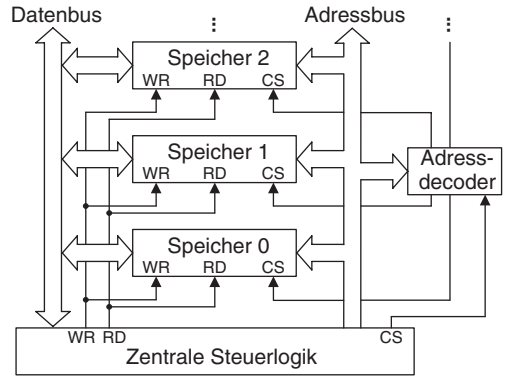


Abb. 11.21 Aufbau eines Speichersystems aus mehreren Speichermodulen

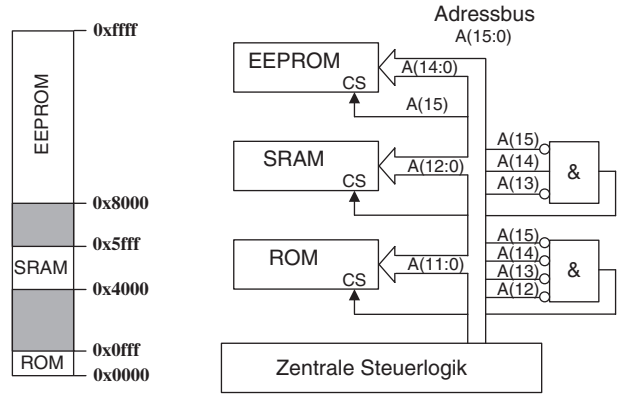


Abb. 11.22 Adressraum und -decoder für ein Speichersystem mit drei Speichermodulen

- 0x8000 – 0xffff: EEPROM
- 0x6000 – 0x7fff: unbelegt
- 0x4000 – 0x5fff: SRAM
- 0x1000 – 0x3fff: unbelegt
- 0x0000 – 0x0fff: ROM

Die Steuerlogik kann also beispielsweise durch Angabe von Adresse 0x0123 auf das ROM sowie durch Adresse 0x4567 auf das SRAM zugreifen.

Auf der rechten Seite von Abb. 11.22 ist die Logik des Adressdecoders abgebildet. Die 16 Adressleitungen werden teils für die Selektion des Speichermoduls verwendet, teils gehen sie in das Speichermodul. Ein Chip-Select-Signal der Steuerlogik wird hier nicht verwendet; die Speicher werden über Read und Write angesteuert.

Die Logik des Adressdecoders und die Wortbreite der Adressen ergeben sich aus Speichergröße und Position im Adressraum.

- Das EEPROM benötigt für 32 kByte Speichergröße eine Adresse der Wortbreite 15 bit. Das oberste Bit der Adresse selektiert den Speicher, wenn die Adresse größer als 0x8000 ist. Als Chip-Select-Signal des EEPROMs kann direkt Adressleitung 15 verwendet werden.
- Das SRAM benötigt für 8 kByte Speichergröße eine Adresse der Wortbreite 13 bit. Die vorderen drei Bit der Adresse selektieren den Speicher für Adressen zwischen 0x4000 und 0x5fff. In diesem Adressbereich sind A(15) bis A(13) gleich 010_2. Das Chip-Select-Signal wird durch ein UND-Gatter mit drei Eingängen, zwei davon negiert, erzeugt.
- Das ROM benötigt für 4 kByte Speichergröße eine Adresse der Wortbreite 12 bit. Die vorderen vier Bit der Adresse selektieren den Speicher für Adressen zwischen 0x0000 und 0x1fff. In diesem Adressbereich sind A(15) bis A(12) gleich 0. Das Chip-Select-Signal wird durch ein UND-Gatter mit vier negierten Eingängen erzeugt.

11.5.2 Multiplexing des Datenbusses

Jetzt betrachten wir den Datenbus auf der linken Seite von Abb. 11.21. Daten können von der Steuerlogik zu einem Speicher oder vom Speicher zur Steuerlogik übertragen werden. Für die Implementierung gibt es zwei Möglichkeiten. Entweder sind getrennte Datenleitungen für Schreib- und Leseoperationen vorhanden, die dann durch Multiplexer ausgewählt werden. Oder es wird eine gemeinsame Datenleitung verwendet, auf die alle Busteilnehmer mit Tri-State-Ausgängen schreiben.

Innerhalb Integrierter Schaltungen werden stets getrennte Datenleitungen für Schreib- und Leseoperationen verwendet. Tri-State-Leitungen sind innerhalb eines ICs zwar technisch möglich, aber für Fertigung und Herstellungstest sehr problematisch. Auch für die Ansteuerung diskreter Speicherbausteine können getrennte Datenleitungen verwendet werden. Beispiele dafür sind das QDR-II-SRAM und das FRAM aus Abschn. 11.4.

Der Schaltungsaufbau bei getrennten Datenleitungen ist in Abb. 11.23 dargestellt, wiederum für das Speichersystem mit EEPROM, SRAM und ROM. Alle drei Speichermodule haben einen Datenausgang, aber nur das SRAM hat einen Dateneingang. Hier wird angenommen, dass die Programmierung des EEPROMs über ein eigenes Programmier-Interface erfolgt (nicht dargestellt); die Steuerlogik schreibt nicht in das EEPROM. Die Richtung von Schreiben und Lesen bezieht sich jeweils auf die Sicht der Steuerlogik. Der Schreibbus führt direkt vom Datenausgang *(D_OUT)* der Steuerlogik an den Dateneingang *(D_IN)* des Speichermoduls. Auch mehrere Speichermodule können an den Schreibbus angeschlossen werden, da nur die Steuerlogik Daten schreiben kann.

Der Lesebus hat mehrere Quellen, und zwar die Datenausgänge aller Speichermodule. Darum ist ein Multiplexing erforderlich, damit nur die Daten des adressierten

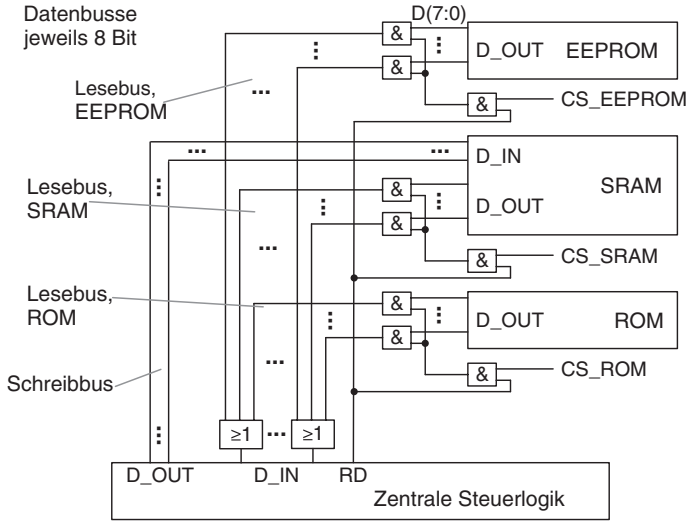


Abb. 11.23 Datenbusse des Speichersystems mit drei Speichermodulen

Speichermoduls an die Steuerlogik gegeben werden (siehe Abb. 11.23). Zunächst wird für jedes Speichermodul die *RD*-Leitung mit der *CS*-Leitung UND-verknüpft. Das verknüpfte Signal ist 1, wenn ein Lesezugriff auf das entsprechende Modul erfolgt. Zum Multiplexing wird der Datenausgang des Speichermoduls durch UND-Gatter weitergegeben. Falls das Modul nicht aktiv ist, bleibt der Ausgang dieser UND-Gatter auf 0. Da immer nur ein Speichermodul adressiert sein kann, ist auch nur ein Lesebus aktiviert und die anderen Lesebusse sind 0. Für den Dateneingang der Steuerlogik werden die einzelnen Lesebusse ODER-verknüpft.

Ein Datenbus mit Tri-State-Leitungen kann für die Ansteuerung diskreter Bauelemente verwendet werden. Dies hat den Vorteil, dass die Datenleitungen gemeinsam zum Lesen und Schreiben verwendet werden, denn die Anzahl der Anschlüsse eines ICs ist begrenzt. Beispiele dafür sind das DDR3-SDRAM und das Flash-EEPROM aus Abschn. 11.4. Beim Flash-EEPROM wurde der Datenbus auch für Kommandoworte und Adresse genutzt, um noch mehr Pins zu sparen.

Abb. 11.24 zeigt ein Speichersystem mit Tri-State-Leitungen. Die Blöcke für Speicher und Steuerlogik stellen jetzt jeweils eigene diskrete Bauelemente dar und sind zur Verdeutlichung mit dickeren Linien gezeichnet. Sowohl die Speicher als auch die Steuerlogik müssen für den Betrieb an einem Tri-State-Bus vorgesehen sein und entsprechende Treiber an den Anschlüssen besitzen. Im Chip der Steuerlogik wird der Tri-State-Treiber durch das Write-Signal angesteuert, bei den Speichern durch UND-Verknüpfung aus Read und jeweiligem Chip-Select-Signal.

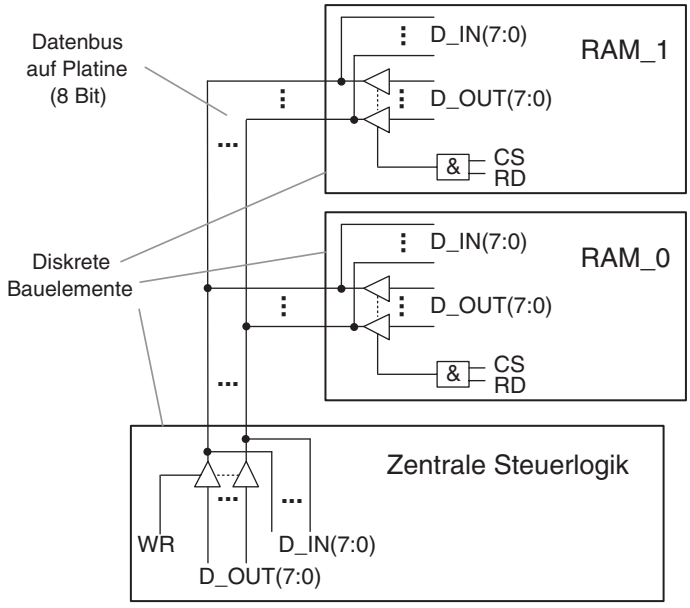


Abb. 11.24 Speichersystem mit diskreten Bauelementen und Tri-State-Leitungen

Die Datenleitungen werden in den Bauelementen gleichzeitig als Ausgang und Dateneingang für die interne Logik genutzt. Eine Steuerlogik (hier nicht dargestellt) entscheidet, ob der Dateneingang verwendet wird.

11.5.3 Ansteuerung diskreter Speicherbausteine

Die vier in Abschn. 11.4 vorgestellten diskreten Speicherbausteine haben Schnittstellen, die unterschiedlich komplexe Ansteuerungen benötigen:

- Das serielle NVRAM kann relativ einfach angesprochen werden.
- Das NAND-EEPROM hat ein recht einfaches Interface, erfordert jedoch Fehlerkorrektur und Beachtung defekter Blöcke.
- Das SRAM hat ein einfaches logisches Interface, erfordert jedoch bei höheren Taktfrequenzen eine spezielle Taktbehandlung sowie eine physikalische Ansteuerung mit Logikpegeln bezogen auf eine Referenzspannung.
- Das DRAM erfordert ein komplexes Protokoll für die Ansteuerung, Beachtung der Bankstruktur sowie Taktbehandlung und physikalische Ansteuerung mit Referenzspannung.

Für die Ansteuerung diskreter Speicherbausteine sind verschiedene Funktionselemente vorhanden, die für den Aufbau eines Systems genutzt werden können.

Logisches Interface
Für die logische Ansteuerung werden Controller angeboten, welche die Ansteuerung der Bausteine ausführen. Diese Controller sind teilweise als VHDL-Code oder Gatter-Netzliste verfügbar und können in eigene Schaltungsentwürfe übernommen werden. Solche Schaltungsbeschreibungen werden als Intellectual Property (IP) bezeichnet und müssen üblicherweise als Lizenz gekauft werden. Für programmierbare Bausteine (FPGAs) werden von den Herstellern IP-Blöcke angeboten. Diese sind für Käufer der FPGAs oft ohne weitere Kosten verfügbar.

Physikalisches Interface
Die physikalische Ansteuerung von SRAMs und DRAMs erfolgt über Pins mit speziellen Logikpegeln. Für Tri-State-Busse sind ebenfalls entsprechende Pins erforderlich. Die Hersteller von ICs und FPGAs bieten diese Ein- und Ausgangstreiber als Bibliothekselement an.

11.6 Übungsaufgaben

Haben Sie den Inhalt des Kapitels verstanden? Prüfen Sie sich mit den Aufgaben und Fragen am Kapitelende. Die Lösungen und Antworten finden Sie am Ende des Buches.

Bei den Auswahlfragen ist immer genau eine Antwort korrekt.

Aufgabe 11.1
Welche der folgenden Technologien ist ein ‚flüchtiger Speicher'?

a) FRAM
b) EPROM
c) Flash
d) SRAM
e) EEPROM

Aufgabe 11.2
Wie viele Transistoren hat eine normale SRAM-Zelle?

a) 9
b) 2
c) 1
d) 5
e) 6

Aufgabe 11.3
Wie viele Transistoren hat eine normale DRAM-Zelle?

a) 6
b) 5
c) 2
d) 1
e) 9

Aufgabe 11.4
Wozu wird beim DRAM ein ‚Refresh' benötigt?

a) Zugriff auf verschiedene Speicherblöcke
b) Zugriff auf Zeilen und Spalten der Speichermatrix
c) Auswahl eines Speicherblocks
d) Aufladen von Kondensatoren
e) Löschen von Datenbereichen

Aufgabe 11.5
Was passiert beim ‚Flash' eines Flash-Speichers?

a) Zugriff auf Zeilen und Spalten der Speichermatrix
b) Auswahl eines Speicherblocks
c) Löschen von Datenbereichen

d) Zugriff auf verschiedene Speicherblöcke
e) Aufladen von Kondensatoren

Aufgabe 11.6
Wie erfolgt die Datenspeicherung in EEPROMs?

a) Kondensator
b) Magnetfeld
c) Rückkopplung von Invertern
d) Transistor mit ‚Floating-Gate'
e) Brennen von Sicherungen

Aufgabe 11.7
a) Wie viele Adressleitungen braucht ein SRAM mit 16K Datenworten?
b) Wie viele Adressleitungen braucht ein SRAM mit 256K Datenworten?

Aufgabe 11.8
a) Ein SRAM hat 16 Adressleitungen und 8 Datenleitungen. Wie hoch ist die Speicherkapazität?
b) Ein SRAM hat 20 Adressleitungen und 16 Datenleitungen. Wie hoch ist die Speicherkapazität?

Aufgabe 11.9
Abb. 11.25 zeigt ein einfaches Speichermodul mit 6 Adressleitungen und 1 bit Wortbreite. Damit soll ein Primzahl-Detektor realisiert werden. Programmieren Sie die Speicherelemente mit 0 oder 1, so dass der Speicher eine 1 ausgibt, wenn eine Primzahl am Eingang anliegt.

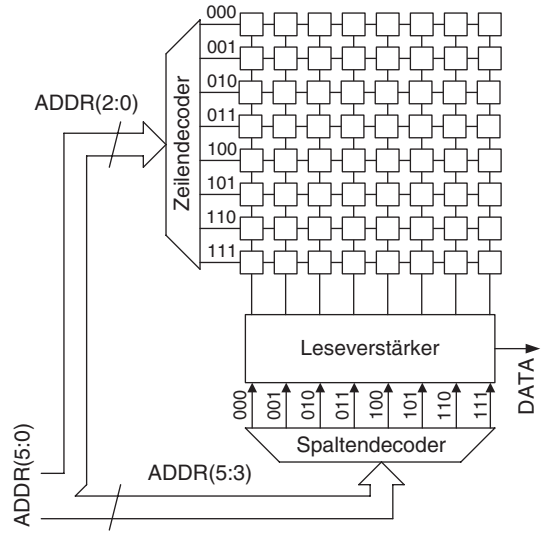


Abb. 11.25 Einfaches Speichermodul für Primzahl-Detektor

Analog-Digital- und Digital-Analog-Umsetzer

12

Analog-Digital-Umsetzer (ADU), engl. *Analog-Digital-Converter (ADC),* sind Bindeglieder zwischen analogen Signalquellen wie Messwandler für Druck, Temperatur, Weg, Beschleunigung, Mikrofonen und digital arbeitenden Systemen. Sie wandeln einen analogen Spannungswert in eine digitale Darstellung. *Digital-Analog-Umsetzer (DAU)* engl. *Digital-Analog-Converter (DAC),* wandeln dann einen digitalen Wert wieder in die analoge Welt.

Technische Herausforderungen beim Einsatz von ADUs und DAUs liegen in den Anforderungen an Genauigkeit und Geschwindigkeit der Umsetzung. Wirtschaftliche Herausforderungen liegen in den Kosten der Umsetzer, denn ein Gesamtsystem kann aus Analog-Digital-Umsetzer, digitaler Verarbeitung und Digital-Analog-Umsetzer bestehen. Im Vergleich mit einem rein analogen System müssen die Kosten vergleichbar sein oder die digitale Verarbeitung muss eine bessere Qualität der Verarbeitung ermöglichen.

Generelle Vorteile der digitalen gegenüber der analogen Technik bestehen unter anderem durch:

- Geringere Störanfälligkeit digitaler Signale, bzw. Fehlerkorrektur nach Störungen
- Einsatzmöglichkeit besonders hoch integrierter Digitalbausteine wie FPGAs, Mikroprozessoren, Signalprozessoren, Speicher usw.
- Möglichkeit zur Datenkomprimierung und Verschlüsselung von Daten

12.1 Grundprinzip von Analog-Digital-Umsetzern

Analog-Digital-Umsetzer sind Systeme, die einer analog vorliegenden elektrischen Messgröße (z. B. einer Spannung U) eine digitale Repräsentationsgröße (z. B. eine binäre Zahl) zuordnen. Bei analogen Systemen liegt demgegenüber die Repräsentationsgröße,

W. Gehrke et al., *Digitaltechnik,* Springer-Lehrbuch,
DOI 10.1007/978-3-662-49731-9_12

beispielsweise der Zeigerausschlag eines Messgerätes, in analoger Form vor. Analoge Größen sind zeit- und wertkontinuierlich wie Abb. 12.1 zeigt.

Ein ADU ordnet der analogen Eingangsgröße eine zeit- und wertdiskrete Repräsentationsgröße zu, beispielsweise Binärzahlen, wie Abb. 12.2 zeigt. Ein ADU bildet demzufolge ein Signalintervall *(Quantisierungsintervall Q)* auf einen diskreten Wert ab. Dadurch werden systematische Fehler, die Quantisierungsfehler, verursacht.

Beim Vorliegen zeit- und wertkontinuierlicher, also analoger Signale bewirkt der ADU eine *Diskretisierung* in zweifacher Hinsicht:

- Diskretisierung in eine endliche Anzahl zugelassener Amplitudenwerte, auch *Quantisierung* genannt.
- Diskretisierung im Zeitbereich, denn ein Amplitudenwert gilt für eine bestimmte Mindestzeit. Diesen Vorgang nennt man *Abtastung*.

Weiterhin liefert der ADU die digitale Information in einem bestimmten Code, beispielsweise dem Dual-Code. Dieser Vorgang heißt *Codierung*. Die erforderlichen Verarbeitungsschritte beim Übergang vom analogen zum digitalen Signal sind in Abb. 12.3 veranschaulicht.

Wesentliche Anwendungsgebiete für ADUs und DAUs sind:

- Digitalmessinstrumente: Analoge Messgrößen wie Strom, Spannung, Widerstand, Frequenz, Temperatur, Gewicht usw. werden mit endlicher Auflösung als Ziffern angezeigt.

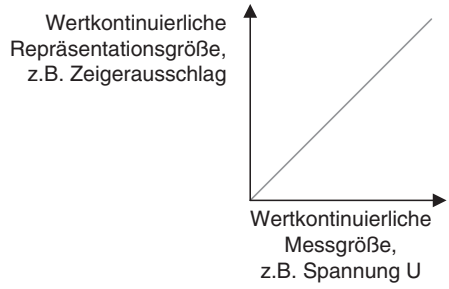


Abb. 12.1 Prinzipielle Wirkungsweise eines Analog-Messgerätes

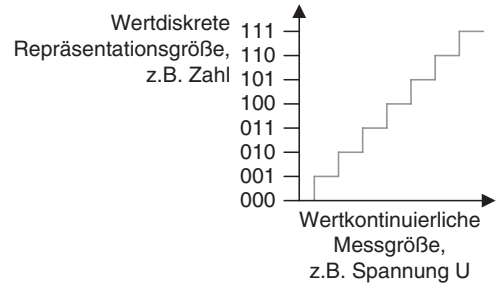


Abb. 12.2 Prinzipielle Wirkungsweise eines Analog-Digital-Umsetzers

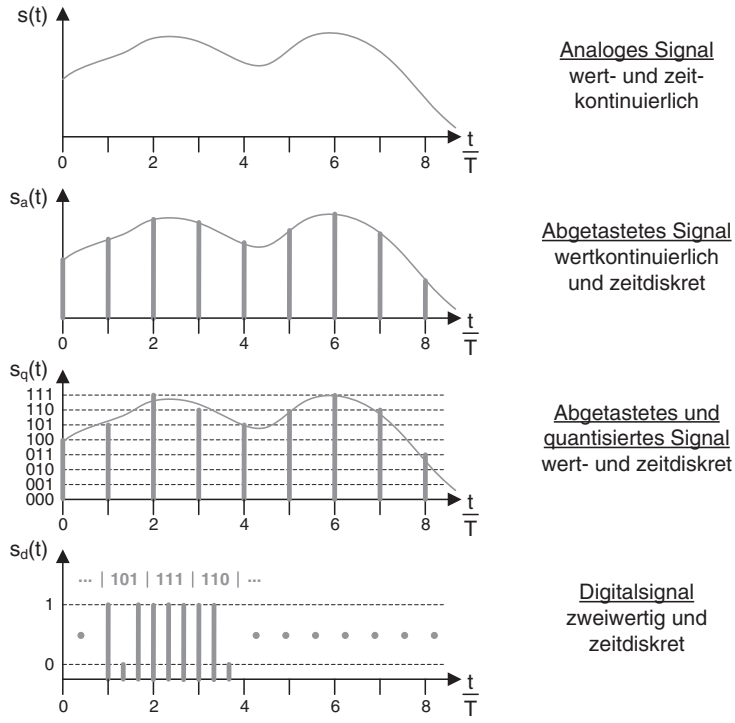


Abb. 12.3 Verarbeitungsschritte beim Übergang von analogen zu digitalen Signalen

- Nachrichtentechnische Einrichtungen: Sprach- und Videosignale, die zunächst in analoger Form vorliegen, werden digitalisiert und digital übertragen oder gespeichert. Beispiele: Telefonie per Voice-over-IP, Videocodierung für Digitalfernsehen oder Blu-Ray Disc.
- Digitale Signalverarbeitung: Sprach-, Bild- und Videosignale werden durch digitale Verarbeitung verändert. Beispiel: Bildverbesserung in Digitalkameras.
- Digitale Regelungssysteme und Prozesssteuerung: Ein Digitalregler kann einen oder mehrere Regelkreise betreiben. Beispiele: Werkzeugmaschinen, Lebensmittelproduktion, allgemeine Prozessabläufe, Überwachung von Verbundsystemen zur elektrischen Energieversorgung.

12.1.1 Systeme zur Umsetzung analoger in digitale Signale

Die Analog-Digital-Umsetzung umfasst prinzipiell die folgenden vier Schritte:

1. Bandbegrenzung durch Tiefpassfilter
2. Abtastung im *Abtasthalteglied (AHG,* engl. *Sample & Hold)*
3. Quantisierung
4. Codierung

Ein System zur Digitalisierung analoger Signale lässt sich somit durch das Blockschaltbild in Abb. 12.4 beschreiben. Integrierte Schaltungen zur AD-Umsetzung vereinen die Funktionen meist auf einem Baustein. Es ist jedoch auch möglich, die Umsetzfunktion auf mehrere Bauelemente aufzuteilen.

Der Eingangstiefpass mit der Grenzfrequenz f_g ist für den Fall erforderlich, dass das Analogsignal nicht hinreichend bandbegrenzt ist. Seine Dimensionierung wird durch das Abtasttheorem bestimmt (Abschn. 12.1.2). Als nächster Block ist ein Abtasthalteglied (AHG) vorgesehen. Dieses hält während der Umsetzungsdauer des ADU das umzusetzende Analogsignal konstant (Abschn. 12.1.3). Das AHG speist direkt den eigentlichen Umsetzer, der aus Quantisierer und Codierer besteht. Verschiedene Architekturen werden in Abschn. 12.2 vorgestellt. Eine Ablaufsteuerung koordiniert die Aufgaben der einzelnen Blöcke.

12.1.2 Abtasttheorem

Das *Abtasttheorem* von Shannon gibt an, in welchen zeitlichen Abständen dem vorliegenden Analogsignal mindestens Proben (Abtastwerte) entnommen werden müssen, damit nach einer späteren DA-Umsetzung das Ursprungssignal (bis auf die Quantisierungsfehler) fehlerfrei rekonstruiert werden kann.

> Das Abtasttheorem lautet: Eine auf f_g bandbegrenzte Signalfunktion $s(t)$ wird vollständig bestimmt durch zeitdiskrete und äquidistante Abtastwerte $s_a(t)$ im zeitlichen Abstand von $T = T_{\text{abt}} < 1/(2f_g)$

Das bedeutet, die in einem Signalgemisch auftretende höchstfrequente spektrale Komponente muss wenigstens zweimal pro Periode T_g abgetastet werden. Dieses lässt sich sowohl im Zeit- als auch im Spektralbereich begründen. Die Formel für T_{abt} wird auch mit dem Formelzeichen kleiner-gleich angegeben, aber das Gleichheitszeichen hat nur theoretische und keine praktische Bedeutung. Wird das Abtasttheorem verletzt, entstehen Signalfehler, die in der Regel nicht zu eliminieren sind.

Als Beispiel soll ein Audiosignal betrachtet werden. Das menschliche Gehör kann Frequenzen bis zu 20 kHz wahrnehmen. Die Abtastrate muss darum größer als 40 kHz sein. Für die Speicherung auf einer Audio-CD wird eine Abtastrate von 44,1 kHz verwendet.

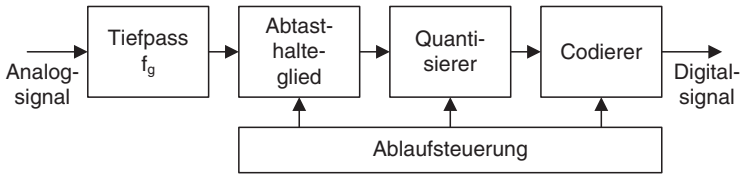


Abb. 12.4 Gesamtsystem zur Digitalisierung analoger Signale

12.1.3 Abtasthalteglied (AHG)

Das Abtasthalteglied soll dem vorliegenden Signal in Abständen, die durch das Abtasttheorem festgelegt sind, Signalproben entnehmen und diese während der Umsetzdauer t_u des ADUs konstant halten (speichern), wie Abb. 12.5 zeigt. Die Haltedauer t_H muss größer als die Umsetzdauer t_u des ADUs gewählt werden, so dass gilt:

$$t_u \leq t_H \leq T = T_{\text{abt}}$$

Ist allerdings die Umsetzdauer t_u sehr viel kleiner als T_{abt} kann theoretisch auf eine Abtasthaltung verzichtet werden. Diese Forderung lässt sich in der Praxis allerdings selten sinnvoll erfüllen, da der ADU dadurch sehr teuer werden würde. Die Zusammenhänge sollen an einem Beispiel konkretisiert werden.

Wird kein AHG benutzt, kann sich während der Umsetzdauer t_u des ADU das Eingangssignal $s(t)$ um ds ändern, was zu einem falschen Umsetzungsergebnis führt. Soll die maximale Genauigkeit eines ADU von 1/2 LSB (Least Significant Bit) erhalten bleiben, muss im Sinne einer Worst-Case-Betrachtung gefordert werden, dass an der Stelle der größtmöglichen Signalsteigung die Signaländerung kleiner als 1/2 LSB bleibt. Beispielsweise führt diese Forderung bei einem vollaussteuernden Sinussignal $s(t) = A \cdot sin(\omega_g \cdot t)$ zu folgendem Ergebnis:

$$\left(\frac{ds}{dt}\right)_{\text{max}} = A \cdot \omega_g = S_{\text{max}}$$

Das ist die maximale Steigung des Signals mit der Amplitude $A = m \cdot Q/2$, wobei Q die Quantisierungsintervallbreite und m die Quantisierungsstufenzahl im Aussteuerbereich sind. Weiter gelte $f_g = 1/(2 \cdot T_{\text{abt}})$ als Grenzfall für die Abtastung. Dann folgt:

$$S_{\text{max}} = \frac{m \cdot Q \cdot 2 \cdot \pi \cdot f_g}{2} = \frac{m \cdot Q \cdot \pi}{2 \cdot T_{\text{abt}}}$$

Mit der oben formulierten Bedingung $S_{\text{max}} \cdot t_u \leq Q/2$ folgt durch Gleichsetzen

$$S_{\text{max}} = \frac{m \cdot Q \cdot \pi}{2 \cdot T_{\text{abt}}} \leq \frac{Q}{2 \cdot t_u}$$

und nach t_u umgestellt:

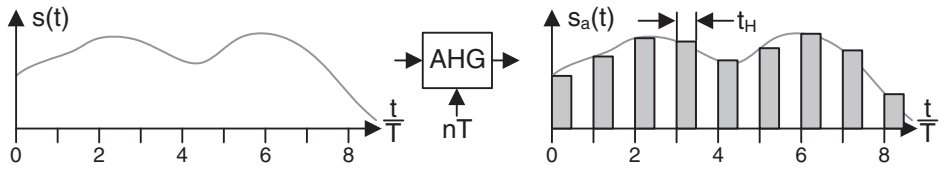


Abb. 12.5 Prinzipielle Wirkungsweise eines Abtasthaltegliedes

$$t_u \leq \frac{T_{\text{abt}}}{m \cdot \pi}$$

Dieses ist die erforderliche Umsetzdauer eines ADUs, welche bei der Abtastung ein AHG entbehrlich macht. Diese Forderung ist sehr weitreichend, da in der Regel m sehr viel größer als 1 ist.

Als Beispiel wird eine Abtastperiodendauer von 125 µs für die digitale Sprachsignalverarbeitung mit Telefonqualität betrachtet. Es wird ein linearer ADU mit $n = 12$ bit verwendet, sodass $m = 2^n - 1 = 4095$ ist. Dann gilt für die Umsetzdauer:

$$t_u \leq \frac{125\mu s}{4095 \cdot \pi} = 9,72ns$$

Dieses lässt sich nicht sinnvoll realisieren, da ADUs mit diesen Leistungsmerkmalen zwar technisch möglich, jedoch für diese Anwendung zu aufwendig sind. Wird dagegen ein AHG eingesetzt, darf die Umsetzdauer t_u des ADU näherungsweise T_{abt}, also im vorliegenden Beispiel 125 µs betragen. Darin liegen Sinn und Vorteil eines AHG.

Die Arbeitsweise eines AHG zeigt das prinzipielle Schaltbild in Abb. 12.6. Wird der Schalter S in die obere Stellung gebracht, lädt sich der Haltekondensator C_H auf die Signalspannung auf. Dieses entspricht der Abtastphase. Nach Bewegen des Schalters S in die Mittelstellung beginnt die Haltephase, während der das Signal in C_H gespeichert bleibt. Die Spannung U_H ist durch einen hochohmigen Leseverstärker als $s_a(t)$ verfügbar. R_S ist der Eingangswiderstand des AHG, R_i der Innenwiderstand der Signalquelle.

In modernen Pipeline-ADUs in CMOS-Technologie werden die AHGs mit geschalteten Kondensatoren (Switched Capacity Circuits) realisiert, wie in Abb. 12.7 dargestellt ist.

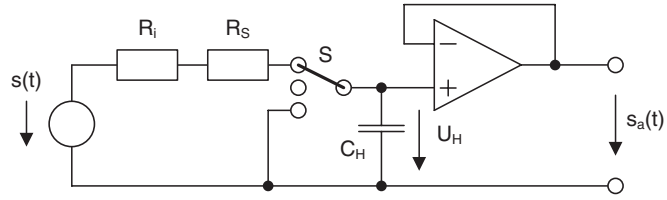


Abb. 12.6 Prinzipieller Aufbau eines AHGs und sein Anschluss an die Signalquelle s(t)

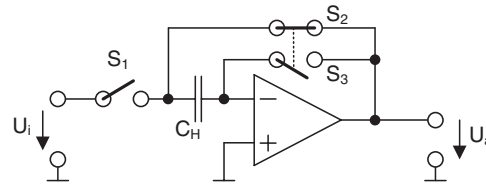


Abb. 12.7 Prinzipschaltbild eines Abtasthalteglieds mit geschaltetem Kondensator in CMOS-Technik, wie es z. B. in Pipeline-ADUs verwendet wird

In der Abtastphase sind die Schalter S_1 und S_3 geschlossen, und S_2 ist offen. Da der Summationspunkt des Operationsverstärkers auf Bezugspotenzial liegt, wird der Kondensator C_H mit der Eingangsspannung U_i geladen.

Für die Haltephase werden die Schalter S_1 und S_3 geöffnet und S_2 geschlossen und damit die Haltekapazität in den Gegenkopplungskreis des Operationsverstärkers gelegt. Da die Ladung von C_H nicht über den Summationspunkt abfließen kann, bleibt sie erhalten, und die Ausgangsspannung U_a nimmt den Wert U_i an.

12.1.4 Erreichbare Genauigkeit für ADUs abhängig von der Codewortlänge

Durch die Codewortlänge n ist die Anzahl der möglichen Codewörter gegeben. Hieraus lassen sich die Quantisierung der Messwerte und die erreichbare Genauigkeit berechnen. Zur besseren Anschaulichkeit wird im Folgenden angenommen, dass der Messbereich bei 0 V beginnt.

Ein Wert für die mögliche Genauigkeit eines ADUs ist die Quantisierungsintervallbreite Q. Sie berechnet sich aus der Codewortlänge n und dem Aussteuerbereich $U_{\max}$. Für einen n-Bit-ADU ergibt sich Q als:

$$Q = \frac{U_{\max}}{2^n}$$

Der höchste codierbare Spannungswert beträgt dann:

$$U^*_{\max} = \left(2^n - 1\right) \cdot Q$$

Dieser Wert ergibt sich daraus, dass von den 2^n möglichen Codewörtern der erste Wert für die Spannung 0 V benötigt wird und die folgenden $2^n - 1$ Codewörter jeweils um den Wert Q größer sind.

Als einfaches Beispiel betrachten wir einen Aussteuerbereich $U_{\max}$ von 1 V und eine Wortbreite von $n = 3$ bit. Dann beträgt $Q = 1\,\text{V}/2^3 = 1\,\text{V}/8 = 125\,\text{mV}$. Die Zahl der Quantisierungsintervalle beträgt 8, sodass der höchste codierbare Spannungswert 7 Intervalle höher als 0 V ist:

$$U^*_{\max} = 7 \cdot 125mV = 0,875V$$

Die Quantisierungskennlinie dieses ADUs ist in Abb. 12.8 dargestellt. Es existieren 8 darstellbare Spannungswerte und 7 Intervalle zwischen diesen Werten. Jedem Codewort entspricht ein Repräsentationswert, beispielsweise für den Code 011 der Wert 0,375 V. Die Eingangswerte des zugehörigen Quantisierungsintervalls werden diesem Wert zugewiesen. Für den Code 011 sind beispielsweise die Übergangswerte 0,3125 und 0,4375 V. Der höchste darstellbare Digitalwert ist 0,875 V und um ein Quantisierungsintervall kleiner als die maximale Eingangsspannung $U_{\max}$.

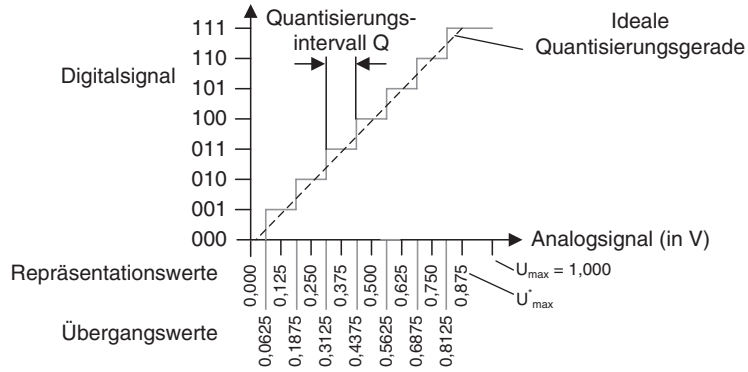


Abb. 12.8 Quantisierungskennlinie eines 3-Bit-ADUs mit $U_{max} = 1$ V

Die beiden Punkte in den Ecken des Diagramms legen die ideale Quantisierungsgerade fest. Diese verläuft durch die Mittelpunkte aller Quantisierungsintervalle einer idealen Quantisierungskennlinie. Bei einer realen Quantisierungskennlinie ergibt sich für die Mittelpunkte aller Quantisierungsintervalle jedoch im allgemeine keine Gerade. Darin äußern sich unterschiedliche Fehler realer Umsetzer, wie sie in Abschn. 12.4 im Einzelnen erläutert werden.

Als Beispiel für reale Größenordnungen wird ein Aussteuerbereich U_{max} von 10 V und eine Wortbreite von $n = 12$ bit betrachtet. Dann beträgt $Q = 10\,\text{V}/2^{12} = 1\,\text{V}/4096 = 2{,}44$ mV. Der höchste codierbare Spannungswert beträgt $U^*_{max} = 4095 \cdot (10\,\text{V}/4096) = 9{,}976$ V.

Bei einer idealen Quantisierungsgerade sind alle Quantisierungsintervalle Q gleich groß und man spricht von linearer Quantisierung. Dann ist der maximale Quantisierungsfehler F_{abs} die Hälfte des Quantisierungsintervalls:

$$F_{abs} = \frac{Q}{2} = \frac{U_{max}}{2^{n+1}}$$

Der relative Fehler F_{rel} hängt von der aktuellen Aussteuerung ab, er nimmt bei Vollaussteuerung sein Minimum an:

$$F_{rel} = \frac{F_{max}}{U^*_{max}} \approx \frac{F_{max}}{U_{max}} = \frac{1}{2^{n+1}}$$

Für einen 3-Bit-ADU beträgt der Fehler F_{rel} beispielsweise 1/16 = 6,25 %.

Wird bei einer Digitalisierung eine bestimmte relative Genauigkeit F_{rel} verlangt, so kann daraus die erforderliche Wortbreite für den ADU berechnet werden. Sie ergibt sich aus dem nächstgrößeren ganzzahligen Wert und dem Zweierlogarithmus der benötigten Genauigkeit nach der Formel:

$$n^* \geq n = -1 - ld\, F_{rel}$$

Tab. 12.1 Repräsentationswerte und Codeworte für einen 12-Bit-ADU

Codewort-Nummer	Repräsentationswert in V	Codierung
0	0	0000 0000 0000
1	0,0024414	0000 0000 0001
2	0,0048828	0000 0000 0010
…	…	…
1024	2,5000000	0100 0000 0000
…	…	…
2048	5,0000000	1000 0000 0000
…	…	…
4095	9,9975586	1111 1111 1111

Soll beispielsweise eine relative Genauigkeit bei Vollaussteuerung von 1 % erreicht werden, berechnet sich der Zweierlogarithmus von 0,01 zu $ld\,0{,}01 = -6{,}64$. Daraus ergibt sich $n = 5{,}64$, sodass für den ADU mindestens $n^* = 6$ bit nötig sind.

12.1.5 Codierung der ADU-Werte

Für das eben genannte Beispiel eines 12-Bit-ADU mit Aussteuerbereich U_{max} von 10 V ist die Codetabelle in Tab. 12.1 auszugsweise dargestellt. Die Codeworte werden als Dualzahl dargestellt.

Falls mit dem ADU auch negative Spannungen gemessen werden, ist eine Darstellung als Dualzahl mit Offset oder als Zweierkomplement möglich. Bei der Offsetdarstellung beginnt der Code bei der geringsten Spannung mit dem Codewort Null und steigt bis zum höchsten codierbaren Spannungswert an. Bei der Zweierkomplementdarstellung werden negative Spannungswerte durch eine negative Zahl angegeben. Manche Bausteine bieten auch die Datenausgabe im Gray-Code. Die Codierung eines ADUs ist im Datenblatt definiert und kann teilweise durch Konfigurationssignale ausgewählt werden.

12.2 Verfahren zur Analog-Digital-Umsetzung

Für die eigentliche AD-Umsetzung sind verschiedene Verfahren möglich, die sich in Aufwand und Geschwindigkeit deutlich unterscheiden. Für die folgenden Erläuterungen werden meistens die Repräsentationswerte der einzelnen Quantisierungsschritte verwendet, da dies anschaulicher ist (siehe Abb. 12.8). In realen Schaltungen erfolgen Vergleiche hingegen mit den Übergangswerten.

12.2.1 Parallelverfahren

Umsetzer nach diesem Verfahren heißen *Parallel-, Direkt-* oder *Flash-Umsetzer*. Das Messverfahren ähnelt der Längenmessung mit einem Zollstock. An die unbekannte Größe wird der Zollstock angelegt, der in m Teile des Quantisierungsintervalls, sogenannte Normale, eingeteilt ist. Der nächstliegende ganzzahlige Wert ist die gesuchte Länge.

Für die elektronische Realisierung dieses Verfahrens ist Folgendes wichtig:

- Es ist nur ein Messschritt nötig, das Verfahren arbeitet schnell.
- Es sind m Normale nötig, also großer Aufwand an Präzisionsbauelementen.

Elektrisch kann dieses Normalenlineal durch eine Spannungsteilerkette mit m gleichgroßen Präzisionswiderständen realisiert werden. Jeder Widerstand ergibt eine darstellbare Stufe. Zusätzlich ist noch der Wert Null vorhanden, so dass $m + 1$ Werte möglich sind. Das Blockschaltbild des entsprechenden Parallelumsetzers ist in Abb. 12.9 dargestellt. Es gilt: $R_1 = R_2 = \ldots = R_m$. R_{ref} wird entsprechend dem Verhältnis von U_{max} zu U_{ref} gewählt.

Mittels m Komparatoren wird die unbekannte Spannung U_x mit den einzelnen Abgriffen des Normalen-Spannungsteilers verglichen. Alle Komparatoren, deren Spannungen an den Teilereingängen größer als U_x sind, liefern am Ausgang eine logische 1, alle anderen eine logische 0. Diese Werte werden mit einem Abtastimpuls in ein Register übernommen und in der Decodierlogik in die entsprechende Anzahl von $n = ld(m + 1)$ bits umgesetzt. Das Register realisiert eine digitale Abtasthaltung, sodass dieser Umsetzer ohne ein zusätzliches AHG betrieben werden kann.

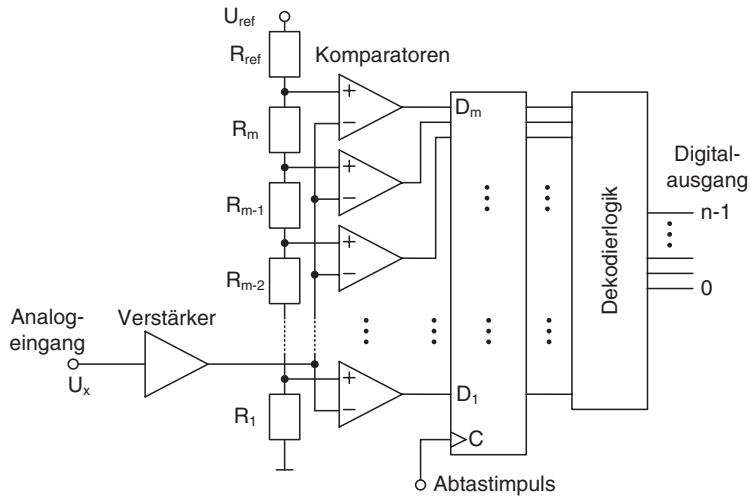


Abb. 12.9 Blockschaltbild eines ADUs nach dem Flash-Verfahren

Der hohe Aufwand zeigt sich in der großen erforderlichen Anzahl von Präzisionswiderständen und Komparatoren. Daher wird dieses Verfahren normalerweise nur bis Auflösungen von 12 bit eingesetzt. Technische Probleme bei hoher Auflösung liegen außerdem im Eingangsverstärker, der m Komparatoreingänge treiben muss und in den Komparatoren selbst, die eine kleine Hysterese und hohe Gleichtaktunterdrückungen aufweisen müssen. Ein weiterer Nachteil ist die vergleichsweise hohe Verlustleistung dieses Umsetzertyps.

Die Geschwindigkeit des Umsetzers wird durch den langsamsten Komparator bestimmt, der erst eingeschwungen sein muss, bevor der Abtastimpuls eintrifft. Anwendungsschwerpunkte für Umsetzer dieses Typs liegen bei der digitalen Signalverarbeitung, insbesondere der Bildverarbeitung mit Datenraten von mehr als 80 Mbit/s und bei Transientenrekordern.

Eventuell ist Ihnen aufgefallen, dass beim Flash-ADU die Quantisierungsschritte immer bis zum nächsten Repräsentationswert reichen. Bei der Auflösung aus Abb. 12.8 würde die Codierung „000“ also dem Wertebereich von 0 V bis 0,125 V entsprechen. Dies liegt daran, dass alle Widerstände des Spannungsteilers gleich groß gewählt sind und die Komparatoren die Eingangsspannung mit den Repräsentationswerten vergleichen. Bezogen auf die Dezimalzahl wird durch diese Vereinfachung der Nachkommaanteil abgeschnitten und es findet nicht die eigentlich erforderliche Rundung statt.

In der Praxis wird diese Verschiebung durch Halbierung des Werts für R_1 und entsprechender Erhöhung von R_{ref} oder durch einen Offset im Eingangsverstärker gelöst. Um die Beschreibung der Schaltungsstrukturen in diesem Kapitel übersichtlich zu halten, sollen solche Rundungsfragen nicht beachtet werden.

12.2.2 Wägeverfahren

Beim *Wägeverfahren* wird pro Messschritt ein Bit des Digitalwortes erzeugt. Der Name dieses Verfahrens stammt von dem bei einer Balkenwaage üblichen Messvorgang: Das Wägegut unbekannten Gewichts wird in eine Waagschale gelegt. In die andere kommt zunächst das größte verfügbare Gewicht. Ist dieses zu schwer, wird es wieder entfernt und eine Null notiert. Ist es nicht zu schwer, bleibt es liegen und es wird eine Eins notiert. Anschließend werden nacheinander alle verfügbaren kleineren Gewichte in gleicher Weise benutzt. Das unbekannte Gewicht entspricht der Summe aller mit Eins markierten Gewichte.

Das Wägeverfahren benutzt mehrere Normale q_i mit dualer Abstufung ihres Wertes. Die Auflösung entspricht einer Quantisierungsstufe Q, also dem LSB des fertigen Codewortes. Die erste Normale q_0 hat den Wert Q, die folgenden Normale sind $q_1 = 2 \cdot Q$, $q_1 = 4 \cdot Q$ bis $q_{n-1} = 2^{n-1} \cdot Q$.

Da die Anwendung jedes Normals q_i ein Bit liefert, sind für einen n-Bit-ADU also n Normale nötig. Das größte umfasst den halben Messbereich, also $U_{max}/2$ und die Summe aller Normale ergibt den insgesamt darstellbaren Messbereich $U_{max} - Q$.

Eine Messung der Eingangsgröße x beginnt mit dem Vergleich mit dem größten Normal q_{n-1}.

- Gilt $x \geq q_{n-1}$, wird $b_{n-1} = 1$ und q_{n-1} bleibt angelegt.
- Gilt dagegen $x < q_{n-1}$, wird $b_{n-1} = 0$ und q_{n-1} wird entfernt.

Damit ist das MSB (Most Significant Bit) des Digitalwertes b gebildet. Im nächsten Schritt wird der verbleibende Rest der Messgröße mit dem nächstkleineren Normal verglichen.

- Gilt $x - b_{n-1} \cdot q_{n-1} \geq q_{n-2}$, wird $b_{n-2} = 1$ und q_{n-2} bleibt angelegt.
- Gilt dagegen $x - b_{n-1} \cdot q_{n-1} < q_{n-2}$, wird $b_{n-2} = 0$ und q_{n-2} wird entfernt.

Dieser Vorgang wird mit den restlichen Normalen bis zum kleinsten Normal q_0 der Größe Q fortgesetzt. Die Zahl Z der Messschritte entspricht der Normalenzahl N und damit der Bitanzahl, also gilt $Z = N = n$.

Das Messergebnis lautet $x = b_{n-1} \cdot 2^{n-1} + b_{n-2} \cdot 2^{n-2} + \cdots + b_2 \cdot 2^2 + b_1 \cdot 2 + b_0$.

Als Beispiel wird ein Wägecodierer mit $m = 255$ Quantisierungsintervallen betrachtet. Er hat eine Auflösung von $Q = 1/256$ und damit sind $Z = N = 8$ und $n = ld(m + 1) = 8$ bit. Daher sind $N = 8$ Normale und $Z = 8$ Messschritte erforderlich. Das größte Normal hat den Wert $q_{n-1} = 2^{(n-1)} \cdot Q = 128 \cdot Q$ und das kleinste den Wert $q_0 = Q$.

Die Umsetzzeit im Wägecodierer ist größer als beim Direktumsetzer, da mehr Schritte erforderlich sind. Dafür werden weniger Normale benötigt, d. h. der Aufwand an Präzisionsbauteilen ist prinzipiell geringer.

Bei der technischen Realisierung des Wägeverfahrens unterscheidet man zwischen Umsetzern mit *schrittweiser Annäherung (Sukzessive Approximation, Successive Approximation)* und *Kaskadenumsetzer (Pipeline-A/D-Umsetzer).* In der Praxis wird von diesen beiden Varianten meist das Verfahren mit schrittweiser Annäherung verwendet und darum wird dieses hier anhand des Rückkopplungscodierers erläutert. Das Blockschaltbild in Abb. 12.10 zeigt den Aufbau mit der Rückkopplung als wesentliches Merkmal.

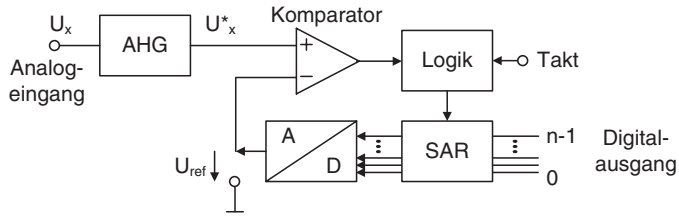


Abb. 12.10 Blockschaltbild des Rückkopplungscodierers mit schrittweiser Annäherung (Sukzessive Approximation)

Als Rückkopplung wird mit einem Digital-Analog-Umsetzer (DAU) eine Referenzspannung U_{ref} erzeugt. Diese wird entsprechend der Summe der aktivierten Normale schrittweise variiert. Im ersten Schritt ist das MSB gesetzt und $U_{\text{ref}} = U_{\text{max}}/2$. Der Komparator vergleicht die Referenzspannung mit der Eingangsspannung und ermittelt das MSB des Digitalausgangs, der im Successive Approximation Register (SAR) gespeichert wird. Der DAU erzeugt dann den nächsten analogen Vergleichswert, der wieder mit dem Eingangssignal verglichen wird um das nächste Ausgangsbit zu ermitteln.

Als Beispiel soll das Verfahren für einen ADU mit $n = 3$ bit und $U_{\text{max}} = 1\,\text{V}$ durchgespielt werden. Das Quantisierungsintervall Q ist damit 0,125 V. Als Eingangsspannung wird $U_x = 0{,}8\,\text{V}$ angenommen.

- Im Abtasthalteglied AHG wird die Eingangsspannung gehalten, damit während der schrittweisen Umsetzung stets der gleiche Analogwert U^*_x anliegt.
- Der erste Vergleich erfolgt mit dem Wert $U_{\text{ref}} = 2^2 \cdot Q = 4 \cdot 0{,}125\,\text{V} = 0{,}5\,\text{V}$. Der Eingangswert von 0,8 V ist größer als die Referenzspannung, also ist das MSB b_2 gleich 1.
- Der zweite Vergleich addiert zu der bisher ermittelten Spannung von 0,5 V die nächste Normale, mit halber Größe der vorherigen Normale. Es ergibt sich also der Wert $U_{\text{ref}} = b_2 \cdot 2^2 \cdot Q + 2^1 \cdot Q = 0{,}5\,\text{V} + 2 \cdot 0{,}125\,\text{V} = 0{,}75\,\text{V}$. Der Eingangswert von 0,8 V ist wieder größer als die Referenzspannung, also ist das nächste Bit b_1 gleich 1.
- Der dritte Vergleich addiert wieder zu der bisher ermittelten Spannung von 0,75 V die nächste Normale, mit halber Größe der vorherigen Normale. Es ergibt sich der Wert $U_{\text{ref}} = b_2 \cdot 2^2 \cdot Q + b_1 \cdot 2^1 \cdot Q + 2^0 \cdot Q = 0{,}75\,\text{V} + 0{,}125\,\text{V} = 0{,}875\,\text{V}$. Der Eingangswert von 0,8 V ist kleiner als die Referenzspannung, also ist Bit b_0 gleich 0.
- Damit ergibt sich nach drei Schritten der digitale Ausgangswert 110, der einer Spannung von $6 \cdot Q = 0{,}75\,\text{V}$ entspricht.

Das Ergebnis ist innerhalb der erreichbaren Genauigkeit für die gewählte Codewortlänge.

12.2.3 Zählverfahren

Beim *Zählverfahren* handelt es sich um ein rein seriell arbeitendes Verfahren. Es existiert nur ein Normal der Länge Q und während der Messung wird gezählt, wie oft dieses Normal an den unbekannten Wert x angelegt werden muss, um diesen zu erreichen. Das Zählergebnis entspricht dann dem gesuchten Digitalwert von x.

Die Zahl der erforderlichen Vergleichsschritte Z hängt von der Messgröße ab und beträgt maximal $Z = m = 2^n - 1$, denn falls beim $(2^n - 1)$ten Messschritt immer noch gilt $x > (2^n - 1) \cdot Q$, dann muss x im letzten Quantisierungsintervall liegen.

Der Vorteil dieses Umsetzertyps ist, dass nur ein Normal, also ein geringer Aufwand an Präzisionsbauelementen, benötigt wird. Da die Anzahl der Messschritte jedoch von allen Umsetzverfahren am größten ist, arbeitet es auch am langsamsten.

Das Zählverfahren kann elektronisch durch eine Abwandlung des soeben vorgestellten Rückkopplungsumsetzers realisiert werden. Dazu wird das SAR durch einen Zähler ersetzt und damit U_{ref} pro Messschritt nur um eine Quantisierungsstufe Q erhöht.

Vergleicht man die drei bisher dargestellten Umsetzverfahren miteinander, so zeigt sich, dass Aufwand (bzw. Kosten) und Umsetzungsdauer bis zu einem gewissen Grade untereinander austauschbar sind. Dieses ist in Abb. 12.11 anschaulich dargestellt. Häufig besteht bei der Anwendung von ADUs jedoch der Wunsch, die Auswahl hinsichtlich Geschwindigkeit und Kosten präziser an das vorliegende Digitalisierungsproblem anzupassen, als es die drei bisher genannten Verfahren zulassen. Dafür stehen zwei weitere Verfahren zur Verfügung: Das erweiterte Parallel- und das erweiterte Zählverfahren. Beide werden in den nächsten Abschnitten vorgestellt.

12.2.4 Erweitertes Parallelverfahren

Das Parallelverfahren ist zwar sehr schnell, hat aber den Nachteil, dass der Aufwand an Präzisionsbauteilen exponentiell mit der Auflösung steigt; denn es werden $N = m = 2^n - 1$ Normale für einen n-Bit-Umsetzer benötigt. Abhilfe schafft hier das erweiterte Parallelverfahren, dessen Funktion zwischen Parallel- und Wägeverfahren liegt.

Im folgenden Abschnitt wird zunächst das allgemeine Prinzip des *erweiterten Parallelverfahrens* dargelegt und anschließend eine moderne Realisierung dieses Prinzips anhand des *Pipeline-A/D-Umsetzers* erläutert.

12.2.4.1 Allgemeines Prinzip des erweiterten Parallelverfahrens

Man erhöht, ausgehend von einem Parallelverfahren, die Anzahl der Messschritte von $Z = 1$ auf $Z > 1$, beispielsweise auf $Z = 2$, bildet im ersten Schritt $(m' + 1)$ Grobstufen und unterteilt die Grobstufe, in der der unbekannte Wert x liegt, in $(m'' + 1)$ Feinstufen. Die Gesamtauflösung beträgt dann $m + 1 = (m' + 1) \cdot (m'' + 1)$, und die Zahl der Normale verringert sich auf $N = m' + m''$.

Das soll am Beispiel verdeutlicht werden. Für einen 8-Bit-ADU gilt $m = 2^8 - 1 = 255$. Dann muss z. B. für $Z = 2$ Messschritte gelten: $m + 1 = (m' + 1) \cdot (m'' + 1) = 256$. Hierfür gibt es die in Tab. 12.2 dargestellten Möglichkeiten.

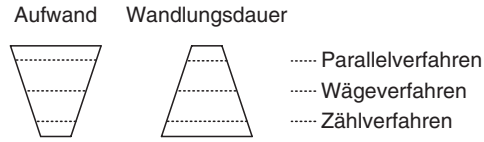


Abb. 12.11 Vergleich der drei klassischen AD-Umsetzverfahren hinsichtlich Hardwareaufwand und Geschwindigkeit

Tab. 12.2 Möglichkeiten für die Realisierung eines ADUs nach dem erweiterten Parallelverfahren, mit n = 8 bit und Z = 2 Schritten

Grobstufen	Feinstufen	N=m′+m″	Bemerkungen
1	256	255	Direktverfahren
2	128	128	
4	64	66	
8	32	38	
16	16	30	Minimale Normalenzahl
32	8	38	Ab hier Wiederholung

Allgemein gilt, dass die minimale Normalenzahl, also der kleinste Hardwareaufwand, im Fall $(m' + 1) = (m'' + 1)$ erreicht wird.

Geht man allgemein auf $Z > 2$ Messschritte über, muss gelten $(m' + 1) \cdot (m'' + 1) \cdot (m''' + 1) \cdots \cdot (m^{(Z)} + 1) = m + 1 = 2^n$ und die Normalenzahl beträgt:

$$N = \sum_{i=1}^{Z} m^{(i)}$$

Die Zahl der nötigen Normale wird wiederum minimal, wenn für alle $m^{(i)} = m' = konstant$ gilt. Dann beträgt die Anzahl an Quantisierungsstufen pro Messschritt:

$$\left(m' + 1\right) = \left(m'' + 1\right) = \left(m^{(Z)} + 1\right) = \sqrt[Z]{m + 1} = \sqrt[Z]{2^n}$$

und die erforderliche Normalenzahl beträgt $N = Z \cdot m'$.

Als einfaches Beispiel soll ein 8-Bit-ADU in $Z = 4$ Schritten mit minimaler Normalenzahl aufgeteilt werden. Die minimale Normalenzahl ergibt sich für

$$\left(m' + 1\right) = \left(m'' + 1\right) = \left(m''' + 1\right) = \left(m'''' + 1\right) = \sqrt[4]{2^8} = \sqrt[4]{256} = 4$$

Dies bedeutet pro Umsetzerstufe werden 2 Bit generiert. Die Zahl der Normale beträgt $N = Z \cdot m' = 4 \cdot 3 = 12$.

Falls die Einzelquantisierungsstufenzahl keine Potenz von 2 ergibt, ist eine andere Aufteilung nötig. Soll beispielsweise der 8-Bit-ADU in $Z = 3$ Schritte aufgeteilt werden, ergibt sich für die Anzahl an Quantisierungsstufen der Wert $\sqrt[3]{256} = 6{,}35$ also keine Zweierpotenz. Die drei Stufen müssen dann so gewählt werden, dass sie einer Zweierpotenz entsprechen und sich insgesamt die benötigten 256 Quantisierungsstufen ergeben. Dies erfolgt, durch zwei Stufen mit 8 und einer Stufe mit 4 Einzelquantisierungsstufen, die insgesamt $8 \cdot 8 \cdot 4 = 256$ Stufen ergeben. In Bit gerechnet ergeben die Einzelstufen zweimal 3 und einmal 2 Bit, insgesamt also 8 Bit. Die minimale Normalenzahl ist 17. Die Umsetzerstruktur ist in Abb. 12.12 gezeigt.

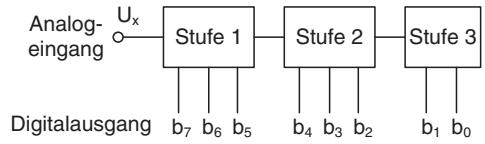


Abb. 12.12 Struktur eines dreischrittigen 8-Bit-ADU nach dem erweiterten Parallelverfahren mit minimaler Normalenzahl

Umsetzer nach dem erweiterten Direktverfahren mit der Schrittzahl $Z = 2$ sind als Half-Flash-Umsetzer auf dem Markt vertreten. Das vereinfachte Blockschaltbild des Half-Flash-Umsetzers AD 7821 (Analog Devices) mit 8 Bit ist in Abb. 12.13 dargestellt. Ein 4-Bit-Direktumsetzer erzeugt im ersten Schritt die vier höchstwertigen Bits (MSB). Deren Analogäquivalent wird anschließend von der analogen Eingangsspannung subtrahiert. Aus der verbleibenden Differenz werden dann mit einem zweiten 4-Bit-Direktumsetzer die vier niederwertigsten Bits ermittelt (LSB).

12.2.4.2 Pipeline-Analog-Digital-Umsetzer

Bei einem Pipeline-ADU erfolgt die Umsetzung ebenfalls in mehreren Schritten. Anders als beim allgemeinen Verfahren werden die Werte in jeder Stufe mit einem AHG gehalten und nach der Differenzbildung verstärkt. Durch das Halten der Zwischenwerte ist eine Verarbeitung im Pipeline-Verfahren möglich, denn während die zweite Stufe die nachfolgenden Bits ermittelt, kann die erste Stufe bereits den nächsten Abtastwert bearbeiten. Die Verstärkung ermöglicht der nachfolgenden Stufe mit höheren Signalpegeln zu arbeiten.

Abb. 12.14 zeigt das Prinzip eines Pipeline-Analog-Digital-Umsetzers mit vier Stufen eines 3-Bit-Umsetzers, angelehnt an den Baustein AD9200 von Analog Devices. Vom analogen Eingangssignal werden in der ersten Stufe die drei höchstwertigen Bits in einem Flash-AD-Umsetzer (ADU) ermittelt und das digitale Teilergebnis mit einem DA-Umsetzer (DAU) wieder in einen analogen Wert umgesetzt. Der Eingangswert wird im ersten Abtast-Halte-Glied (AHG) gespeichert und von ihm wird jetzt der Ausgang des DAUs abgezogen. Da im ersten Schritt drei Bit des Ergebnisses ermittelt wurden, kann die Differenz um den Faktor $2^3 = 8$ verstärkt werden. Dadurch hat die Differenz wieder

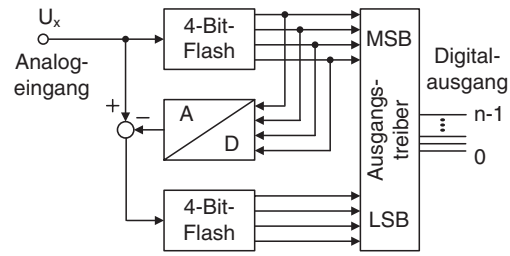


Abb. 12.13 Vereinfachtes Blockschaltbild eines 8-Bit-Half-Flash-Umsetzers

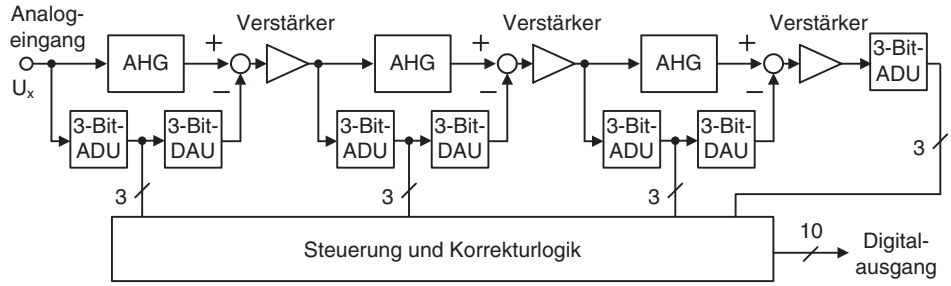


Abb. 12.14 Blockschaltbild eines Pipeline-Analog-Digital-Umsetzers

den gleichen Pegel wie das Eingangssignal und die zweite Stufe kann genauso wie die erste Stufe dimensioniert werden. Dies ist für den Schaltungsentwurf und die Genauigkeit der AD- und DA-Umsetzung vorteilhaft. Nach der zweiten Stufe werden in der dritten und vierten Stufe die weiteren Bits ermittelt.

Für den praktischen Entwurf ist es vorteilhaft, wenn sich die Bereiche der einzelnen Stufen etwas überlappen. Pro Stufe wird darum nicht der volle 3-Bit-Messbereich von 0 bis 7 genutzt, sondern nur etwa der Wertebereich von 0 bis 5. Diese 6 Werte entsprechen rund 2,5 bit Auflösung und somit erfolgt dann die Verstärkung zwischen den Stufen auch mit dem Faktor 6. Eine Korrekturlogik setzt aus den vier Teilergebnissen den Messwert mit der Genauigkeit von 10 bit zusammen. In dieser Korrekturlogik kann sichergestellt werden, dass sich der gesamte ADU über den Messbereich möglichst linear verhält. Insbesondere wird vermieden, dass *Missing Codes* auftreten, das heißt beim Übergang zwischen Messbereichen wird kein Codewort übergangen.

12.2.5 Erweitertes Zählverfahren

Das erweiterte Zählverfahren liegt in seiner Funktion zwischen dem Zähl- und dem Wägeverfahren. Das Zählverfahren hat zwar den Vorteil minimalen Aufwands an Präzisionsbauelementen, dafür ist aber die Schrittzahl und damit die Umsetzdauer die höchste der drei klassischen Umsetzverfahren. Beispielsweise sind für einen 8-Bit-Umsetzer 255 Schritte erforderlich.

Eine Reduzierung der Umsetzdauer lässt sich prinzipiell durch eine Aufteilung in eine Grobmessung und eine Feinmessung erreichen. In der Grobmessung könnte ein Normal der Größe $2 \cdot Q$ verwendet werden, was bei 8 Bit 127 Schritte erfordert. In der Feinmessung wird dann in einem einzelnen Schritt ein Normal der Größe Q verwendet. Somit wird die Anzahl der erforderlichen Schritte etwa halbiert. Auch eine weitere Aufteilung mit Zwischenmessungen ist denkbar.

Eine praktische Bedeutung bei der Realisierung von ADUs hat das erweiterte Zählverfahren bislang nicht.

12.2.6 Single- und Dual-Slope-Verfahren

Bisher wurden ausschließlich Umsetzverfahren betrachtet, bei denen die elektrische Spannung direkt gemessen wurde. Bei indirekten Verfahren wird dagegen die Messgröße zunächst in eine Hilfsgröße überführt, welche genauer, schneller oder mit kleinerem Aufwand messbar ist. Die wichtigsten Hilfsgrößen sind eine messgrößenproportionale Frequenz sowie eine messgrößenproportionale Zeit.

Die Messung mithilfe einer variablen Zeit erfolgt durch Zählung mit einem Taktsignal. Das Grundprinzip dieses Verfahren wird als *Single-Slope-Verfahren* bezeichnet. Die Funktion wird an einem Blockschaltbild beschrieben (Abb. 12.15). Ein Sägezahngenerator erzeugt eine linear ansteigende Spannung U_k, die zum Beginn einem Messzyklus gestartet wird. Wenn diese die Spannung Null erreicht, wechselt der logische Pegel am unteren Komparator auf High und der Zähler startet. Erreicht U_k die unbekannte Messspannung U_x, wechselt der logische Pegel am oberen Komparator auf Low und der Zähler stoppt. Der so ermittelte Zählerstand ergibt den digitalen Messwert. Es handelt sich also um ein Zählverfahren, wobei die Spannungsänderung dU_k / dt während einer Taktperiode einem Quantisierungsintervall entspricht.

Dieses einfache Verfahren wird in der Praxis jedoch nicht eingesetzt, denn der Sägezahngenerator hat durch alterungs- oder temperaturbedingte Änderungen seiner Bauelemente nur eine begrenzte Genauigkeit.

Praktisch eingesetzt wird das *Doppelflanken-* oder *Dual-Slope-Verfahren*. Hierbei wird, anders als beim Single-Slope-Verfahren, die Messgröße U_x und nicht eine Referenzspannung über eine feste Zeit t_1 integriert. Abb. 12.16 und 12.17 zeigen Prinzipschaltbild und Zeitverlauf bei einer Messung. Während der festen Messdauer t_1 wird in einem Integrator die unbekannte Spannung U_x bis zur Endspannung U_a aufintegriert. Anschließend schaltet der Zähler für t_1 um und die Spannung U_a wird über eine negative Referenzspannung $-U_{\text{ref}}$ wieder bis auf die Spannung 0 integriert. Die Zeit t_2, die hierfür erforderlich ist, wird gemessen und ergibt den digitalen Messwert für die Spannung U_x.

Der Vorteil dieser Messung liegt darin, dass die Bauteile R und C für beide Integrationszyklen verwendet werden. Dadurch ist die Messung unabhängig von Parameterschwankungen bei diesen Bauteilen. Es ist lediglich eine stabile Referenzspannung erforderlich, die durch Band-Gap-Dioden mit hoher Genauigkeit zur Verfügung steht.

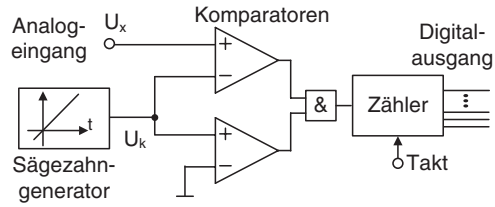


Abb. 12.15 Blockschaltbild eines ADUs nach dem Single-Slope-Verfahren

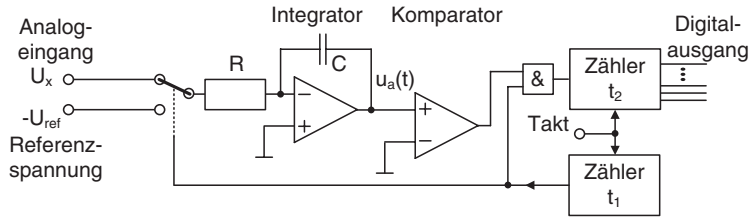


Abb. 12.16 Prinzipschaltbild eines AD-Umsetzers nach dem Dual-Slope-Verfahren

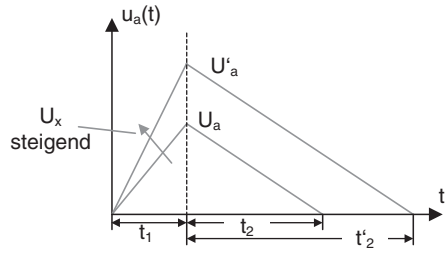


Abb. 12.17 Spannungsverläufe im Dual-Slope-Umsetzer während des Messzyklus

Die Messdauer t_1 kann zu einem Vielfachen der Periodendauer der 50 Hz Netzspannung, also zu $n \cdot 20$ ms gewählt werden. In diesem Fall haben Störungen durch die Netzspannung keinen Einfluss auf das Messergebnis.

Die Vorteile dieses Messverfahrens sind also:

- gute Störspannungsunterdrückung, da integrierendes Verfahren
- unabhängig von alterungs- und temperaturbedingten Änderungen der Bauelemente und des Taktoszillators
- Die Langzeitpräzision wird nur durch U_{ref} bestimmt, die sehr präzise erzeugt werden kann
- erzielbare Genauigkeit: ca. 0,001 %, d. h. 15–16 bit bzw. 5 Dezimalstellen

Nachteil:

- Das Verfahren arbeitet relativ langsam

Die häufigste Anwendung findet dieser Umsetzertyp in Digitalvoltmetern.

12.2.7 Sigma-Delta-Umsetzer

Ein *Sigma-Delta-Umsetzer* (ΣΔ-Umsetzer) kombiniert die Rückführung eines 1-Bit-DA-Umsetzers mit einem Integrator und einer Überabtastung. Das Blockschaltbild Abb. 12.18 zeigt den Aufbau. Der Eingangswert U_x wird mit der Rückführung

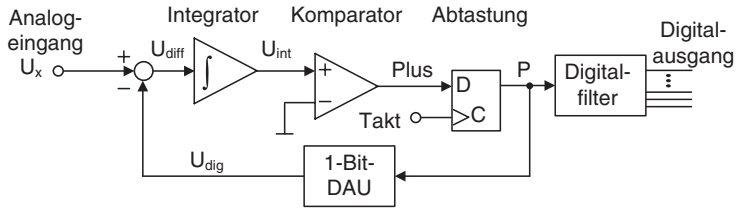


Abb. 12.18 Blockschaltbild eines Sigma-Delta-Umsetzers

kombiniert und in einem Integrator weiterverarbeitet. Dieser Integrator ist ähnlich wie beim Dual-Slope-Verfahren aufgebaut. Ein Komparator ermittelt, ob das Integral positiv oder negativ ist und arbeitet daher als 1-Bit-ADU. Der ADU-Ausgang *Plus* ist eine binäre Information und geht an ein D-Flip-Flop, wo der Wert mit hoher Taktfrequenz abgetastet wird. Als Rückführung geht der abgetastete Vergleichswert *P* auf einen 1-Bit-DA-Umsetzer, dessen Ausgang vom Eingangswert abgezogen wird.

Der Digitalausgang berechnet sich durch einen digitalen Filter aus der Folge von Vergleichswerten *P*. Der Name Sigma-Delta bildet sich aus den Funktionselementen Integration (Sigma) und der Differenzbildung mit der Rückkopplung (Delta).

Zum Verständnis des Funktionsprinzips wird der Zeitablauf bei einer Umsetzung in Tab. 12.3 Schritt für Schritt erläutert. Als Messbereich wird ±1 V angenommen und auch der Ausgang des DAU beträgt +1 V oder − 1 V. Als Spannung am Analogeingang U_x soll 0,6 V gemessen werden. Für den Zeitschritt 1 wird zum besseren Verständnis die Rückführung weggelassen, daher ist $U_{dig} = 0$ V (in der Tabelle mit *0** markiert). Ebenfalls wird angenommen, dass der Integrator mit der Spannung $U_{int} = 0$ V startet.

In den Zeitschritten erfolgen dann die folgenden Berechnungen:

1. U_x plus U_{dig} ergeben 0,6 V, die im Integrator verarbeitet werden. Dieser Wert ist positiv, daher ist Plus gleich 1.
2. Die Rückführung nimmt den vorherigen Wert von *Plus* und ergibt darum $U_{dig} = 1$ V. Dieser Wert wird von U_x abgezogen, sodass $U_{diff} = -0{,}4$ V ist. Addiert zum vorherigen Wert des Integrators bleibt $U_{int} = 0{,}2$ V. Dieser Wert ist positiv, daher ist *Plus* gleich 1.
3. Die Rückführung ergibt erneut $U_{dig} = 1$ V, sodass wiederum $U_{diff} = -0{,}4$ V ist. Der Wert des Integrators wird $U_{int} = -0{,}2$ V. Dieser Wert ist negativ, daher ist *Plus* gleich 0.
4. Wegen des negativen Werts im Integrator ergibt die Rückführung nun $U_{dig} = -1$ V. Daher ist $U_{diff} = 1{,}6$ V und der Wert des Integrators wird $U_{int} = 1{,}4$ V. Dieser Wert ist positiv, daher wird *Plus* wieder gleich 1.
5. Die Rückführung ergibt wieder $U_{dig} = 1$ V, darum wird $U_{diff} = -0{,}4$ V. Der Wert des Integrators wird $U_{int} = 1$ V. Dieser Wert ist positiv, daher ist *Plus* gleich 1.
6. Der weitere Zeitablauf kann aus der Tabelle abgelesen werden.

Beim Zeitablauf in Tab. 12.3 besteht die Pulsfolge am digitalen Filter aus vier Einsen und einer Null. Die Pulsfolge wird Tiefpass-gefiltert und ergibt einen 1-Anteil von 80 %.

Tab. 12.3 Zeitablauf einer AD-Umsetzung mit Sigma-Delta-Umsetzer

Zeit-schritt	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
U_x [in V]	0,6	0,6	0,6	0,6	0,6	0,6	0,6	0,6	0,6	0,6	0,6	0,6	0,6	0,6	0,6
U_{dig} [in V]	*0**	+1	+1	−1	+1	+1	+1	+1	−1	+1	+1	+1	+1	−1	+1
U_{diff} [in V]	0,6	−0,4	−0,4	1,6	−0,4	−0,4	−0,4	−0,4	1,6	−0,4	−0,4	−0,4	−0,4	1,6	−0,4
U_{int} [in V]	0,6	0,2	−0,2	1,4	1,0	0,6	0,2	−0,2	1,4	1,0	0,6	0,2	−0,2	1,4	1,0
Plus [binär]	1	1	0	1	1	1	1	0	1	1	1	1	0	1	1

Dieser Wert bezieht sich auf den Messbereich von ±1 V und entspricht $U_x = -1\,\text{V} + 0{,}8 \cdot 2\,\text{V} = -1\,\text{V} + 1{,}6\,\text{V} = 0{,}6\,\text{V}$.

Die Tabelle zeigt die Umsetzung eines konstanten Eingangswertes. Wenn sich U_x ändert, wird sich auch die Pulsfolge nach mehreren Schritten an den geänderten Eingangswert anpassen.

Der Sigma-Delta-Umsetzer versucht also mit Pulsen von +1 V und −1 V die Eingangsspannung nachzubilden. Dies sind recht grobe Schritte; im Gegenzug dafür wird die Frequenz der Schritte sehr hoch gewählt. Der Unterschied zwischen höchster Frequenz des Eingangssignals und Abtastrate wird als Oversampling Ratio OSR bezeichnet und hierfür sind Faktoren von 100 und höher möglich. Diese Arbeitsfrequenz passt sehr gut zu modernen CMOS-Prozessen, die hohe Taktfrequenzen ermöglichen.

Das Messprinzip des Sigma-Delta-Umsetzers unterscheidet sich damit maßgebend von dem der bisher dargestellten Umsetzer. Letztere liefern bei einer Abtastrate, die möglichst nahe der unteren durch das Abtasttheorem erlaubten Grenze liegt, jeweils ein vollständiges Codewort. Der Sigma-Delta-Umsetzer liefert dagegen eine 1-Bit-Folge mit sehr viel höherer Frequenz. Dieses Verfahren nennt man daher auch *Oversampling-Technik*. Der Sigma-Delta-Umsetzer hat gegenüber anderen Umsetzern eine Reihe von Vorteilen:

1. Er kann nahezu völlig aus digitalen Komponenten aufgebaut werden. Die Anforderungen an die 1-Bit-Umsetzung sind nicht sehr hoch.
2. Er wirkt für das Eingangssignal wie ein Tiefpass, für das Quantisierungsfehlersignal jedoch wie ein Hochpass. Das Spektrum des Quantisierungsfehlersignals wird daher schwerpunkthaft in die Nähe der sehr hohen Abtastfrequenz verschoben. Der digital arbeitende Tiefpass eliminiert erhebliche Teile davon und kann so dimensioniert werden, dass er 50 Hz-Störungen unterdrückt.
3. Dem Umsetzerprinzip ist eine monotone Quantisierungskennlinie inhärent.

4. Wegen der sehr hohen Abtastfrequenz kommt der Sigma-Delta-Umsetzer generell ohne Abtast-Halteglied aus.
5. Derzeit liefert dieses Verfahren die höchsten verfügbaren Auflösungen.

Den Vorteilen stehen auch einige Nachteile gegenüber:

1. Wegen des mittelwertbildenden digitalen Filters gibt es eine große Latenzzeit zwischen dem ersten Abtastwert und dem ersten Codewort. Daher eignet sich dieser Umsetzer nicht zum Multiplexbetrieb für mehrere Signalquellen.
2. Gegenüber Flash-Umsetzern arbeitet das Sigma-Delta-Verfahren langsam.

Sigma-Delta-Umsetzer nach dem Oversampling-Prinzip haben sich inzwischen mit Auflösungen von 16 bit in der hochwertigen Tonsignalverarbeitung etabliert. Weiterhin wird dieses Verfahren in der Telemetrie und zur präzisen Überwachung langsam veränderlicher Signale, beispielsweise bei Dehnungsmessstreifen eingesetzt.

12.3 Verfahren zur Digital-Analog-Umsetzung

Digital-Analog-Umsetzer (DAU) dienen der Rückgewinnung des Analogsignals aus codierten digitalen Werten. Dabei verursacht die Zeitdiskretisierung prinzipiell keine Fehler, wenn das Abtasttheorem eingehalten wird. Die Wertdiskretisierung führt zu Quantisierungsfehlern, die systematischer Natur sind und nicht mehr eliminiert werden können. Durch Wahl einer hohen Auflösung können die Quantisierungsfehler jedoch sehr klein gehalten werden.

Bei der Umsetzung liefert der DAU Impulse endlicher Breite t_s und mit der Höhe, die durch die Digitalwerte vorgegeben ist (Abb. 12.19). Dieses Signal ist also noch zeitdiskret. Durch anschließende Filterung in einem Tiefpass (Interpolator-Tiefpass) wird dieses Signal wieder zu einer stetigen Analogfunktion interpoliert.

Theoretisch sollte die Impulsbreite t_s möglichst klein sein, um keine zusätzlichen Frequenzen für das Ausgangssignal zu erzeugen. In der Realität wird jedoch aus zwei Gründen eine größere Impulsbreite gewählt, die meist der Periodendauer der Abtastung T_{abt} entspricht.

- Durch die größere Breite des Signals ist eine höhere Signalleistung vorhanden.
- Es ist keine Abschaltung des Signals zwischen den Ausgabewerten erforderlich.

Das resultierende Rechtecksignal ergibt eine merkliche Verzerrung des Ausgangssignals, denn das Spektrum des digitalen Signals wird mit dem Spektrum einer Rechteckfunktion der Breite T_{abt} multipliziert. Diese Verzerrung kann jedoch durch ein nachfolgendes analoges Filter wieder eliminiert werden. Die Struktur eines DAUs entspricht damit Abb. 12.20.

12.3.1 Direktverfahren

Im *Direktverfahren* werden die möglichen Ausgangsspannungen des n-Bit-Umsetzers in einem Spannungsteiler aus 2^n gleichen Widerständen gebildet. Durch einen Multiplexer wird eine Spannung ausgewählt und an den Ausgang gegeben (Abb. 12.21). Die Schalter des Multiplexers sind natürlich keine mechanischen Schalter, sondern werden durch Transistoren implementiert. Die Widerstandsreihe führt auch zu der englischen Bezeichnung „String Architecture“. Das Verfahren ähnelt dem Parallelverfahren zur AD-Umsetzung aus Abb. 12.9.

Der Vorteil dieses Verfahrens ist eine relativ gleichmäßige Schrittweite der Umsetzungskennlinie, denn die Toleranzen der Widerstände entsprechen der Schrittweite zweier Ausgabewerte. Dadurch können insbesondere keine Monotoniefehler (siehe Abschn. 12.4.1.6) auftreten.

Der Nachteil des Verfahrens ist der hohe Aufwand an Widerständen und Schaltern, insbesondere bei höheren Wortbreiten. Es gibt jedoch erweiterte Strukturen, bei denen nicht alle 2^n Ausgangsspannungen direkt erzeugt werden, sondern eine Interpolation zwischen Abgriffen der Widerstandsreihe erfolgt.

12.3.2 Summation gewichteter Ströme

Das Verfahren der *Summation gewichteter Ströme* basiert auf dem Prinzip, dass für jedes auf 1 gesetzte Bit des Dualwortes ein dem Bitgewicht entsprechender Strom erzeugt wird. Dann werden alle Ströme rückwirkungsfrei summiert, beispielsweise durch einen Operationsverstärker (OP). Für einen DAU mit n bit ergibt sich daraus die in Abb. 12.22 gezeigte Schaltung. Das digitale Codewort steuert die Schalter b_0 bis b_{n-1}. Die Ausgangsspannung des OPs beträgt dann:

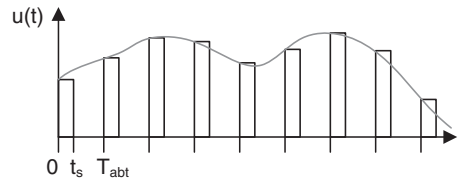


Abb. 12.19 Das Ausgangssignal eines DAU besteht prinzipiell aus Impulsen endlicher Breite

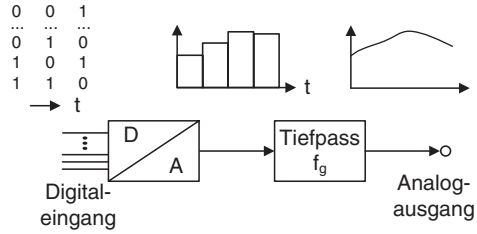


Abb. 12.20 Prinzipschaltbild eines Umsetzers digitaler in analoge Signale

$$U_{DA} = -R_F(b_0 \cdot i_0 + b_1 \cdot i_1 + b_2 \cdot i_2 + \ldots + b_{n-1} \cdot i_{n-1})$$

Für die Ströme gilt

$$i_k = \frac{U_{\text{ref}}}{R/2^k} = \frac{U_{\text{ref}} \cdot 2^k}{R}$$

Damit berechnet sich die Ausgangsspannung U_{DA} des Umsetzers zu

$$U_{DA} = -U_{\text{ref}} \frac{R_F}{R} \sum_{k=0}^{n-1} b_k \cdot 2^k$$

Es ist ersichtlich, dass die Ausgangsspannung U_{DA} eine Form hat, die dem vorgegebenen dualen Wert bis auf eine multiplikative Konstante entspricht. Die elektronischen Schalter können in Bipolar- oder CMOS-Technik realisiert werden.

Ein wesentlicher Nachteil der Schaltungsstruktur ist, dass sich die Widerstandswerte für einen n-Bit-Umsetzer um den Faktor 2^{n-1} unterscheiden. Dieses ist in monolithischer Technik schwer zu realisieren, da der herstellbare Wertebereich technologisch begrenzt ist. Außerdem sind die Anforderungen an die Präzision der kleineren Widerstände sehr hoch. Der kleinste Widerstand R hat den höchsten Strombeitrag und sein Stromfehler sollte kleiner als 1/2 Stelle des Ergebnisses sein. Das bedeutet, der Fehler darf nur so groß wie die Hälfte des Strombeitrags des größten Widerstands $R/2^{n-1}$ sein. Darum muss die Genauigkeit besser als 2^{-n} sein. Bei einem 12-Bit-Umsetzer benötigt der kleinste Widerstand also die Genauigkeit von $2^{-12} = 2{,}44 \cdot 10^{-4}$ und dieser Wert ist praktisch nicht zu erreichen.

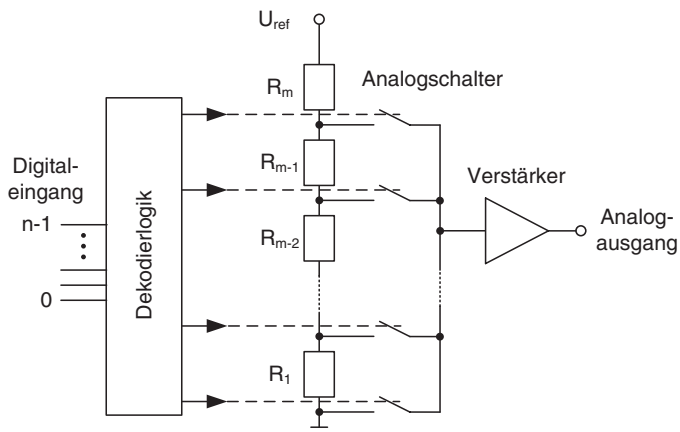


Abb. 12.21 DA-Umsetzung im Direktverfahren

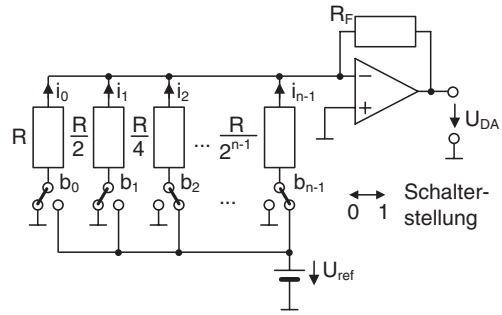


Abb. 12.22 Prinzipschaltbild eines DAU nach dem Summationsprinzip gewichteter Ströme

Aus diesen Gründen werden integrierte DAUs nicht nach dem oben dargestellten Prinzip realisiert, sondern durch fortgesetzte Spannungsteilung in einem Kettenleiternetzwerk. Dieses Verfahren wird im nächsten Kapitel beschrieben.

Eingesetzt wird die Umsetzung mit Summation gewichteter Ströme bei Anwendungen mit geringer Wortbreite. Ein Beispiel ist die Codierung von Tasten für Mikrocontroller. Wenn ein Mikrocontroller durch mehrere Taster bedient werden soll, wäre prinzipiell für jeden Taster eine Eingangsleitung erforderlich. Stattdessen können vier bis sechs Taster über ein Widerstandsnetzwerk wie in Abb. 12.22 an einen einzigen Analogeingang des Mikrocontrollers gegeben werden, so dass Eingangsleitungen gespart werden. Der Operationsverstärker ist dabei nicht erforderlich.

12.3.3 R-2R-Leiternetzwerk

Die Arbeitsweise dieses DA-Umsetzertyps basiert prinzipiell auf dem gleichen Verfahren wie der zuvor dargestellte, denn es werden Ströme addiert, die dem Wert der einzelnen Dualstellen des vorgegebenen Digitalwortes entsprechen. Allerdings werden hier die Ströme mit stufenweise gleichgroßen Widerständen anhand fortgesetzter Spannungsteilung in einem Leiternetzwerk erzeugt. Grundelement ist dabei ein π-Glied, das als belasteter Spannungsteiler mit folgenden Eigenschaften betrieben wird:

- Belastet man den Spannungsteiler mit einem Abschlusswiderstand Z, so soll sein Eingangswiderstand ebenfalls Z sein. Das ermöglicht eine einfache Kettenschaltung der einzelnen Spannungsteiler.
- Der Teilerfaktor in jeder abgeschlossenen Teilerstufe soll entsprechend der dualen Abstufung 2 sein.

Diese Forderungen lassen sich mit symmetrischen Vierpolen erreichen, die mit ihrem Wellenwiderstand abgeschlossen sind. Eine Rechnung liefert das in Abb. 12.23 dargestellte verlängerbare Kettenleiternetzwerk. Wegen der charakteristischen

Widerstandswerte wird diese Schaltung auch als R-2R-Leiternetzwerk bezeichnet. Der Wert für den Widerstand R kann frei gewählt werden.

Für die Verwendung als ADU wird an die Klemmen A und B eine Referenzspannung U_{ref} angeschlossen. Der Spannungsteilerkette werden über Stromschalter die Einzelströme gemäß dem vorliegenden Binärwort entnommen und am Summationspunkt eines OP rückwirkungsfrei addiert (Abb. 12.24).

Für die eingetragenen Spannungen gilt:

$$\begin{aligned} U_3 &= U_{\text{ref}} \\ U_2 &= U_3/2 \\ U_1 &= U_2/2 \\ U_0 &= U_1/2 \end{aligned}$$

Damit ergeben sich die Ströme zu:

$$\begin{aligned} I_3 &= \frac{U_3}{2R} = \frac{U_{\text{ref}}}{2R} \\ I_2 &= \frac{U_2}{2R} = \frac{U_{\text{ref}}/2}{2R} = \frac{I_3}{2} \\ I_1 &= \frac{U_1}{2R} = \frac{U_{\text{ref}}/4}{2R} = \frac{I_3}{4} \\ I_0 &= \frac{U_0}{2R} = \frac{U_{\text{ref}}/8}{2R} = \frac{I_3}{8} \end{aligned}$$

Die Stromschalter werden in Bipolar- oder CMOS-Technik realisiert. Es tritt lediglich das gut realisierbare Widerstandsverhältnis 2:1 auf. Ein typischer Wert für R ist 500 Ω. Nach diesem Prinzip arbeiten die meisten käuflichen DAUs in monolithischer und hybrider Technik. Außerdem ist in ADUs mit sukzessiver Approximation im Gegenkopplungspfad ein DAU dieses Typs enthalten.

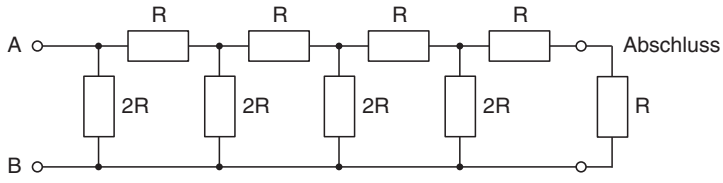


Abb. 12.23 R-2R-Leiternetzwerk für einen Digital-Analog-Umsetzer

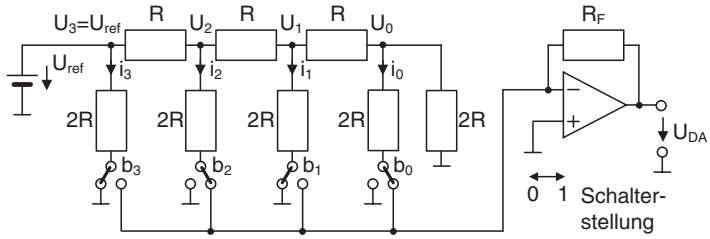


Abb. 12.24 Prinzipschaltbild eines 4-Bit-DAUs mit R-2R-Leiternetzwerk

12.3.4 Pulsweitenmodulation

Die *Pulsweitenmodulation* (PWM) erzeugt eine analoge Ausgangsgröße durch schnellen Wechsel zwischen zwei Spannungswerten. Das Verhältnis zwischen den Zeiten für die Ausgangspegel bestimmt die analoge Ausgangsgröße. Abb. 12.25 zeigt den Zeitablauf für zwei Ausgabewerte. Das Ausgangssignal wechselt periodisch zwischen High-Pegel U_H und Low-Pegel U_L. Die Dauer des High-Pegels t_H dividiert durch die Periodendauer T_{Per} wird als Tastverhältnis bezeichnet.

Aus der Pulsfolge kann durch einen Tiefpass eine Mittelwertbildung erfolgen, um eine analoge Ausgangsspannung zu erzeugen; im einfachsten Fall reicht ein Kondensator. Die analoge Ausgangsspannung berechnet sich zu

$$U_{DA} = U_L + \frac{t_H}{T_{Per}}(U_H - U_L)$$

Als Beispiel wird für den Zeitverlauf in Abb. 12.25 ein High-Pegel von 3,3 V und ein Low-Pegel von 0 V angenommen. Dann ergibt sich für das Tastverhältnis von 70 % die Ausgangsspannung 2,31 V und für 20 % die Spannung 0,66 V.

Es gibt jedoch auch Anwendungen, bei denen keine analoge Ausgangsspannung benötigt wird, sondern stattdessen der angesteuerte Aktor oder der nachfolgende Sensor einen Mittelwert bildet.

- Ein Gleichstrommotor kann durch eine PWM angesteuert werden und ergibt verschiedene Drehgeschwindigkeiten. Die Masse der Achse und die Motorlast sorgen für die Mittelwertbildung.
- Wird eine Leuchtdiode mit einer PWM angesteuert, erscheint sie verschieden hell. Die LED ist abwechselnd leuchtend und nicht-leuchtend und das menschliche Auge sorgt für die Mittelwertbildung.

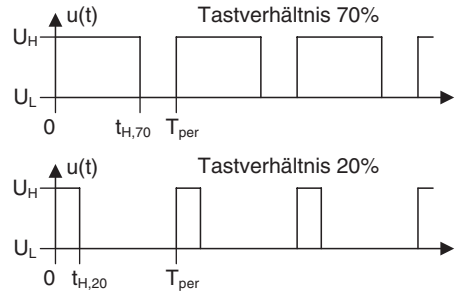


Abb. 12.25 Zeitablauf für eine Pulsweitenmodulation (PWM) für die Ausgabewerte 20 und 70 %

12.4 Eigenschaften realer AD- und DA-Umsetzer

Reale Umsetzerbausteine sind mit Fehlern behaftet. Sie sind bauelemente-, schaltungs- oder prinzipbedingt und können sowohl im ADU als auch im DAU auftreten. Sie lassen sich in statische und dynamische Fehler unterteilen.

Die zunächst betrachteten statischen Fehler treten in ADUs und bis auf den Quantisierungsfehler auch in DAUs auf. Die in den folgenden Kapiteln hierzu dargestellten Diagramme beziehen sich auf ADUs. Durch Spiegelung an der Einheitsgeraden erhält man daraus die entsprechenden Darstellungen für DAUs. Bei dynamischen Fehlern muss zwischen ADUs und DAUs unterschieden werden.

12.4.1 Statische Fehler

Als statische Fehler werden solche Fehler bezeichnet, die nach dem Abklingen aller Einschwingvorgänge übrig bleiben.

12.4.1.1 Quantisierungsfehler

Die Beschränkung auf eine endliche Anzahl darstellbarer Amplitudenstufen bei der AD-Umsetzung verursacht systematische Fehler, deren Amplitude im Allgemeinen $\pm 0{,}5 \cdot Q$ erreichen kann. Nach der DA-Umsetzung ergibt sich dadurch ein Fehlersignal, der *Quantisierungsfehler,* der rauschsignalähnlichen Charakter hat und den Signal-Rausch-Abstand begrenzt. Der Quantisierungsfehler ist auch interpretierbar als Auswirkung der nichtlinearen Stufenkennlinie eines Quantisierers auf das Signal. Da in praktischen Fällen die Stufen der Quantisiererkennlinie sehr klein sind, kann man auch von einer mikroskopischen Nichtlinearität sprechen.

Setzt man eine lineare Quantisierung, ein in jedem Quantisierungsintervall gleich verteiltes Signal und einen mitten im Quantisierungsintervall Q liegenden Repräsentationswert voraus, beträgt die Quantisierungsgeräuschleistung (Noise) N:

$$N = Q^2/12$$

Als Zahlenbeispiel wird ein vollaussteuerndes Sinussignal bei einem Umsetzer mit $m \cdot Q \approx 2^n$ Quantisierungsintervallen angenommen, wobei n die Codewortbreite ist. Hier beträgt die Signalleistung S:

$$S = \left(\frac{m \cdot Q}{2 \cdot \sqrt{2}}\right)^2 = \frac{2^{2n} \cdot Q^2}{8}$$

Dann beträgt der maximal erreichbare Signal-Rausch-Abstand (Signal to Noise Ratio) für das mit n bit digitalisierte Sinussignal

$$SNR = 10 \cdot log \frac{S}{N} = (1,76 + 6,02 \cdot n) dB$$

Unter den oben getroffenen Voraussetzungen ist daher mit einem 12-Bit-Umsetzer ein max. Rauschabstand von $SNR = 74$ dB erreichbar. Dieser Wert entspricht einer guten Signalqualität, somit kann der Quantisierungsfehler bei der Digitalisierung mit einem erträglichen technischen Aufwand relativ klein gehalten werden.

Für die nächsten Betrachtungen wird die Stufenkennlinie mittels einer Geraden durch die Quantisierungsintervallmitten ersetzt (Umsetzerkennlinie). Der ideale lineare AD-Umsetzer hat dann eine Umsetzerkennlinie, wie sie in Abb. 12.26 dargestellt ist. Verwendet man für Ein- und Ausgangsgrößen gleiche Maßstäbe, verläuft die ideale Kennlinie unter 45°. Weicht ein Umsetzer von dieser Kennlinie ab, ist er fehlerhaft.

12.4.1.2 Offsetfehler

Anschaulich gesehen liegt ein *Offsetfehler (Zero Error)* vor, wenn die Umsetzerkennlinie gegenüber der idealen Kennlinie parallelverschoben ist (Abb. 12.27, links). Ursache hierfür ist beispielsweise ein Offsetfehler des Eingangsverstärkers. Konkret entspricht dieser Fehler der Lageabweichung des ersten Übergangswerts oberhalb von Null von der Ideallage bei $0{,}5 \cdot Q$ (siehe auch Abb. 12.8). Der Offsetfehler verursacht einen konstanten absoluten Fehler im gesamten Aussteuerbereich und ist auf null abgleichbar.

Die Angabe des Offsetfehlers im Datenblatt erfolgt üblicherweise in Bruchteilen des Aussteuerbereichs. Der Offsetfehler hat darüber hinaus einen Temperatur-Koeffizienten, der nur mit großem Aufwand kompensiert werden kann.

12.4.1.3 Verstärkungsfehler

Anschaulich gesehen liegt ein *Verstärkungsfehler (Gain Error)* vor, wenn die Kennliniensteigung von der idealen Steigung 1 abweicht (Abb. 12.27, rechts). Er verursacht einen konstanten relativen Fehler im Aussteuerbereich und ist auf null abgleichbar.

Die exakte Definition des Verstärkungsfehlers ist die Abweichung der real vorliegenden Spannungsdifferenz zwischen dem ersten Übergangswert bei $0{,}5 \cdot Q$ und dem letzten bei $U_{max} - 1{,}5 \cdot Q$ vom idealen Wert (siehe Abb. 12.8).

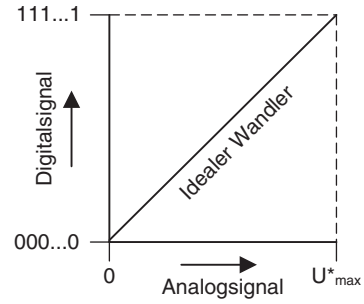


Abb. 12.26 Kennlinie eines idealen AD-Umsetzers

Die Angabe des Verstärkungsfehlers im Datenblatt erfolgt entweder absolut in LSB oder relativ in % des Aussteuerbereichs. Der Verstärkungsfehler hat einen Temperatur-Koeffizienten, der nur mit großem Aufwand kompensiert werden kann.

12.4.1.4 Nichtlinearität

Die *Nichtlinearität (Nonlinearity)* eines Umsetzers, auch Integrale Nichtlinearität (INL) genannt, entspricht der maximalen Kennlinienabweichung von der Geraden durch die Endpunkte des Diagramms.

Nach Abgleich der Offset- und Verstärkungsfehler entspricht sie der maximalen Abweichung von der idealen Kennlinie (Abb. 12.28). Gelegentlich wird allerdings in Datenblättern die Nichtlinearität auch als maximale Abweichung von der bestmöglichen Geraden interpretiert. Dann ist ein Offsetfehler einzustellen, damit die Nichtlinearität den Herstellerangaben entspricht.

Der Grund für Nichtlinearitäten sind ungleich große Quantisierungsintervalle. Die Nichtlinearität kann durch mehrere benachbarte Quantisierungsintervalle verursacht werden, welche Abweichungen in gleicher Richtung haben. Die Angabe der Nichtlinearität erfolgt üblicherweise in Bruchteilen des LSB.

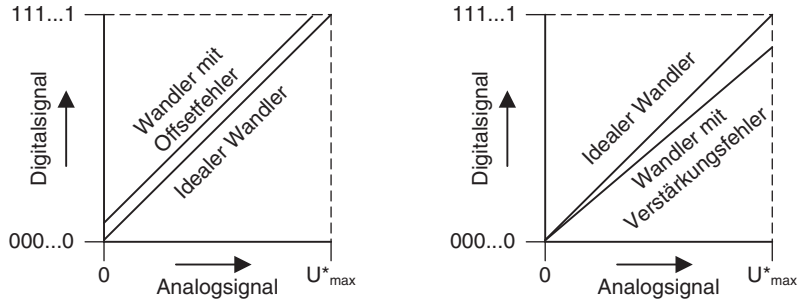


Abb. 12.27 Kennlinien mit Offsetfehler und Verstärkungsfehler

12.4.1.5 Differenzielle Nichtlinearität

Als *differenzielle Nichtlinearität (Differential Nonlinearity)* bezeichnet man die Abweichung der Breite eines Quantisierungsintervalls vom Idealwert *Q*. Dabei bezieht man sich auf dasjenige Quantisierungsintervall mit der größten Abweichung (Abb. 12.29).

Die Angabe im Datenblatt erfolgt üblicherweise in Bruchteilen eines LSB. Ist die differenzielle Nichtlinearität im Datenblatt beispielsweise mit ±0,5 LSB angegeben, müssen alle Quantisierungsintervalle im Bereich 1 LSB ± 0,5 LSB liegen. Eine Sonderform der differenziellen Nichtlinearität liegt vor, wenn einzelne Codeworte fehlen *(Missing Code)*. In diesem Falle beträgt sie 1 LSB.

12.4.1.6 Monotoniefehler

Ein Umsetzer hält die *Monotonität* (Monotonicity) ein, wenn die Umsetzerkennlinie für steigende Eingangswerte stufenweise monoton ansteigt. Hinreichende Bedingung für Monotonität ist, dass die Nichtlinearität kleiner als 2 LSB bleibt. Eine Kennlinie, die diese Bedingung nicht einhält, ist in Abb. 12.30 gezeigt.

12.4.1.7 Betriebsspannungsabhängigkeit der Umsetzerparameter

Die Ausgangsgrößen von Umsetzern sind auch von der Betriebsspannung abhängig. In den Datenblättern wird diese Eigenschaft als *Power Supply Sensitivity* (bzw. *Power Supply Rejection*) bezeichnet. Die Angabe erfolgt als „prozentuale Änderung der Ausgangsgrößen“ dividiert durch „prozentuale Änderung der Betriebsspannung“. In der Regel bezieht sie sich auf Tracking-Netzteile, bei denen die beiden Spannungen unterschiedlicher Polarität sich nur symmetrisch ändern können. Eine Verwendung getrennter Netzteile für positive und negative Betriebsspannung wirkt sich in dieser Beziehung nachteilig aus.

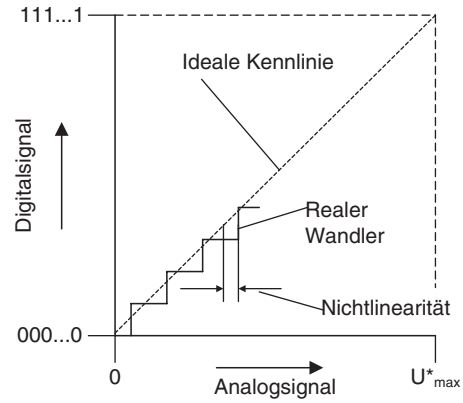


Abb. 12.28 Integrale Nichtlinearität

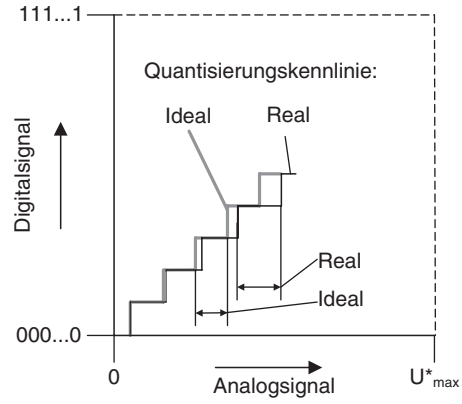


Abb. 12.29 Differenzielle Nichtlinearität

12.4.2 Dynamische Fehler

Dynamische Fehler an Umsetzern treten auf, wenn diese unter nichtstatischen Bedingungen, insbesondere in der Nähe ihrer maximalen Geschwindigkeit, betrieben werden. Sie lassen sich aus den statischen Fehlerkenndaten in der Regel nicht gewinnen.

Dabei muss stets das gesamte System betrachtet werden, das heißt auch das Abtasthalteglied bei ADUs sowie Analogverstärker am Eingang von ADUs und am Ausgang von DAUs tragen zur Systemcharakteristik bei. Sie können die dynamischen Umsetzereigenschaften wegen ihrer Einschwingcharakteristik deutlich einschränken.

Die wichtigsten heute weiterhin üblichen Kenndaten zur Beschreibung des dynamischen Verhaltens von ADUs sind der Signal-Rausch-Abstand, die Effektive Auflösung, die Harmonischen Verzerrungen und das Histogramm. Sie werden in den folgenden Kapiteln dargestellt. Ihre Messung erfolgt auf digitaler Ebene mit schnellen Rechnern und, bis auf das Histogramm, anhand der Fast-Fourier-Transformation (FFT). Daher werden für die Charakterisierung keine Präzisions-DAUs benötigt. Eine für DAUs wichtige dynamische Kenngröße ist die Glitch-Fläche.

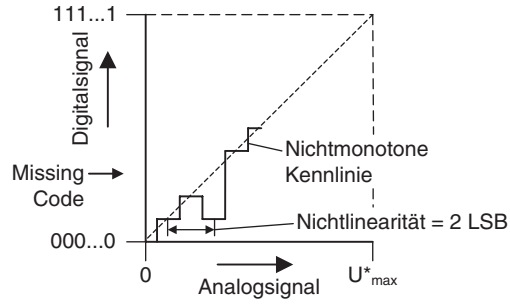


Abb. 12.30 Umsetzerkennlinie mit Monotoniefehler

12.4.2.1 Einschwingzeit

Die *Einschwingzeit (Acquisition Time)* eines DAUs ist die Zeit, die nötig ist, damit sich die Spannung bzw. der Strom bei einem Sprung über den gesamten Aussteuerbereich in einen Toleranzschlauch zurückzieht, der die Breite eines LSB hat und symmetrisch zum stationären Endwert liegt (Abb. 12.31). Die Einschwingzeit setzt sich aus Verzögerungs-, Anstiegs- und Überschwingzeit zusammen. Erst nach Verstreichen der Einschwingzeit entsprechen die Messwerte der geforderten Genauigkeit. Die Überschwingzeit wird auch als *Settling Time* bezeichnet.

12.4.2.2 Signal-Rausch-Abstand und Effektive Auflösung

Das Verhältnis der Leistung S eines den ADU vollkommen aussteuernden Sinussignals zur Quantisierungsgeräuschleistung N entspricht dem *Signal-Rausch-Abstand SNR (Signal to Noise Ratio):*

$$SNR = 10\,log\frac{S}{N}dB$$

Für eine praxisgerechte Größe müssen neben den Quantisierungsfehlern alle weiteren Fehler D *(Distortion)* bei der Umsetzung berücksichtigt werden. D ist die Leistung der weiteren Verzerrungen, die durch nichtideales Verhalten der Bauelemente entstehen. Die daraus resultierende Kenngröße wird als *SINAD (SIgnal to Noise And Distortion ratio)* bezeichnet und wird durch Messungen ermittelt:

$$SINAD = 10\,log\frac{S}{N+D}\,dB$$

Der Signal-Rausch-Abstand eines idealen ADUs berücksichtigt nur die Quantisierungsfehler und errechnet sich zu (siehe Abschn. 12.4.1):

$$SNR = (1,76 + 6,02 \cdot n)dB$$

Für einen idealen ADU mit einer Auflösung von $n = 12$ bit ergibt sich daraus ein Wert von $SNR = 74$ dB.

Reale Umsetzer liefern kleinere Werte, die darüber hinaus mit steigender Signalfrequenz abnehmen. Die Darstellung des über die FFT gemessenen SINAD über

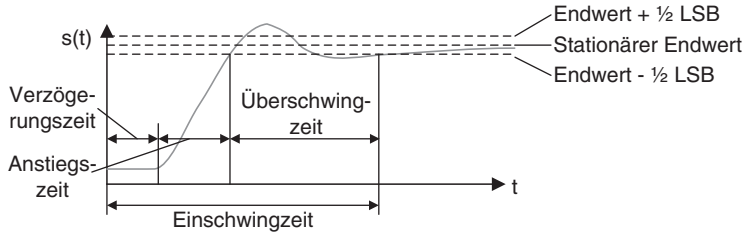


Abb. 12.31 Definition der Einschwingzeit (Acquisition Time) eines DAU oder Abtast-Haltegliedes

der Signalfrequenz wird daher zur Beurteilung der dynamischen Qualität eines ADUs herangezogen.

Benutzt man die gemessenen SINAD-Werte, setzt sie in die obige Beziehung ein und stellt diese nach n um, gewinnt man als äquivalentes Qualitätskriterium die effektive Auflösung n' (Effective Number Of Bits, ENOB) gemäß:

$$n' = \frac{SINAD - 1,76dB}{6,02dB}$$

Ein realer ADU mit der Auflösung von n bit entspricht also in seinem dynamischen Verhalten einem fiktiven idealen ADU mit der Auflösung von n' bit, wobei n' kleiner als n ist. Der Wert von n' ist abhängig von der Frequenz des gemessenen Signals und nimmt für höhere Frequenzen ab. Ein typischer Verlauf der effektiven Auflösung ist in Abb. 12.32 dargestellt.

12.4.2.3 Harmonische Verzerrungen

Zur Bestimmung der *Harmonischen Verzerrungen THD (Total Harmonic Distortion)* werden bezüglich der Anzahl verwendeter Oberwellen unterschiedliche Definitionen benutzt. Sie reichen von 2 bis zur Gesamtzahl aller messbaren Oberwellen. Die Firma Analog Devices benutzt zum Beispiel 5 Oberwellen. Damit ergibt sich die Total Harmonic Distortion zu:

$$THD = 10 \cdot log \frac{U_1^2 + U_2^2 + U_3^2 + U_4^2 + U_5^2}{U_0^2} dB$$

Dabei entspricht U_0 dem Effektivwert der Grundwelle und U_i ist der Effektivwert der i-ten Oberwelle.

12.4.2.4 Histogramm

Das *Histogramm* gestattet Aussagen darüber, wie sich bei einem ADU unter dynamischer Belastung Integrale und Differenzielle Nichtlinearitäten verhalten. Dazu wird der ADU mit einem vollaussteuernden Eingangssignal konstanter Verteilungsdichte gespeist und in einem Digitalrechner die Häufigkeitsverteilung der einzelnen Codeworte durch Zählung ermittelt. Wird ein anderes Testsignal (z. B. Sinus) verwendet, kann die Abweichung von einer konstanten Verteilungsdichte rechnerisch kompensiert werden.

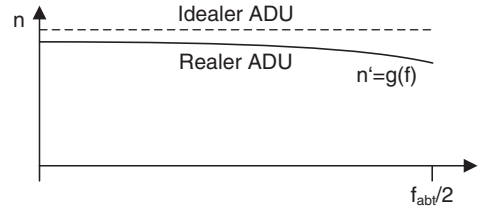


Abb. 12.32 Prinzipieller Verlauf der effektiven Auflösung n′ in bit über der Frequenz für einen realen n-Bit-ADU

Die grafische Darstellung der relativen Häufigkeiten H über den Codeworten ist das Histogramm (diskrete Verteilungsdichte). Ein Prinzipbeispiel zeigt Abb. 12.33.

Für einen in dieser Hinsicht idealen ADU gilt $H = konstant$. Nichtideale Umsetzer weichen hiervon ab. Zeigt das Histogramm etwa benachbarte Spitzen oder Einbrüche, sind das Hinweise auf Differenzielle Nichtlinearitäten. Fehlt eine Linie völlig, ist das zugehörige Codewort nicht ansprechbar *(Missing Code)*.

12.4.2.5 Glitch-Fläche

Die dynamischen Eigenschaften speziell von DAUs können durch die Einschwingzeit nicht hinreichend beschrieben werden. Abhängig von Toleranzen der elektronischen Stromschalter können nämlich am Ausgang kurzzeitig sehr hohe Störimpulse, sogenannte Glitches, auftreten.

Als Beispiel wird betrachtet, dass sich der Eingangscode eines 8-Bit-DAU von 0111 1111 auf 1000 0000 ändert. Alle elektronischen Stromschalter am Leiternetzwerk werden in diesem Falle umgeschaltet. In der Realität geschieht dieses jedoch nicht exakt gleichzeitig. Es wird angenommen, dass der Schalter für das MSB schneller als die anderen schaltet. Dann wird kurzzeitig der Zwischencode 1111 1111 angenommen. Dieses führt am Ausgang zu einem Störimpuls, dessen Höhe dem halben Aussteuerbereich nahekommt, obwohl der Wert sich eigentlich nur um 1 LSB ändern soll.

Im Datenblatt wird diese Größe durch das Integral über die Glitch-Funktion, also die Glitch-Fläche, zum Beispiel in der Einheit nVs bei spezifiziertem Messmodus angegeben. Dieser Wert sollte möglichst klein sein.

Einige Hersteller sehen einstellbare Korrekturschaltungen zur Minimierung der Glitch-Fläche vor. Glitches können auch vermieden werden, indem der Ausgang des DAUs nach Abklingen der Einschwingvorgänge durch Track-and-Hold-Glieder abgetastet und bis zur nächsten Umsetzung konstant gehalten wird. Teilweise sind derartige Deglitch-Einrichtungen bereits in den DAUs enthalten. Allerdings vergrößert sich dadurch die Gesamteinschwingzeit des Umsetzers.

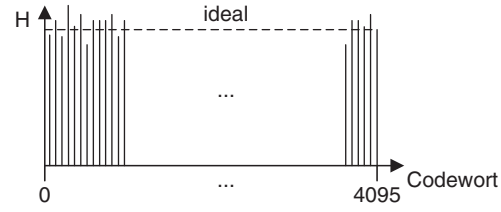


Abb. 12.33 Prinzipielle Darstellung eines Histogramms H für einen ADU mit 4096 darstellbaren Stufen, entsprechend 12 Bit

12.5 Ansteuerung von diskreten AD- und DA-Umsetzern

Analog-Digital- und Digital-Analog-Umsetzer können als Teil eines größeren ASICs implementiert werden. Ein ASIC mit analogen und digitalen Schaltungsteilen wird als *Mixed-Signal-ASIC* bezeichnet. Beispiele hierfür sind:

- Ein Controller für einen LCD-Monitor nimmt Videosignale aus der analogen VGA-Schnittstelle entgegen. Sie werden dann digital verarbeitet, also, wenn erforderlich, auf Bildschirmgröße skaliert, mit On-Screen-Display überlagert und dann an das eigentliche Display weitergegeben.
- Ein ASIC für einen USB-Musik-Player liest digitale Daten aus einem NVRAM und decodiert sie aus dem komprimierten Format. Die digitalen Signale werden dann auf dem ASIC in analoge Signale umgesetzt und ausgegeben.
- Mikrocontroller enthalten oft Analog-Digital-Umsetzer, um analoge Informationen direkt verarbeiten zu können.

Oftmals werden allerdings auch rein digitale ASICs verwendet und eine AD- und DA-Umsetzung in diskreten Bausteinen implementiert. Die Aufteilung eines Systems in Digital-ASIC und diskrete Umsetzer hat insbesondere folgende Vorteile:

- Es ist eine Vielzahl von diskreten ADUs und DAUs verfügbar, die eine Wahl bezüglich Geschwindigkeit, Genauigkeit und Preis ermöglichen.
- Die digitale Verarbeitung in einem Mixed-Signal-ASIC kann den analogen Schaltungsteil stören und die Qualität der Umsetzung einschränken.
- Ein Mixed-Signal-ASIC ist aufwendiger als ein Digital-ASIC und daher teurer.
- Der Zugang zu Mixed-Signal-Fertigungstechnik ist schlechter verfügbar. Außerdem müssen im Entwicklerteam ausreichende Kompetenzen für analoge Schaltungstechnik vorhanden werden.
- Bei FPGAs gibt es kaum Bausteine mit AD- oder DA-Umsetzen.

In diesem Abschnitt wird erläutert, wie diskrete AD- und DA-Umsetzer angesteuert werden. Dabei werden serielle und parallele Ansteuerung verwendet.

12.5.1 Serielle Ansteuerung

Die serielle Ansteuerung diskreter Umsetzer hat den Vorteil, dass nur wenige Leitungen benötigt werden. Die Taktgeschwindigkeit normaler Signalleitungen liegt meist im Bereich von 10 bis 100 MHz. Für einen Datenwert sind, je nach Protokoll und Auflösung, 10 bis 20 Bit erforderlich. Eine serielle Ansteuerung ist also für Abtastraten im Bereich von kHz bis wenige MHz geeignet.

Als Beispiel für Umsetzer mit serieller Ansteuerung werden die Bausteine MCP3201 und MCP4921 von der Firma Microchip betrachtet. Sie verwenden das Serial Peripheral Interface (SPI), welches auch in Kapitel 11 für ein FRAM mit seriellem Interface verwendet wurde.

12.5.1.1 AD-Umsetzer MCP3201

Der Baustein MCP3201 ist ein Analog-Digital-Umsetzer mit 12 bit Auflösung und einer Abtastrate von 100 kHz bei 5 V Betriebsspannung und 50 kHz bei 2,7 V Betriebsspannung. Die Umsetzung erfolgt mit dem Wägeverfahren und sukzessiver Approximation (SAR). Der Baustein benötigt lediglich acht Pins und ist damit sehr kompakt. Seine Anschlüsse sind:

- *IN+* und *IN-,* analoge Eingänge
- *DOUT,* Datenausgang
- *CLK,* Takteingang
- */CS,* Chip-Select und Shutdown
- *VREF,* Referenzspannung
- *VDD* und *VSS,* Versorgungsspannung und Masse

Anmerkung: Im Datenblatt werden für Datenausgang und Takt die Bezeichnungen *DOUT* und *CLK* verwendet. Um die Beschreibung allgemein zu halten, werden hier die SPI-Bezeichnungen *SDO* und *SCK* verwendet.

Der Baustein ermittelt die Differenz zwischen den beiden analogen Eingänge *IN+* und *IN−*. Dabei gibt es jedoch die Einschränkung, dass *IN−* nur um ±100 mV vom Massepegel abweichen darf, sodass kein vollständiger differenzieller Eingang vorhanden ist. Die getrennten Spannungsversorgungen *VDD* und *VREF* ermöglichen die Verwendung einer besonders stabilisierten Referenzspannung.

Die Ansteuerung und Datenübertragung ist in Abb. 12.34 dargestellt. Eine AD-Umsetzung wird durch Wechsel des Eingangs */CS* von 1 auf 0 gestartet. Von der nächsten fallenden Flanke an *SCK* wird für eineinhalb Taktzyklen der analoge Eingangswert im Abtast-Halte-Glied (AHG) erfasst. Die Taktfrequenz an *SCK* darf 1,6 MHz betragen, so dass eine Sample-Zeit t_{sample} von etwa 1 µs möglich ist. Mit den nächsten Takten an *SCK* erfolgt dann die Umsetzung in sukzessiver Approximation und es werden nacheinander eine 0 sowie die Stellen des ermittelten Wertes ausgegeben. In der sukzessiven Approximation wird zuerst das höchstwertige Bit (MSB) ermittelt und darum wird dieses Bit auch zuerst ausgegeben. Nach zwölf Takten ist die Umsetzung beendet *(t_u)* und es ist noch ein weiterer Takt für die Ausgabe des LSB erforderlich *(t_{data})*. Weitere Takte sind erlaubt; eine neue AD-Umsetzung wird jedoch erst durch eine fallende Flanke an */CS* gestartet.

Die Ansteuerung kann direkt durch die SPI-Schnittstelle eines Mikrocontrollers erfolgen. Dazu werden Steuerbefehle gegeben, die zwei Byte einlesen. Die

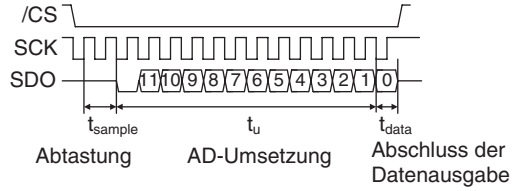


Abb. 12.34 Ansteuerung und Datenübertragung des ADUs MCP3201 von Microchip

SPI-Schnittstelle erzeugt damit 16 Flanken an *SCK* und liest 16 Bit Daten. Aus diesen 16 Bit werden die gültigen 12 Bit der AD-Umsetzung extrahiert.

12.5.1.2 DA-Umsetzer MCP4921

Der Baustein MCP4921 ist ein Digital-Analog-Umsetzer mit 12 bit Auflösung und externer Referenzspannung. Er arbeitet im Direktverfahren. Es gibt verwandte Produkte mit 10 und 8 bit Auflösung, mit zusätzlicher interner Referenzspannung sowie mit zwei Ausgängen. Genau wie der zuvor betrachtete MCP3201 hat auch der MCP4921 acht Pins und ist sehr kompakt. Seine Anschlüsse sind:

- *VOUT,* analoger Ausgang
- *SDI,* Dateneingang
- *SCK,* Takteingang
- */CS,* Chip-Select
- */LDAC,* Übernahmesignal für Daten (Latch DAC, Verwendung optional)
- *VREF,* Referenzspannung
- *VDD* und *VSS,* Versorgungsspannung und Masse

Anmerkung: Hier werden im Datenblatt schon die SPI-Bezeichnungen verwendet.

Die Ansteuerung des Bausteins zeigt Abb. 12.35. Mit */CS* auf 0 wird der Datentransfer begonnen. Dann werden für einen analogen Ausgabewert 16 Bit im SPI-Protokoll übertragen. Das erste Übertragungsbit ist 0, danach kommen drei Steuerbits (werden im nächsten Absatz erläutert) und darauf die 12 Bits, welche als Analogwert ausgegeben werden sollen. Nach der Übertragung wird durch Setzen von */LDAC* auf 0 der Analogwert am Ausgang VOUT aktualisiert. Durch diese Steuerleitung können mehrere DAUs zeitgleich ihre Ausgabe ändern. Falls keine Synchronisation durch */LDAC* benötigt wird, kann dieser Wert konstant auf 0 gelegt werden und der Ausgang wird nach der Datenübertragung automatisch aktualisiert.

Bei der Übertragung werden drei Steuerbits angegeben, die folgende Bedeutung haben. In Abb. 12.35 werden zur besseren Übersichtlichkeit die ersten Buchstaben *B, G, S* angegeben.

- **BUF:** Die Referenzspannung kann gepuffert werden, was zu höherer Eingangsimpedanz bei leichten Einschränkungen in der Umsetzungsqualität führt.

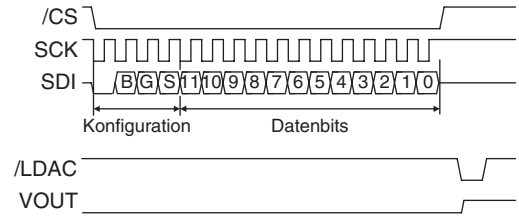


Abb. 12.35 Ansteuerung des DAUs MCP4921 von Microchip

- **/GA (Gain):** Es ist ein Ausgabeverstärker vorhanden, für den der Faktor 1 oder 2 gewählt werden kann.
- **/SHDN (Shutdown):** Der Analogausgang kann zur Verringerung der Verlustleistung abgeschaltet werden.

12.5.2 Parallele Ansteuerung

Für höhere Datenraten ist eine Datenübertragung über SPI nicht mehr möglich. Eine Geschwindigkeitssteigerung kann über parallele Datenleitungen erfolgen.

12.5.2.1 AD-Umsetzer AD9200 mit einfachem Parallelausgang

Der Baustein AD9200 ist ein Pipeline-Analog-Digital-Umsetzer von Analog Devices mit 10 bit Auflösung und einer Abtastrate von 20 MHz. Es sind zwei Gehäuse mit 28 und 48 Pins verfügbar. Die digitale Schnittstelle besteht aus den Anschlüssen:

- *D9* bis *D0,* Datenausgang mit 10 bit Wortbreite
- *OTR,* Out-of-Range Indicator
- *STBY,* Standby, setzt den AD-Umsetzer in den Ruhezustand
- *THREE-STATE,* schaltet die Ausgangsleitungen ab
- *CLK,* als Takt für die interne Operation des Umsetzers sowie für den Datenausgang

Dieses Dateninterface ist sehr einfach. Bei jedem Takt wird ein Datenwort mit 10 Bit ausgegeben. Der Out-of-Range Indicator gibt an, wenn die Eingangswerte außerhalb des Messbereichs liegen. Der Datenausgang wird dann auf den kleinsten oder größten Wert limitiert. In Verbindung mit dem MSB des Datenausgangs kann unterschieden werden, ob ein Überlauf oder ein Unterlauf auftritt.

12.5.2.2 AD-Umsetzer AD9467 mit differenziellem Parallelausgang

Bei höheren Taktgeschwindigkeiten wird ab etwa 100 MHz die Datenübertragung auf einer Platine störanfällig. Für bessere Übertragungseigenschaften werden dann differenzielle Leitungen eingesetzt. Dies bedeutet, ein Ausgang verwendet nicht mehr eine einzelne Leitung, sondern zwei Leitungen, die entgegengesetzte Spannungspegel

einnehmen. Diese werden durch ‚+' und ‚−' gekennzeichnet, das heißt beispielsweise der Takt *CLK* wird auf den Leitungen *CLK*+ und *CLK*− übertragen.

Durch die differenzielle Übertragung kann der Spannungshub zwischen Low- und High-Pegel deutlich verringert werden, denn Störungen betreffen immer beide Leitungen. Aufgrund des geringeren Spannungshubs sind dann auch höhere Taktfrequenzen möglich. Die differenzielle Übertragung wird als *LVDS (Low Voltage Differential Signaling)* bezeichnet.

Der Baustein AD9467 ist ein Pipeline-Analog-Digital-Umsetzer von Analog Devices mit 16 bit Auflösung und einer Abtastrate bis zu 250 MHz. Das Gehäuse hat 72 Pins und die digitale Schnittstelle besteht aus einem parallelen LVDS-Datenausgang und einem seriellen Steuereingang.

Der parallele LVDS-Datenausgang arbeitet mit Double-Data-Rate (DDR), einer Technik, die bereits in Kapitel 11 in Zusammenhang mit DDR-SDRAMs vorgestellt wurde. Das heißt, es werden pro Taktzyklus nacheinander zwei Bit auf einer Datenleitung ausgegeben. Diese Datenleitung ist dann in zwei Polaritäten vorhanden, also als ‚+' und ‚−'. Die Datenleitungen für beispielsweise die Bits 15 und 14 werden als *D15*+/*D14*+ und *D15*−/*D14*− bezeichnet. Der Datenausgang hat insgesamt die folgenden Anschlüsse:

- *D15*+/*D14*+ und *D15*−/*D14*− bis *D1*+/*D0*+ und *D1*−/*D0*−, Datenausgang mit 8 bit LVDS-Werten auf 16 Leitungen
- *OTR*+ und *OTR*−, Out-of-Range Indicator (2 Leitungen)
- *CLK*+ und *CLK*−, Takteingang (2 Leitungen)
- *DCO*+ und *DCO*−, Taktausgang (2 Leitungen)

Der Zeitablauf von Datenerfassung und -ausgabe ist in Abb. 12.36 dargestellt. Die steigende Flanke am Takteingang *CLK*+ bestimmt die Abtastzeitpunkte des analogen Eingangssignals. Der Datenausgang hat ein eigenes Taktsignal *DCO,* mit dem die Datenbits von der nachfolgenden Schaltung übernommen werden müssen.

Die Umsetzung des analogen Eingangswerts benötigt aufgrund des Pipeline-Verfahrens 16 Takte, sodass der Ausgangswert erst nach dieser Latenzzeit ausgegeben wird. Während der Umsetzung werden weitere Daten erfasst und jeweils nach der Latenzzeit von 16 Takten ausgegeben. Die Latenzzeit entspricht der Wortbreite des ADUs von 16 bit.

Der vergrößerte Ausschnitt in Abb. 12.36 zeigt den Zeitablauf bei der Datenausgabe. Mit der steigenden Flanke von *DCO*+ wird Bit 15 für den Abtastwert N-16 ausgegeben (Bezeichnung: $D15^{N-16}$). Mit der fallenden Flanke an *DCO*+ folgt einen halben Takt später Bit 14 für diesen Abtastwert. Im darauffolgenden Takt folgen die Daten für Abtastwert N-15. Parallel liegen auf den anderen 7 LVDS-Leitungen die weiteren Bits des Datenworts an.

Außerdem enthält der Baustein AD9467 einen seriellen Steuereingang mit SPI-Protokoll (vergleiche Abb. 12.34). Hierüber können Konfigurationsregister geschrieben

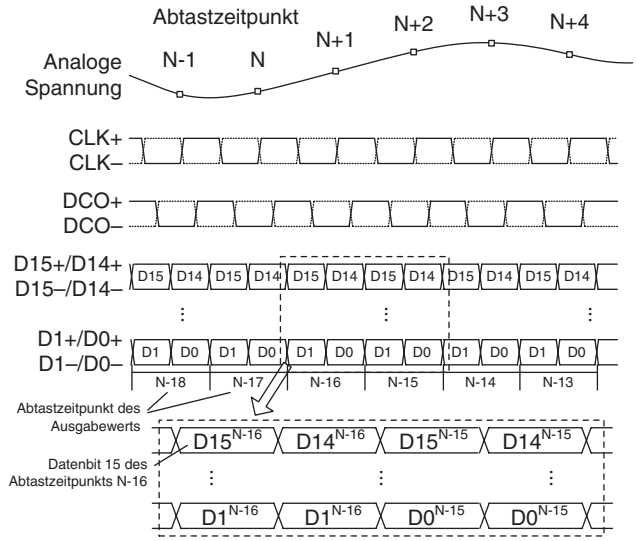


Abb. 12.36 Datenerfassung und LVDS-Datenausgang des ADUs AD9467

und gelesen werden. Diese Konfiguration betrifft analoges und digitales Verhalten, beispielsweise:

- Justierung des Spannungsmessbereichs
- Wahl des Datenformats zwischen dual, 2er-Komplement und Gray-Code
- Ausgabe vordefinierter Daten zu Testzwecken

12.5.3 Serielle Hochgeschwindigkeitsschnittstelle JESD204B

Die im vorherigen Abschnitt vorgestellte parallele Schnittstelle benötigt 20 Leitungen, die auf einer Platine paarweise parallel geführt müssen und zudem die gleichen Länge haben sollen. Eine Alternative zu dieser aufwendigen Verbindung ist die serielle Hochgeschwindigkeitsschnittstelle *JESD204B*, welche von der Standardisierungsorganisation *JEDEC (Joint Electron Device Engineering Council)* definiert wird.

Ein wesentliches Problem bei hohen Übertragungsgeschwindigkeiten auf der Platine ist nicht unbedingt die Geschwindigkeit des Datensignals sondern die Synchronisierung von Daten und Takt. Aus diesem Grund wird beim, im vorherigen Abschnitt beschriebenen, parallelen Interface des AD9467 der Takt zusammen mit den Daten ausgegeben, damit diese möglichst die gleiche Laufzeit haben. Noch höhere Taktraten sind möglich, wenn der Empfänger den Takt aus den empfangenen Daten rekonstruieren kann. Dieses Prinzip wird für die JESD204B-Übertragung genutzt.

Für die Taktrekonstruktion muss sichergestellt sein, dass die Daten genügend Taktflanken besitzen. Wird beispielsweise der Wert 0 mit 0000 0000 codiert und mehrfach nacheinander ausgegeben, kann der Empfangsbaustein hieraus keinen Takt erkennen. Als Lösung dieses Problems wird ein spezieller Code mit redundanter Wortlänge verwendet. Dazu werden die 8 Bit eines Byte mit 10 Stellen codiert. Von den 1024 möglichen Codewörtern werden nur solche verwendet, bei denen mindestens alle 5 Takte eine Signalflanke auftritt. Damit ist sichergestellt, dass der Takt aus den Daten zurückgewonnen werden kann. Der entsprechende Code wird als *8b/10b-Code* bezeichnet und auch für andere Anwendungen in der Kommunikationstechnik verwendet.

Der Baustein ADC32J45 von Texas Instruments ist ein ADU mit JESD204B-Schnittstelle. Er hat zwei Analogeingänge und setzt diese mit einer Abtastrate von 160 MHz und 14 bit Genauigkeit um. Für die Konfiguration des Bausteins ist zusätzlich ein SPI-Port vorhanden.

Abb. 12.37 zeigt die wichtigsten Signale des JESD204B-Datenausgangs in vereinfachter Darstellung. Der Baustein erhält vom Signalverarbeitungs-ASIC den Takt *CLK* und das Steuersignal *SYNC,* beide als differentielles LVDS-Signal. Vom ADU gehen zwei LVDS-Signale *DA* und *DB* mit den Daten der beiden Analogeingänge an das Signalverarbeitungs-ASIC. Die Datenübertragung erfolgt im 8b/10b-Format mit 10-facher Geschwindigkeit des Taktsignals. Bei positivem und bei negativem Pegel von *CLK* wird jeweils ein Byte und somit pro Taktzyklus die 14 Bit des Messwertes und 2 ungenutzte Bits übertragen.

Mit dem Steuersignal SYNC wird am Beginn einer Übertragung der Empfangstakt im Signalverarbeitungs-ASIC synchronisiert. Ist *SYNC+* gleich 0 sendet der ADU ein festes Steuerwort. Sobald dieses Steuerwort mehrfach korrekt erkannt wurde, ist die Taktsynchronisierung erfolgt und *SYNC+* wird auf 1 gesetzt. Danach sendet der ADU noch Steuerworte zur sogenannten Framesynchronisierung und dann folgen die Daten der AD-Umsetzung.

Für die Synchronisierung und Decodierung des 8b/10b-Codes ist im Signalverarbeitungs-ASIC ein entsprechendes Schaltungsmodul erforderlich. Für FPGAs

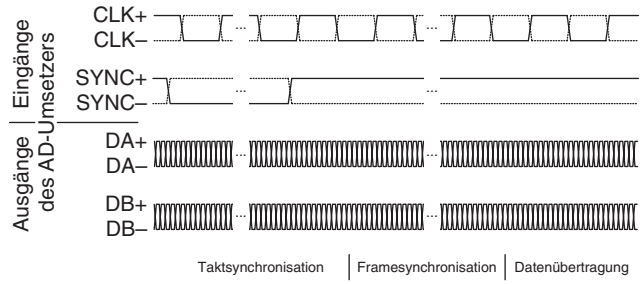


Abb. 12.37 Vereinfachter Zeitablauf am Datenausgang des ADUs ADC32J45 mit JESD204B-Schnittstelle

werden von den Herstellern JESD204B-Interfaces angeboten, welche die Verwendung dieser Schnittstelle vereinfachen.

12.6 Übungsaufgaben

Haben Sie den Inhalt des Kapitels verstanden? Prüfen Sie sich mit den Aufgaben und Fragen am Kapitelende. Die Lösungen und Antworten finden Sie am Ende des Buches.

Aufgabe 12.1
Ordnen Sie den AD-Umsetzern jeweils die passende Kurzbeschreibung zu.

a) Dual-Slope-Verfahren
b) Parallelverfahren
c) Sigma-Delta-Umsetzer
d) Wägeverfahren

1. Gleichzeitiger Vergleich mit $2^n - 1$ Komparatoren
2. Integration von Referenzspannung und Messspannung
3. Hohe Überabtastung des Eingangswertes und 1-Bit-Umsetzung
4. Schrittweise Bestimmung der einzelnen Bits

Aufgabe 12.2
Ordnen Sie den DA-Umsetzern jeweils die passende Kurzbeschreibung zu.

a) Pulsweitenmodulation
b) Summation gewichteter Ströme
c) Direktverfahren
d) R-2R-Leiternetzwerk

1. Verwendung von 2^n gleichen Widerständen
2. Verwendung von einer Widerstandskette mit Widerständen gleicher Größenordnung
3. Verwendung von Widerständen mit den Werten R, R/2, R/4, R/8, R/16, R/32, …
4. Mittelwertbildung aus zwei Spannungswerten

Aufgabe 12.3
Ein ADU hat einen Aussteuerbereich $U_{\max}$ von 3 V und eine Wortbreite von $n = 10$ bit.

a) Wie groß ist die Quantisierungsintervallbreite Q?
b) Wie groß ist der höchste codierbare Spannungswert?
c) Welche Codierung ergibt sich für die Spannung 1,2 V?
d) Am Ausgang wird der Code 00 0100 1011 ausgegeben. In welchem Bereich ist der Spannungswert?

Aufgabe 12.4
Ein ADU im Wägeverfahren hat einen Aussteuerbereich U_{max} von 2 V und eine Wortbreite von $n = 8$ bit.

a) Wie groß ist die Quantisierungsintervallbreite Q?
b) Am Eingang liegt die Spannung 0,7 V an. Geben Sie die Schritte der AD-Umsetzung an. Welche Codierung ergibt sich für die Spannung?

Aufgabe 12.5
Ein Sigma-Delta-Umsetzer hat einen Messbereich von ±1 V und der Analogeingang U_x beträgt − 0,2 V.

a) Geben Sie den Zeitverlauf einer AD-Umsetzung an. Nehmen Sie an, dass im ersten Zeitschritt die Rückführung $U_{dig} = 0$ V ist und dass der Integrator mit der Spannung $U_{int} = 0$ V startet (Tab. 12.4).
b) Interpretieren Sie die Ausgabewerte.

Aufgabe 12.6
Ein PWM-Ausgang hat den in Abb. 12.38 dargestellten Zeitverlauf. Welche Ausgangsspannung wird durch das Signal erzeugt?

Tab. 12.4 Zeitablauf für Übungsaufgabe zum Sigma-Delta-Umsetzer

Zeitschritt	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
U_x [in V]	−0,2	−0,2	−0,2	−0,2	−0,2	−0,2	−0,2	−0,2	−0,2	−0,2	−0,2	−0,2	−0,2	−0,2	−0,2
U_{dig} [in V]	0														
U_{diff} [in V]															
U_{int} [in V]															
Plus [binär]															

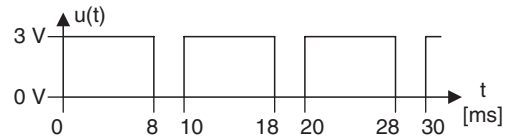


Abb. 12.38 Zeitablauf der Pulsweitenmodulation (PWM)

Grundlagen der Mikroprozessortechnik 13

Mikroprozessoren und Mikrocontroller sind komplexe, hochintegrierte digitale Schaltkreise, auf deren Basis leistungsfähige Rechnersysteme realisiert werden.

Einer der wichtigsten Meilensteine auf dem Weg zu modernen Rechnersystemen war die kommerzielle Realisierung des ersten integrierten Mikroprozessors. Zu Anfang der 1970er Jahre gelang dies der Firma Intel. Im Jahr 1977 schloss sich die Einführung von Mikrocontrollern an, die ein gesamtes Rechnersystem mit Prozessor, Speicher und weiteren Komponenten auf einem einzelnen Chip integrieren. In den folgenden Jahren setzte eine stürmische Entwicklung im Bereich der Mikroprozessortechnik ein, die insbesondere durch die fortschreitende Integrationsdichte digitaler Schaltkreise unterstützt wurde. Heute sind Rechnersysteme, die auf Mikroprozessoren oder Mikrocontrollern basieren, integraler Bestandteil des täglichen Lebens geworden. Sie werden als PCs realisiert und unterstützen den Anwender zum Beispiel bei der Büroarbeit oder dem Recherchieren im Internet. Auch in Mobiltelefonen, Digitalkameras, Geräten der Unterhaltungselektronik oder in Hausgeräten, Kraftfahrzeugen und industriellen Anlagen arbeitet eine Vielzahl von digitalen Rechnersystemen.

Im Rahmen dieses Kapitels werden die wesentlichen Grundlagen der Mikroprozessortechnik beschrieben. Sie bilden die Basis für das Verständnis von konkreten Rechnern, wie zum Beispiel der in Kapitel 14 vorgestellten Mikrocontroller.

13.1 Grundstruktur eines Mikrorechnersystems

Mikrorechner sind digitale Systeme, deren Aufgabe es ist, Daten zu verarbeiten. Diese Aufgabe zerfällt in drei grundlegende Schritte, die von einem Mikrorechnersystem ausgeführt werden müssen: Dateneingabe, Datenverarbeitung und Datenausgabe. Die Steuerung dieser Schritte wird durch ein Programm festgelegt, welches die Reihenfolge der benötigten Verarbeitungsschritte festlegt.

W. Gehrke et al., *Digitaltechnik,* Springer-Lehrbuch,
DOI 10.1007/978-3-662-49731-9_13

In der Regel müssen mehrere eingelesene Daten miteinander verknüpft werden, um einen Ausgabewert zu berechnen. Hieraus ergibt sich zwangsläufig, dass die Möglichkeit bestehen muss, Eingabewerte oder auch Zwischenergebnisse speichern zu können, die sich während der Verarbeitung ergeben.

Aus diesen grundlegenden Betrachtungen können die wesentlichen Komponenten eines Mikrorechnersystems abgeleitet werden: Es werden Eingabe- und Ausgabekomponenten, Speicher und mindestens eine Verarbeitungseinheit benötigt. Ein Rechnersystem auf Basis dieser Komponenten wurde in den 1940er Jahren von John von Neumann beschrieben und ist als sogenannte *Von-Neumann-Architektur* bekannt geworden.

Auch heutige Rechnersysteme, vom PC bis hin zu Mikrorechnersystemen, welche zum Beispiel die Steuerung Ihrer Waschmaschine übernehmen, können als eine spezielle Implementierung der Von-Neumann-Architektur aufgefasst werden (Abb. 13.1).

Eine Von-Neumann-Architektur besteht aus drei wesentlichen Komponenten: Die Ein-/Ausgabe-Einheit dient dem Datenaustausch mit externen Komponenten wie Tastaturen, Anzeigen oder auch Sensoren und Aktuatoren.

Die zentrale Verarbeitungseinheit (engl. *Central Processing Unit, CPU*) dient der eigentlichen Verarbeitung der Daten. Sie besteht aus einem Steuerwerk, einem Rechenwerk und Registern. Die zentrale Aufgabe des Steuerwerks ist die Interpretation der Befehle des auszuführenden Programms und die zugehörige Ablaufsteuerung innerhalb der CPU, während das Rechenwerk (engl. *Arithmetic Logical Unit, ALU*) logische und arithmetische Operationen ausführt. Die Operanden und Ergebnisse der Operationen werden in den CPU-internen Registern, die auch als *Arbeitsregister* bezeichnet werden, abgelegt.

Die dritte Komponente einer Von-Neumann-Architektur ist der Speicher, in welchem sowohl die Befehle des Programms als auch Daten abgelegt werden.

Für die Verbindung der einzelnen Komponenten eines Rechnersystems werden in der Regel sogenannte Busse eingesetzt. Busse können als die logische Zusammenfassung einzelner Signalleitungen verstanden werden, wobei diese Leitungen zusammengehörende Informationen übertragen. Die in einem Mikrorechnersystem verwendeten Busse können grob in drei verschiedene Typen eingeteilt werden:

Der *Adressbus* dient zur Auswahl einer Komponente mit der die CPU Daten austauschen möchte. Dies kann zum Beispiel eine Speicherstelle innerhalb des Speichers

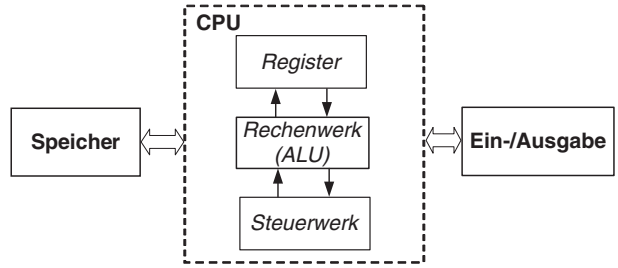


Abb. 13.1 Blockschaltbild eines Rechners auf Basis der Von-Neumann-Architektur

sein. Da die einzelnen Speicherstellen unterschiedliche Adressen besitzen, kann anhand der Adresse das angesprochene Speicherelement ausgewählt werden.

Die Daten selbst werden über den *Datenbus* übertragen. Die Daten können hierbei sowohl von der CPU an die Speicher- und Ein-/Ausgabe-Komponenten als auch von diesen Komponenten an die CPU gesendet werden.

Neben den Leitungen für den Datentransfer und den Adressleitungen wird ein *Steuerbus* benötigt. Der Steuerbus übermittelt alle Informationen, die neben Daten und Adressen an die einzelnen Komponenten des Systems übertragen werden müssen. Ein Beispiel für eine solche Information ist, ob von der ausgewählten Adresse Daten gelesen werden sollen, oder ob die Daten, die von der CPU auf den Datenbus gelegt worden sind, geschrieben werden sollen. Weiterhin kann der Steuerbus beispielsweise zur Übertragung eines Taktsignals zur Systemsynchronisation oder zur Übermittlung von Fehlercodes verwendet werden.

Ein exemplarisches Blockschaltbild eines einfachen Mikrorechnersystems mit einem unidirektionalen Adressbus und einem bidirektionalen Datenbus ist in Abb. 13.2 dargestellt. Dieses System besitzt zwei verschiedene Speicher. Ein Flashspeicher dient der Aufnahme von Daten, die auch nach dem Abschalten der Versorgungsspannung erhalten bleiben sollen. Wird in diesem Speicher das Programm abgelegt, steht es direkt nach dem Einschalten zur Verfügung und kann sofort ausgeführt werden. Darüber hinaus können im Flashspeicher Konstanten abgelegt werden, die für die Ausführung des Programms benötigt werden. Da sich die Variablen eines Programms während der Programmlaufzeit häufig ändern, ist es nicht sinnvoll, Variablen ebenfalls in einem Flashspeicher abzulegen, da das häufige Beschreiben des Flashspeichers eine frühe Alterung des Speichers nach sich ziehen würde. Daher ist in dem Beispielsystem ein RAM-Speicher vorgesehen, der zur Speicherung von Variablen verwendet wird.

Neben den Speichern besitzt das System Eingabe- und Ausgabekomponenten. Die CPU kann mit den Komponenten des Systems kommunizieren, indem die entsprechende Adresse

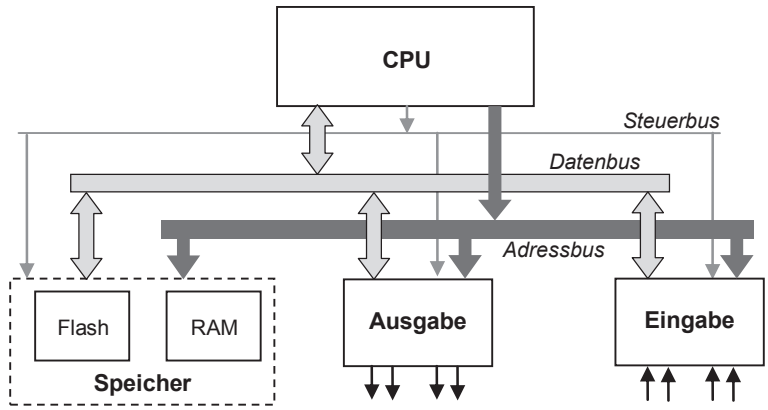


Abb. 13.2 Blockschaltbild eines einfachen Mikrorechnersystems

der jeweils ausgewählten Komponente auf den Adressbus gelegt wird. Da die Adressen von dem Programm, welches von der CPU ausgeführt wird, berechnet werden müssen, ist es für die Programmierung des Systems von wesentlicher Bedeutung, die Adressen zu kennen, unter denen die Systemkomponenten angesprochen werden. Diese Adressen werden häufig in grafischer Form als eine sogenannte *Address Map* zur Verfügung gestellt.

Eine mögliche Address Map für das gezeigte Beispielsystem ist in Abb. 13.3 links dargestellt. Die Auswahl zwischen Speicher und Ein-/Ausgabeeinheiten wird in diesem Fall nur durch die auf dem Adressbus anliegende Adresse durchgeführt. Adressen im Bereich von 0x0000 bis 0x5FFF adressieren die Speicherelemente des Systems, während mit Adressen im Bereich 0xC000 bis 0xFFFF auf Eingabe- und Ausgabekomponenten zugegriffen werden kann. Man spricht in diesem Fall auch davon, dass sich der Speicher und die Ein-/Ausgabeeinheiten „einen gemeinsamen Adressraum teilen“. Der Fachbegriff für diesen Ansatz lautet *Memory-Mapped-I/O*.

Als Alternative können auch getrennte Adressräume für Speicher und Ein-/Ausgabekomponenten verwendet werden. In diesem Fall spricht man von *Port Mapped I/O*. Eine mögliche Adressierung der Systemkomponenten des Beispielsystems ist in Abb. 13.3 rechts angegeben. Die Adressierung der Systemkomponenten erfolgt nun mithilfe der Adresse und der zusätzlichen Information, ob ein Speicherzugriff oder ein Zugriff auf die Ein-/Ausgabe erfolgen soll. Letztere wird den Komponenten mithilfe des Steuerbusses zur Verfügung gestellt.

Beide Ansätze werden für Rechnersysteme in der Praxis verwendet. Ein gemeinsamer Adressraum vereinfacht die Programmierung, was dem Programmierer oder rechnergestützten Werkzeugen wie Compilern zugute kommt. Auf der anderen Seite kann bei Verwendung von zwei getrennten Adressräumen unter Umständen eine Beschleunigung des Zugriffs auf Ein-/Ausgabekomponenten erfolgen, was sich positiv auf die Rechenzeit eines Programms auswirken kann.

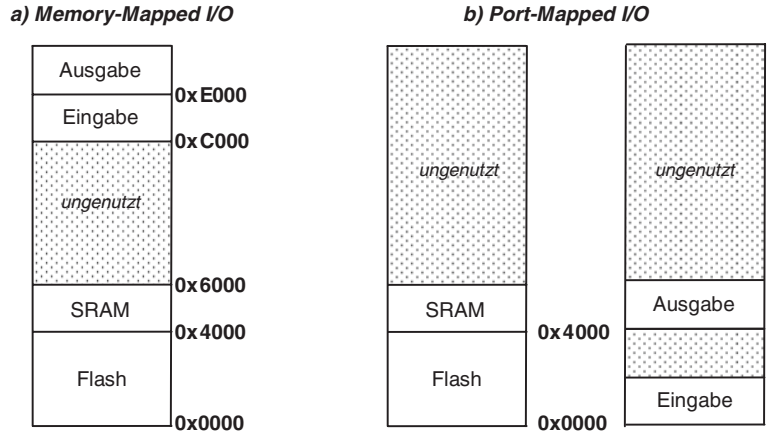


Abb. 13.3 Mögliche Address Maps für das Beispielsystem

13.2 Befehlsabarbeitung in einem Mikroprozessor

Die Befehle eines Programms werden in binärer Form im Programmspeicher abgelegt. Jeder Befehl enthält Informationen über die auszuführende Operation, die benötigten Operanden und wo Ergebnisse des Befehls abgespeichert werden sollen. Für die Abarbeitung eines Befehls ist es erforderlich, die binär codierten Befehle zunächst in der CPU zu decodieren. Im Anschluss hieran werden die benötigten Operanden dem Rechenwerk der CPU zugeführt. Das Rechenwerk führt dann die im Befehl codierte arithmetische oder logische Operation aus und speichert das Ergebnis ab. Somit sind fünf Schritte für die Ausführung eines Befehls erforderlich:

1. Befehl aus dem Programmspeicher holen und in die CPU übertragen
2. Befehl decodieren, die Operanden bestimmen und die auszuführende Operation aus dem Befehlswort extrahieren
3. Werte der Operanden bestimmen, zum Beispiel Werte aus dem Datenspeicher in die CPU übertragen
4. Operation mithilfe des Rechenwerks ausführen
5. Ergebnis der Operation abspeichern

Die Steuerung des Ablaufes dieser Schritte wird vom Steuerwerk der CPU übernommen. Grundsätzlich kann das Steuerwerk als ein endlicher Automat aufgefasst werden, der die benötigten Arbeitsschritte zur Ausführung eines Befehls sequenziell durchläuft. In der Praxis wird das Steuerwerk eines Mikroprozessors häufig nicht als ein einzelner Mealy- oder Moore-Automat realisiert, sondern besteht aus einer hierarchischen Anordnung mehrerer Automaten. Mikroprozessoren sind im Allgemeinen synchrone Systeme, deren interne Abläufe durch ein zentrales Taktsignal synchronisiert werden. Im einfachsten Fall werden die einzelnen Schritte der Befehlsabarbeitung in jeweils einem Taktzyklus ausgeführt. Somit würde die Abarbeitung eines einzelnen Befehls gemäß den oben dargestellten Schritten jeweils 5 Taktzyklen benötigen.

Bei realen Mikroprozessoren kann die Anzahl der benötigten Taktzyklen zur Ausführung sowohl von Prozessor zu Prozessor als auch für die einzelnen Befehle eines Prozessors unterschiedlich sein. Ein Grund für dieses Verhalten ist in den technologischen Randbedingungen zu suchen, die für die Herstellung eines Prozessors gelten. So kann beispielsweise ein Zugriff auf den Programmspeicher im Vergleich zu den anderen Verarbeitungsschritten deutlich mehr Zeit in Anspruch nehmen. In diesem Fall wäre es denkbar, dass der erste Schritt, also der Zugriff auf den Programmspeicher, in einem Taktzyklus ausgeführt wird, während die weiteren Schritte zusammengefasst in einem weiteren Taktzyklus durchgeführt werden. In diesem Fall würde die Abarbeitung eines Befehls also lediglich zwei Taktzyklen benötigen.

Darüber hinaus ist es denkbar, dass einzelne Befehle komplexere Verarbeitungsschritte benötigen als andere Befehle des gleichen Prozessors. Es kann sein, dass für die Übertragung der Operanden einzelner Befehle eine aufwendige Berechnung der

Speicheradresse erforderlich ist, für die mehrere Taktzyklen benötigt werden. Andere Befehle kommen dagegen mit weniger komplexen Berechnungen aus und werden in kürzerer Zeit ausgeführt.

13.3 Typische Befehlsklassen

Bei dem Entwurf eines Mikroprozessors kommt der Frage, welche Befehle zur Verfügung gestellt werden sollen, eine zentrale Bedeutung zu. Hierbei existieren viele Freiheitsgrade. So gibt es nicht einen ultimativen Satz von Befehlen, der von allen Prozessoren gleichermaßen unterstützt wird. Vielmehr besitzt jeder Prozessor einen eigenen Befehlssatz, der mit Rücksicht auf unterschiedliche Kriterien wie CPU-Kosten, Speicherbedarf für Programme, Rechenleistung etc. entworfen worden ist. Auch wenn der Befehlssatz eines Prozessors also nicht allgemeingültig angegeben werden kann, so lassen sich dennoch Gemeinsamkeiten der Befehlssätze erkennen.

Für einen typischen Prozessor können die Befehle in *Befehlsklassen* zusammengefasst werden, die im Folgenden kurz vorgestellt werden.

13.3.1 Aufbau eines Befehlswortes

Ein Programm besteht aus einzelnen Befehlsworten, die nacheinander ausgeführt werden. Mit jedem Befehlswort wird dem Prozessor mitgeteilt, welcher Teilschritt als nächstes auszuführen ist. Dies kann zum Beispiel eine arithmetische Operation oder auch der Sprung an eine andere Stelle im Programm sein. Das Befehlswort besteht aus einer definierten Anzahl von Nullen und Einsen, die vom Steuerwerk des Prozessors interpretiert werden. Sowohl die Wortbreite der einzelnen Befehle als auch die Bedeutung der in einem Befehlswort vorhandenen Bits können bei der Definition eines Instruktionssatzes frei gewählt werden.

Um die Decodierung eines Befehls durch das Steuerwerk zu vereinfachen, benutzen viele Prozessoren Befehlsworte, deren Bits zu Feldern zusammengefasst sind. Eines dieser Felder gibt dann zum Beispiel die auszuführende Operation (zum Beispiel „Addition“ oder „Sprung“) an. Die weiteren Bits stellen ergänzende Informationen zur Verfügung. So muss beispielsweise bei einem arithmetischen Befehl angegeben werden, aus welchen Arbeitsregistern die Operanden geholt werden sollen und in welchem Arbeitsregister das Ergebnis abgelegt werden muss.

Betrachten wir zur Verdeutlichung einen Prozessor, dessen Befehle 32 Bit umfassen, und schauen uns eine mögliche Codierung eines Additions- und eines Sprungbefehls an: Um eine ausreichend große Anzahl an unterschiedlichen Befehlen zu ermöglichen wird die auszuführende Operation mit 6 Bit codiert. Um beispielsweise 32 verschiedene Arbeitsregister auswählen zu können, werden pro Register 5 Bit benötigt. Für eine Addition müssen drei Arbeitsregister ausgewählt werden (zwei für die Summanden und eines

für die Aufnahme des Ergebnisses). Damit werden für diesen Befehl 21 Bit belegt. Die verbleibenden 11 Bit können einen beliebigen Wert besitzen.

Möchte man dagegen eine Addition mit einem Registerwert und einer Konstanten durchführen, wird diese Konstante häufig mit im Befehlswort abgelegt. Da hierbei ein Register weniger ausgewählt werden muss (ein Summand ist ja die Konstante), werden also für die Operation und die Registerauswahl 16 Bit benötigt und es verbleiben 16 Bit für die Konstante, die gegebenenfalls mithilfe einer Vorzeichenerweiterung (vgl. Kapitel 2) auch auf eine größere Wortbreite erweitert werden kann.

Bei einem Sprungbefehl ist es ausreichend die Operation *Sprung* mit 6 Bit zu kennzeichnen. Die verbleibenden 26 Bit geben dann das *Sprungziel* (die Adresse des nächsten Befehls) an.

In Abb. 13.4 ist ein möglicher Aufbau des Befehlswortes für die drei hier diskutierten Beispiele dargestellt.

13.3.2 Arithmetische und logische Befehle

Die Aufgabe eines Mikroprozessors besteht darin, Daten mithilfe von mathematischen Operationen zu verknüpfen. Für die meisten der hierzu benötigten Grundoperationen wird ein entsprechender Befehl zur Verfügung gestellt. Ein typischer Prozessor besitzt arithmetische Befehle, die zum Beispiel die Negierung eines Operanden und die

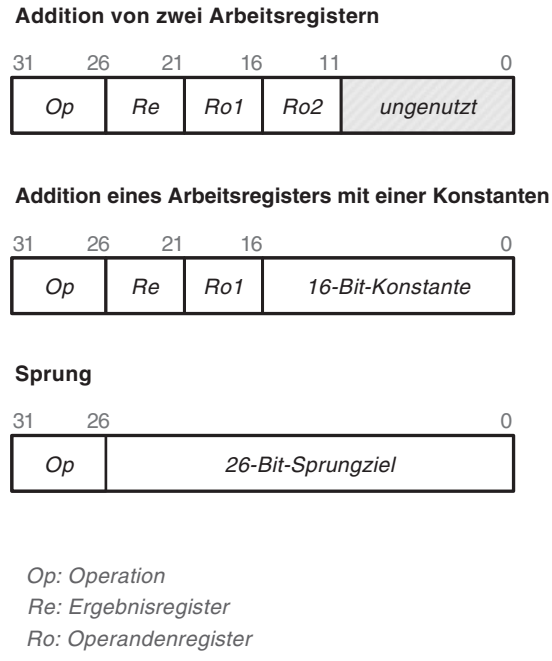


Abb. 13.4 Beispiele für den Aufbau eines 32-Bit-Befehlswortes

Addition oder Subtraktion zweier Operanden unterstützen. Darüber hinaus werden logische Befehle unterstützt, welche die bitweise UND-, ODER, und Exklusiv-ODER-Verknüpfung oder das bitweise Rechts- oder Links-Schieben durchführen.

Der Implementierungsaufwand eines Rechenwerkes für diese Operationen ist relativ gering. Daher werden diese Operationen von allen Prozessoren unterstützt. Ein Befehl zur Multiplikation oder Division erfordert dagegen einen höheren Aufwand für die Realisierung des Rechenwerks und ist daher nicht in allen CPUs enthalten. Fehlt die Hardwareunterstützung für eine arithmetische Operation, müssen diese Funktionen durch eine Folge von mehreren Befehlen, im Fall der Multiplikation beispielsweise durch Additions- und Schiebeoperationen, implementiert werden.

Ein weiterer wichtiger Faktor im Hinblick auf den Implementierungsaufwand des Rechenwerks ist die Wortbreite der Operationen. Einfache Prozessoren besitzen häufig Rechenwerke mit einer Wortbreite von 8 bit. Viele Prozessoren mit einer mittleren Rechenleistung verwenden in der Regel Rechenwerke mit einer Wortbreite von 32 bit. Hochleistungsprozessoren, wie sie zum Beispiel in PCs eingesetzt werden, besitzen dagegen Rechenwerke, welche die Verarbeitung von Operanden mit einer Wortbreite von 128 bit und mehr ermöglichen.

Werden in einem Programm häufig Gleitkommavariablen verwendet, ist es wünschenswert, dass die zugehörigen arithmetischen Grundoperationen mithilfe eines einzelnen Befehls ausgeführt werden können. Hierzu wird innerhalb des Rechenwerkes eine Einheit zur Ausführung von Operationen mit ganzzahligen Operanden (Integer-Unit) und eine Einheit zur Ausführung von Gleitkommaoperationen (Floating-Point-Unit) implementiert. Der hiermit verbundene Realisierungsaufwand ist bei vielen Prozessoren des unteren bis mittleren Kostenbereichs häufig nicht kommerziell sinnvoll. Aus diesem Grund werden Gleitkommaeinheiten in Mikroprozessoren dieses Segmentes in der Regel nicht eingesetzt. In diesem Fall müssen Gleitkommaoperationen durch eine Folge von Ganzzahloperationen realisiert werden, wodurch die Rechenzeit des Programms ansteigt.

13.3.3 Transferbefehle

Sollen zwei Daten, die im Speicher des Systems abgelegt sind, zum Beispiel durch Addition miteinander verknüpft werden, ist dies bei typischen Mikroprozessoren nicht mithilfe eines einzelnen Befehls durchführbar. Vielmehr muss zunächst ein Operand aus dem Speicher des Systems in einen Zwischenspeicher innerhalb der CPU kopiert werden. Im Anschluss daran kann mithilfe eines weiteren Befehls die eigentliche Addition der Daten erfolgen.

Daneben ist es häufig auch erforderlich, Daten zum Beispiel aus einer Eingabeeinheit in den Speicher des Systems zu kopieren, ohne die Daten hierbei zu modifizieren. Für beide Fälle stellen Prozessoren Datentransferbefehle zur Verfügung, mit denen Daten zwischen Speicher und CPU oder Eingabe- oder Ausgabeeinheiten und CPU ausgetauscht werden können. Die unterschiedlichen Befehle zum Kopieren von Daten können unter dem Begriff *Transferbefehle* zusammengefasst werden.

13.3.4 Befehle zur Programmablaufsteuerung

Ist ein arithmetischer Befehl oder ein Transferbefehl von der CPU ausgeführt worden, wird die Programmausführung mit dem nächsten im Programmspeicher abgelegten Befehl fortgesetzt. Die Möglichkeiten zum Erstellen von Programmen sind jedoch allein mit Transferbefehlen oder arithmetischen Befehlen sehr eingeschränkt. Selbst einfache Programme benötigen die Möglichkeit, Befehle wiederholt auszuführen (Schleifen) oder einzelne Programmteile unter bestimmten Bedingungen zu überspringen (bedingte Verzweigungen). Um diese Programmkonstrukte zu unterstützen, stellen Mikroprozessoren Befehle zur Steuerung des Programmablaufs zur Verfügung. Die zu dieser Gruppe zählenden Befehle umfassen:

Unbedingte Sprungbefehle
Nach Ausführung eines unbedingten Sprungbefehls wird die Ausführung des Programms an einer durch den Befehl spezifizierten Adresse im Programmspeicher fortgesetzt und es wird an eine andere Position im Programmspeicher „gesprungen“.

Bedingte Sprungbefehle
Bedingte Sprungbefehle führen, den Sprung nur aus, wenn eine im Befehl angegebene Bedingung erfüllt ist. Ist die Bedingung dagegen nicht erfüllt, wird das Programm mit dem nachfolgenden Befehl fortgesetzt.

Als Bedingungen können Informationen herangezogen werden, die sich aus der Ausführung vorangegangener Befehle ergeben. So kann zum Beispiel eine Programmverzweigung erfolgen, falls das Ergebnis der vorangegangenen Operation Null ist. Ebenso kann eine Verzweigung ausgeführt werden, falls das Ergebnis des zuvor ausgeführten Befehls negativ ist oder ein arithmetischer Überlauf aufgetreten ist.

Unterprogrammaufrufe
Nach dem Ende eines Unterprogramms muss zur aufrufenden Position im Programm zurückgekehrt werden. Die CPU muss beim Aufruf eines Unterprogramms also die aktuelle Befehlsadresse zwischenspeichern.

Ein Befehl zum Aufruf eines Unterprogramms besitzt daher die Funktionalität eines unbedingten Sprungs. Zusätzlich wird bei der Ausführung des Befehls die aktuelle Programmspeicheradresse gesichert. Auch für das Beenden eines Unterprogramms wird ein besonderer Befehl verwendet. Dieser Befehl sorgt dafür, dass das Programm an der beim Aufruf des Unterprogramms gespeicherten Programmspeicherposition fortgesetzt wird.

13.3.5 Spezialbefehle

Viele Mikroprozessoren stellen Befehle zur Verfügung, die nicht einer der zuvor diskutierten Befehlsklassen zugeordnet werden können. Ein Befehl dieser Klasse ist der

NOP-Befehl (engl. *no operation*), der lediglich einen Befehlszyklus ausführt, hierbei jedoch weder Daten transportiert noch Daten in irgendeiner Weise verändert. Dieser auf den ersten Blick wenig sinnvoll erscheinende Befehl kann zum Beispiel für die Realisierung einfacher Warteschleifen eingesetzt werden. Weiterhin besitzen viele Mikroprozessoren spezielle Befehle, die auf den jeweiligen Prozessor zugeschnitten sind und sich nicht in allen typischen Prozessoren wiederfinden lassen.

13.4 Adressierung von Daten und Befehlen

Für die Ausführung einer Operation mithilfe des Rechenwerks müssen zunächst die benötigten Operanden bestimmt werden. Dies bedeutet, dass der auszuführende Befehl Informationen darüber enthalten muss, ob ein Operand zum Beispiel im Datenspeicher des Systems zu finden ist und mit welcher Berechnungsvorschrift die Speicheradresse des Operanden aus den im Befehl enthaltenen Informationen bestimmt werden soll. Die von einem Mikroprozessor für die Adressierung zur Verfügung gestellten Berechnungsvorschriften werden in der Regel als *Adressierungsarten* bezeichnet. In diesem Abschnitt werden typische Adressierungsarten vorgestellt. Zur Vereinfachung bezieht sich die Darstellung auf den Zugriff der Operanden eines Befehls. Die hier vorgestellten Adressierungsarten können, mit Ausnahme der unmittelbaren Adressierung, ebenso für die Adressierung beim Abspeichern des Ergebnisses eines Befehls verwendet werden.

13.4.1 Unmittelbare Adressierung

Die einfachste Adressierungsart ist die unmittelbare Adressierung. In diesem Fall wird der Wert des zu verarbeitenden Operanden direkt als Teil des Befehls angegeben. Da der Wert des Operanden somit Teil des ausgeführten Programms ist und sich während der Programmlaufzeit nicht ändert, wird diese Adressierungsart häufig für Konstanten verwendet.

Abb. 13.5 verdeutlicht die unmittelbare Adressierung, bei dem sich der Operand direkt aus einem Teil des Befehlswortes ergibt. Das aus dem Programmspeicher gelesene Befehlswort ist hierbei abstrakt dargestellt. Insbesondere wurde auf die genauere Darstellung der für die Adressierung irrelevanten Teile des Befehlswortes, wie zum Beispiel die auszuführende Operation, verzichtet. Diese Teile des Befehlswortes sind dunkler dargestellt.

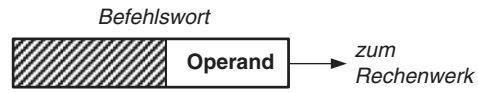


Abb. 13.5 Unmittelbare Adressierung

13.4.2 Absolute Adressierung

Im Fall der absoluten Adressierung ist ebenfalls eine Konstante im Befehlswort abgelegt. Diese wird jedoch anders als im Fall der unmittelbaren Adressierung nicht als Operand sondern als Adresse interpretiert.

Dementsprechend wird diese Konstante auf dem Adressbus ausgegeben. Der adressierte Wert wird aus dem Datenspeicher beziehungsweise einer Ein-/Ausgabekomponente ausgelesen und dem Rechenwerk als Operand zugeführt (Abb. 13.6).

13.4.3 Indirekte Adressierung

Die indirekte Adressierung kann als eine Erweiterung der absoluten Adressierung aufgefasst werden. Die im Befehlswort codierte Konstante wird ebenfalls als Registerauswahl interpretiert. Der in dem ausgewählten Register liegende Wert wird als Adresse verwendet wird.

In Abb. 13.7 ist das Grundprinzip der indirekten Adressierung dargestellt.

Die indirekte Adressierung kann auch mit einer Modifikation des verwendeten Registers kombiniert werden. Dies ist sinnvoll, wenn ein Prozessor auf mehrere aufeinanderfolgende Adressen zugreifen soll. In der Regel ist die Adressmodifikation auf das Inkrementieren (Erhöhung des Wertes um 1) und Dekrementieren (Verringern um 1) beschränkt. Da die Modifikation des Adressspeichers automatisch mit der Ausführung des zugehörigen Befehls stattfindet, spricht man auch von *indirekter Adressierung mit Auto-Inkrement beziehungsweise Auto-Dekrement.*

Bei der Ausführung eines Befehls, der die indirekte Adressierung mit Auto-Inkrement beziehungsweise -Dekrement verwendet, wird einerseits der Datenspeicher adressiert und andererseits ein Registerwert modifiziert. Die Reihenfolge dieser beiden Schritte

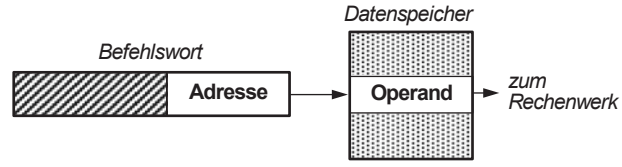


Abb. 13.6 Absolute (direkte) Adressierung

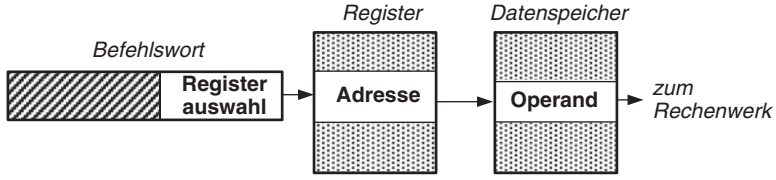


Abb. 13.7 Indirekte Adressierung

ist theoretisch beliebig wählbar. So könnte bei Verwendung eines Befehls mit Auto-Inkrement zunächst das Register inkrementiert werden. Der so erhaltene Wert könnte anschließend zur Adressierung des Operanden verwendet werden. Ebenso ist es denkbar, dass der aus dem Register ausgelesene Wert direkt zur Adressierung verwendet und erst anschließend inkrementiert wird. Der erste Fall wird als Pre-Inkrement, der zweite Fall als Post-Inkrement bezeichnet. Analog kann die indirekte Adressierung ebenso sowohl mit Pre-Dekrement als auch Post-Dekrement implementiert werden. Abb. 13.8 und 13.9 stellen die indirekte Adressierung mit Post-Inkrement und Pre-Dekrement schematisch dar.

Als eine weitere Variante der indirekten Adressierung setzen Mikroprozessoren vielfach die indirekte Adressierung mit Verschiebung ein. Bei Verwendung dieser Adressierungsart ergibt sich die Adresse des Operanden aus der Summe des aus dem Registerwert und eines Offsetwertes der als Konstante im Befehlswort abgelegt ist. Der so berechnete Wert wird lediglich zur Adressierung verwendet. Eine Veränderung des Adressspeichers, wie sie bei der indirekten Adressierung mit Auto-Inkrement beziehungsweise Auto-Dekrement erfolgt, findet hierbei nicht statt.

Darüber hinaus kann der Offset, der bei der indirekten Adressierung verwendet wird, auch in einem zur Laufzeit des Programms veränderbaren Indexspeicher abgelegt werden. In diesem Fall enthält das Befehlswort neben der Registerauswahl auch eine Adresse des Indexspeichers. Beide Speicher werden bei der Ausführung des Befehls ausgelesen. Die Summe der beiden ausgelesenen Werte ergibt die Adresse des Operanden, der dem Rechenwerk zugeführt wird. Diese Adressierungsart wird auch als indirekt indizierte Adressierung oder kurz *indizierte Adressierung* bezeichnet (Abb. 13.10).

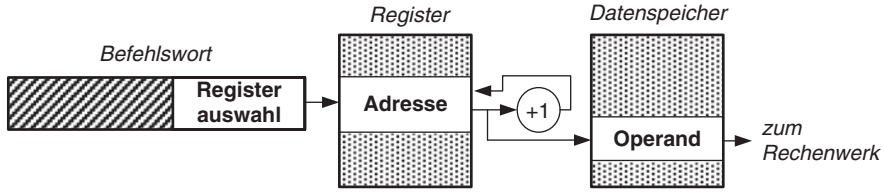


Abb. 13.8 Indirekte Adressierung mit Post-Inkrement

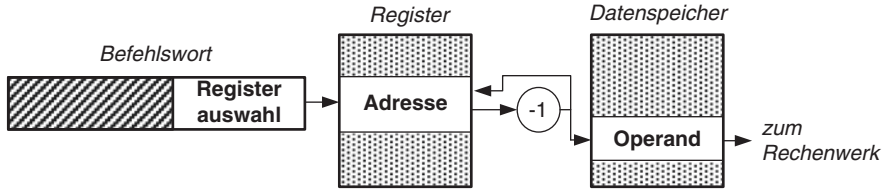


Abb. 13.9 Indirekte Adressierung mit Pre-Dekrement

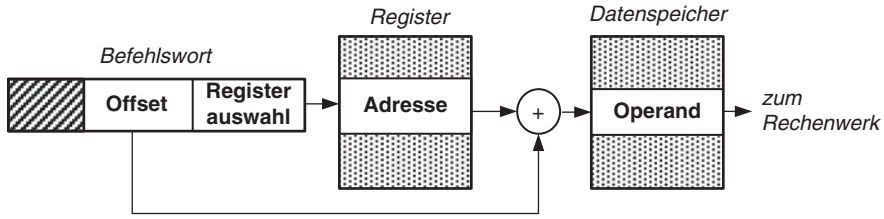


Abb. 13.10 Indirekte Adressierung mit Verschiebung

13.4.4 Indirekte Adressierung mit dem Stackpointer

Eine der wichtigsten Anwendungen der indirekten Adressierung ist die Realisierung eines Stapelspeichers (engl. *Stack*). Bei Verwendung eines Stapelspeichers ist die Adressierung der Daten eingeschränkt. Es gibt zu einem Zeitpunkt immer nur eine Position innerhalb des Speichers, die gelesen oder beschrieben werden kann. Diese Eigenschaft ist vergleichbar mit einem Papierstapel, auf dem nur ein neues Blatt oben auf dem Stapel abgelegt werden kann oder nur das oberste Blatt entfernt werden kann. Durch diese Analogie wird eine weitere wichtige Eigenschaft des Stapelspeichers deutlich: Bei einem Lesezugriff wird der jeweils zuletzt geschriebene Wert vom Stapelspeicher gelesen, genauso wie das zuletzt abgelegte Blatt als erstes von einem Papierstapel entfernt werden würde.

Diese Eigenschaft des Stapelspeichers lässt sich besonders gut für Unterprogrammaufrufe nutzen, bei denen die aktuelle Befehlsadresse zwischengespeichert werden muss. Wird bei dem Aufruf eines Unterprogramms die aktuelle Befehlsadresse auf einem Stapelspeicher abgelegt, sind auch Unterprogrammaufrufe innerhalb eines Unterprogramms einfach realisierbar. Beim Verlassen des zuletzt aufgerufenen Unterprogramms wird die zuletzt abgespeicherte Programmspeicheradresse vom Stapelspeicher entfernt, und die Programmausführung wird mit dem aufrufenden Unterprogramm fortgesetzt. Die Verschachtelungstiefe von Unterprogrammen ist somit lediglich durch die maximale Größe des Stapelspeichers begrenzt.

Die Funktion eines Stacks lässt sich auf verschiedene Weisen realisieren. Für einen typischen Prozessor wird meist eine Variante bevorzugt, bei der die auf dem Stack abgespeicherten Werte im Datenspeicher abgelegt werden. Darüber hinaus wird die aktuelle Schreib-/Leseposition in einem besonderen Register der CPU, dem Stapelzeiger (engl. *Stackpointer*), abgelegt.

Üblicherweise verweist der Stackpointer auf die Speicherstelle, die beim nächsten Schreibzugriff überschrieben wird. Ein Schreibzugriff führt darüber hinaus zum Dekrementieren des Stackpointers. Wiederholte Schreibzugriffe würden also zum Beschreiben des Datenspeichers an niedrigeren Adressen führen. Dieses Verhalten wird häufig auch mit der Aussage „der Stack wächst nach unten“ umschrieben. Für die Implementierung

eines Stapelspeichers mithilfe eines Stapelzeigers kann die indirekte Adressierung mit Auto-Inkrement beziehungsweise Auto-Dekrement eingesetzt werden. Ein Schreibzugriff erfolgt dann mithilfe einer indirekten Adressierung mit Post-Dekrement, während ein Lesezugriff die indirekte Adressierung mit Pre-Inkrement verwendet.

13.4.5 Befehlsadressierung

Für die Adressierung der abzuarbeitenden Befehle verwendet ein Mikroprozessor ein besonderes Register, den sogenannten Programmzähler *(Program Counter, PC)*. Der PC wird vom Steuerwerk der CPU normalerweise mit der Abarbeitung eines Befehls inkrementiert, sodass automatisch der jeweils nachfolgende Befehl im Programmspeicher adressiert wird. Wird dagegen ein Sprungbefehl ausgeführt, muss die Adressierung des Programmspeichers entsprechend modifiziert werden. Hierzu werden von den meisten Mikroprozessoren eine absolute Adressierung, eine relative Adressierung und eine indirekte Adressierung zur Verfügung gestellt. Zur Unterscheidung zwischen Datenadressierung und Befehlsadressierung werden diese Adressierungsarten auch als *PC-absolute, PC-relative* oder *PC-indirekte* Adressierung bezeichnet.

Im Fall der absoluten Adressierung wird der Programmzähler mit einer im Sprungbefehl angegebenen Konstanten geladen. Die Programmausführung wird somit an der Position fortgesetzt, die durch die Konstante festgelegt ist.

Die relative Adressierung verwendet ebenfalls eine im Befehlswort abgelegte Konstante. Die Summe aus dieser Konstanten und dem aktuellen PC ergibt den neuen Programmzähler. Während die absolute Adressierung also einen Befehl ausführt, der sich mit „springe zu Programmspeicheradresse XYZ“ umschreiben lässt, führt die PC-relative Adressierung einen Befehl aus, der mit „springe um XYZ Programmspeicheradressen“ beschrieben werden kann.

Im Fall der PC-indirekten Adressierung wird der neue Wert des PCs, ähnlich der indirekten Datenadressierung, aus einem Adressspeicher ausgelesen und in den Programmzähler übertragen. Die auszulesende Position des Adressspeichers wird hierbei als Konstante im Befehlswort angegeben.

13.5 Maßnahmen zur Steigerung der Rechenleistung

Die Aufgabe eines Mikroprozessors ist es, eine möglichst hohe Rechenleistung unter gegebenen Randbedingungen (Kosten, Verlustleistung, usw.) zur Verfügung zu stellen. In den folgenden Abschnitten werden technische Möglichkeiten aufgezeigt, die zu einer Steigerung der Rechenleistung von Mikroprozessoren eingesetzt werden können.

13.5.1 Erhöhung der Taktfrequenz

Da Mikroprozessoren als synchrone Systeme realisiert werden, ist es ein naheliegender Ansatz, die Taktfrequenz des Systems zu erhöhen. Mit der Erhöhung der Taktfrequenz lässt sich eine annähernd proportionale Steigerung der Rechenleistung erzielen.

Es muss jedoch berücksichtigt werden, dass die Möglichkeit zur Erhöhung der Taktfrequenz für einen Mikroprozessor begrenzt ist. Wird die Dauer eines Taktzyklus über eine kritische Grenze hinaus verringert, können Fehlfunktionen auftreten. Diese kritische Grenze ergibt sich aus dem kritischen Pfad, also der maximal auftretenden Signallaufzeit zwischen zwei Flip-Flops des Systems. Eine Möglichkeit, diese Signallaufzeit zu verringern, stellt das sogenannte Pipelining dar, welches in Abschn. 13.5.3 für Mikroprozessoren erläutert wird. Darüber hinaus ist zu beachten, dass bei Verwendung von CMOS-Technologien, wie sie heute für die Realisierung von Mikroprozessoren verwendet werden, die dynamische Verlustleistung proportional zur Taktfrequenz ansteigt. Dieser Effekt kann ebenfalls zu einer Limitierung der maximal verwendbaren Taktfrequenz führen.

13.5.2 Parallelität

Eine Erhöhung der Rechenleistung kann auch erzielt werden, indem mehrere Operationen gleichzeitig ausgeführt werden. Dies kann sowohl durch parallele Einheiten im Rechenwerk als auch durch die Verwendung mehrerer Mikroprozessoren ermöglicht werden.

Im Idealfall steigt die verfügbare Rechenleistung proportional zu der im Rechenwerk implementierten Parallelität. In der Praxis wird dieser theoretische Anstieg meist nicht erreicht. Programme bilden in der Regel sequenzielle Verarbeitungsschritte ab. Inwieweit diese Verarbeitungsschritte, entgegen der vom Programmierer vorgegebenen sequenziellen Abarbeitungsreihenfolge, auch zeitgleich ausgeführt werden können, ist sehr stark vom Programm abhängig. Im ungünstigsten Fall muss für jede Operation die jeweils vorangegangene Operation abgearbeitet werden, da zum Beispiel das Ergebnis der ersten Operation als Operand für den nachfolgenden Befehl benötigt wird. In diesem Fall kann die Parallelität des Rechenwerks nicht ausgenutzt werden und es wäre keine Erhöhung der Rechenleistung erreichbar.

Geht man davon aus, dass ein Programm aus ideal parallelisierbaren (die benötigte Rechenzeit verhält sich annähernd umgekehrt proportional zur eingesetzten Parallelität) und nicht-parallelisierbaren Anteilen besteht, kann der Rechenleistungsgewinn durch die folgenden Formel angeben werden:

$$G = \frac{1}{(s + p/N)}$$

mit: G – Rechenleistungsgewinn (engl. *Speedup*)

p – Durch Parallelverarbeitung beschleunigter Programmanteil
s – Anteil des Programms mit konstanter Rechenzeit
N – Parallelität des Systems, zum Beispiel Anzahl paralleler Operationen

Die Grundlagen zu dieser Betrachtung wurden erstmals von Gene M. Amdahl formuliert und sind als *Amdahl's Law* in die Geschichte der Computerwissenschaft eingegangen. Auch wenn diese Betrachtung starke Vereinfachungen vornimmt, macht sie dennoch deutlich, dass bereits ein geringer Anteil an nicht-parallelisierbaren Programmteilen zu einer signifikanten Begrenzung des realisierbaren Rechenleistungsgewinns führen kann.

Darüber hinaus erfordert der sinnvolle Einsatz paralleler Einheiten, dass diese mit den jeweils zu verarbeitenden Daten versorgt werden. Hierzu wird häufig ein hoher schaltungstechnischer Aufwand benötigt, der zusätzlich zu dem Aufwand der benötigten parallelen Einheiten erforderlich wird.

Darüber hinaus müssen in den Befehlsworten des Prozessors entweder mehrere Operationen codiert werden oder es müssen mehrere Befehle gleichzeitig verarbeitet werden können, was zu einer weiteren Erhöhung des Realisierungsaufwands führt. Diese Ansätze werden als *Very-Long-Instruction-Word-Architekturen (VLIW)* beziehungsweise *superskalare Architekturen* bezeichnet

Parallele Rechenwerke werden im Bereich der PC-Prozessoren eingesetzt, mit separaten Rechenwerken für Integer- und Floating-Point-Operationen. Bei PC-Prozessoren haben sich Multi-Core-Systeme durchgesetzt, bei denen mehrere Prozessoren in einem Gehäuse integriert werden. Diese Form der Rechenleistungserhöhung wurde notwendig, da sich die zuvor verfolgte Strategie einer mit jeder Prozessorgeneration steigenden Taktfrequenz aus technologischen Gründen nicht mehr durchhalten ließ.

13.5.3 Pipelining

Eine weitere Möglichkeit zur Erhöhung der Rechenleistung ist der Einsatz von Pipelining, welches im deutschen Sprachraum auch häufig mit Fließbandverarbeitung übersetzt wird.

Das Grundprinzip der Fließbandverarbeitung in der industriellen Produktion ist, dass an verschiedenen Stationen spezialisierte Teilaufgaben durchgeführt werden. Nach Durchlaufen aller Stationen ist das Endprodukt fertiggestellt. Da hierbei immer mehrere Stationen gleichzeitig aktiv sind, kann die Fließbandverarbeitung auch als eine besondere Form der Parallelverarbeitung aufgefasst werden. Der Unterschied zu der im vorangegangenen Abschnitt beschriebenen Form der Parallelverarbeitung ist jedoch, dass im Fall des Pipelinings jede Station nur einen ausgewählten Teil der gesamten Verarbeitungsaufgabe ausführt und das so erhaltene Arbeitsergebnis an die nachfolgende Station weiterreicht. Dieses Grundprinzip wird in Mikroprozessoren bei Befehlsabarbeitung eingesetzt.

In Abschn. 13.2 wurden die einzelnen Schritte zur Verarbeitung eines Befehls exemplarisch vorgestellt. Hierbei wurde die Verarbeitung eines Befehls durch die Ausführung

von 5 Teilschritten vorgenommen. Ohne Einsatz von Pipelining würden alle Teilschritte eines Befehls durchlaufen bevor die Ausführung des nachfolgenden Befehls begonnen wird. Nimmt man vereinfachend an, dass alle Teilschritte eine identische Verarbeitungszeit T_S benötigen, würde die Bearbeitung eines Befehls also $5T_S$ erfordern.

Wird dagegen jeder Teilschritt durch eine eigenständige Einheit ausgeführt, kann jede dieser Einheiten nach Bearbeitung eines Teilschritts sofort mit der Ausführung des nachfolgenden Befehls beginnen. Im Idealfall besitzen alle Verarbeitungsschritte identische Verzögerungszeiten. Dann kann bereits nach der Zeit T_S die Verarbeitung eines neuen Befehls mit dem ersten Teilschritt beginnen kann, während für den vorangegangenen Befehl zeitgleich der zweite Teilschritt ausgeführt wird.

In Abb. 13.11 ist der zeitliche Verlauf der Verarbeitung von Befehlen ohne und mit Einsatz von Pipelining dargestellt. Zum Zeitpunkt $t = 0$ beginnt in beiden Fällen die Ausführung des ersten Befehls. Wird Pipelining verwendet, kann bereits zum Zeitpunkt $t = T_S$ mit der Ausführung eines weiteren Befehls begonnen werden. Zum Zeitpunkt $t = 5T_S$ ist für beide Fälle die erste Instruktion komplett abgearbeitet. Bei Verwendung von Pipelining ist zu diesem Zeitpunkt bereits die Verarbeitung von vier weiteren Befehlen begonnen worden, während ohne Einsatz von Pipelining erst die Ausführung des zweiten Befehls begonnen wird. Betrachtet man einen längeren Zeitraum, lässt sich beobachten, dass bei Verwendung von Pipelining 5-mal mehr Instruktionen pro Zeiteinheit verarbeitet werden. Die Rechenleistung wird also um den Faktor 5 gesteigert.

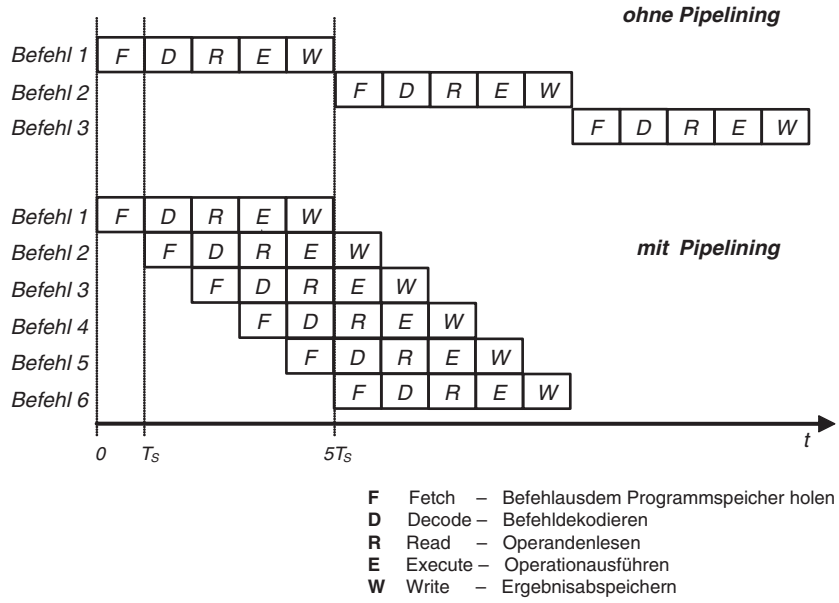


Abb. 13.11 Zeitlicher Verlauf der Befehlsverarbeitung mit und ohne Pipelining

Der schaltungstechnische Aufwand zur Realisierung einer einfachen Befehlspipeline ist moderat. Im einfachsten Fall ist es ausreichend die einzelnen Stufen der Befehlsausführung durch Flip-Flops (sog. Pipeline-Register) zu entkoppeln. Auf diese Weise kann in jedem Taktzyklus die Ausführung eines neuen Befehls gestartet werden.

Auf den ersten Blick mag der Einsatz von Pipelining als ein sehr effizientes Mittel zur Steigerung der Rechenleistung erscheinen. In der Tat setzen die meisten der heute verfügbaren Prozessoren Pipelining ein. Dennoch wird in der Praxis meist nicht ein zu der Anzahl der Pipelinestufen proportionaler Rechenleistungsgewinn erzielt. Der ausschlaggebende Grund für diesen Effekt ist das Bestehen von Abhängigkeiten zwischen den Befehlen, die zeitgleich verarbeitet werden. Exemplarisch soll dies im Folgenden anhand der Datenabhängigkeit zweier Instruktionen verdeutlicht werden: Wird ein Befehl ausgeführt, der als Operanden das Ergebnis des vorangegangenen Befehls benötigt, führt dies zu einem Konflikt. Erst wenn die vorangegangene Instruktion die W-Stufe durchlaufen hat, kann das Ergebnis von der R-Stufe als Operand für einen nachfolgenden Befehl gelesen werden. Betrachtet man zwei aufeinanderfolgende Befehle, wird deutlich, dass der zweite Befehl die R-Stufe bereits durchlaufen hat, wenn sich der erste Befehl in der W-Stufe befindet (vgl. Abb. 13.12). Ohne weitere Maßnahmen zu ergreifen, würde der zweite Befehl somit einen veralteten, falschen Wert als Operanden einlesen.

Die einfachste Möglichkeit diesen Konflikt aufzulösen, besteht darin, die Ausführung des zweiten Befehls zu verzögern. So wird sichergestellt, dass der zweite Befehl die R-Stufe erst durchläuft nachdem der erste Befehl in der W-Stufe verarbeitet wurde (vgl. Abb. 13.13). Diese Verzögerung der Befehlsausführung führt jedoch zu einer Verringerung der pro Zeiteinheit verarbeiteten Befehle, was somit zu einer Verringerung der Rechenleistung führt. Moderne Prozessoren setzen daher verschiedene komplexe Maßnahmen zur Verringerung des negativen Einflusses der Abhängigkeit zwischen

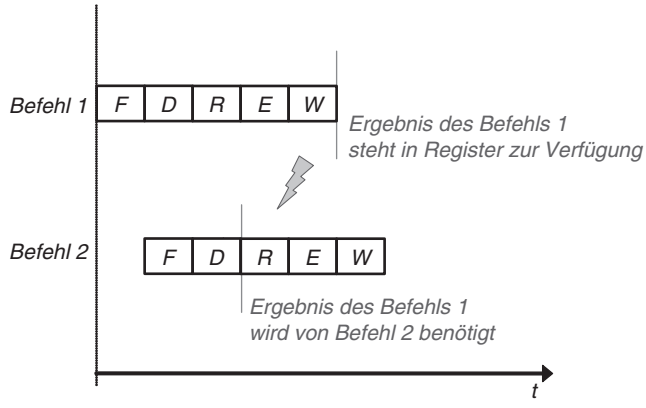


Abb. 13.12 Beispiel eines Konfliktes bei der Befehlsabarbeitung

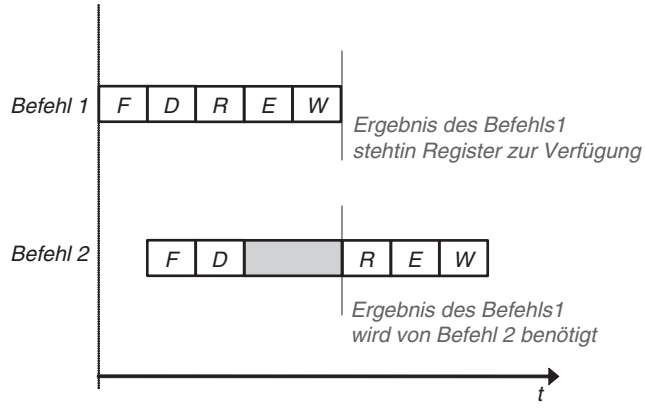


Abb. 13.13 Auflösung eines Pipelinekonfliktes durch Verzögerung der Befehlsabarbeitung

aufeinanderfolgenden Befehlen ein. Dennoch kann auch durch diese Maßnahmen keine völlige Elimination der Rechenleistungsverringerung erzielt werden, sodass auch in diesen Fällen die real erzielbare Rechenleistung unterhalb des theoretisch ermittelten Wertes ohne Berücksichtigung von Befehlsabhängigkeiten bleibt.

13.5.4 Befehlssatzerweiterungen

Für die Frage, ob eine bestimmte Aufgabenstellung von einem bestimmten Mikroprozessor bearbeitet werden kann, ist die Wahl des Befehlssatzes dieses Prozessors relativ unbedeutend. Stellt sich jedoch die Frage nach der Rechenleistung des Prozessors, kommt der Wahl des Befehlssatzes dagegen eine zentrale Bedeutung zu. Bereits in Abschn. 13.3.2 wurde am Beispiel der Multiplikation verdeutlicht, dass die Verwendung eines Multiplikationsbefehls die Rechenleistung eines Prozessors erhöhen kann. Entsprechendes gilt für den Einsatz einer Floating-Point-Unit zur Beschleunigung von Gleitkommaoperationen. Durch den Einsatz einer Gleitkommaeinheit können Fließkommaberechnungen um ein bis zwei Größenordnungen schneller durchgeführt werden.

Das Prinzip, den Befehlssatz auf das Anwendungsgebiet zu optimieren, muss nicht auf grundlegende Operationen wie Multiplikation, Division oder Gleitkommaoperationen beschränkt werden. Viele Mikroprozessoren stellen sogenannte Befehlssatzerweiterungen zur Verfügung. So wurde beispielsweise Mitte der 1990er Jahre die MMX-Befehlssatzerweiterung von der Firma Intel für PC-Prozessoren eingeführt. Eines der Ziele war es, durch diese Erweiterung eine flüssige Wiedergabe von Videosequenzen zu erreichen. In den darauffolgenden Jahren wurden die Erweiterungen des Befehlssatzes unter dem Namen SSE *(Streaming SIMD Extensions)* fortgeführt.

Andere Prozessorhersteller haben ebenso verschiedenste Befehlssatzerweiterungen entwickelt und mit unterschiedlichen Bezeichnungen auf dem Markt etabliert.

Spezielle Befehle zur Unterstützung typischer Operationsfolgen lassen sich nicht nur in PC-Prozessoren finden. Selbst Mikroprozessoren der unteren Preisklasse setzen das Prinzip der Befehlssatzerweiterung ein. Die im nachfolgenden Kapitel vorgestellte AVR-CPU besitzt beispielsweise besondere Befehle zum Setzen oder Löschen einzelner Bits.

13.6 Grundlegende Mikroprozessorarchitekturen

Für den Entwurf und die Auswahl eines Mikroprozessors stellen sich viele Fragen, die die Architektur des Prozessors beeinflussen. Einige dieser Fragestellungen sind:

- Welche Wortbreite wird für die Daten- und Adressbusse verwendet?
- Welche Wortbreite besitzt das Rechenwerk, und ist eine Floating-Point-Unit zur Beschleunigung von Gleitkommaoperationen vorhanden?
- Welche Befehle werden unterstützt?
- Wie werden die Befehle binär codiert und welche Wortbreite wird für die Codierung der Befehle verwendet?
- In welchem Umfang sind innerhalb der CPU Speicherelemente, zum Beispiel zum Abspeichern von Zwischenergebnissen vorhanden?
- In welchen Teilschritten werden die Befehle abgearbeitet?
- In welchem Umfang wird Pipelining für die Befehlsausführung eingesetzt?
- Wie werden Parameter wie Rechenleistung, Kosten und Verlustleistung ausbalanciert?
- Welche Halbleitertechnologie wird für die Realisierung verwendet?

Anhand dieser Auswahl von Fragestellungen wird deutlich, dass für den Entwurf eines Mikroprozessors eine Vielzahl von Freiheitsgraden existiert, die zu unterschiedlichen architektonischen Varianten führt. Trotz dieser Detailvielfalt können Mikroprozessoren in zwei grundlegende Architekturklassen eingeteilt werden, deren Eigenschaften in den folgenden Abschnitten näher beleuchtet werden.

13.6.1 CISC

Die Abkürzung *CISC* steht für *Complex Instruction Set Computer* und bezeichnet Prozessoren, bei denen angestrebt wird, Befehle mit einer möglichst großen Funktionalität zur Verfügung zu stellen.

CISC-Prozessoren zeichnen sich durch einen großen Befehlsumfang und eine große Anzahl unterschiedlicher Adressierungsarten aus. Die Wortbreite der einzelnen Befehle eines CISC-Prozessors variiert, sodass für die Ausführung der Befehle eine unterschiedliche Anzahl von Programmspeicherzugriffen erforderlich ist. Diese

Eigenschaft, sowie die unterschiedliche Komplexität der Befehle, führen dazu, dass die Abarbeitung eines Befehls in der Regel mehrere Taktzyklen erfordert. Die Anzahl der benötigten Taktzyklen variiert bei typischen CISC-Prozessoren zudem in Abhängigkeit vom Befehl. Typische Beispiele für CISC-Architekturen sind die Prozessorfamilien 808x und 80x86 der Firma Intel oder die Prozessoren der 680x0-Serie der Firma Motorola.

CISC-Prozessoren wurden bis in die 1990er Jahre erfolgreich vermarktet. Durch die Fortschritte der Halbleitertechnologie wurden höhere Integrationsdichten und kürzere Verzögerungszeiten der verwendeten Logik- und Speicherelemente ermöglicht. Insbesondere durch die sinkende Zugriffszeit der Speicher war es nicht mehr nötig mit einem Befehl möglichst viele Funktionen auszuführen. Dies brachte einen der Hauptgründe für die Verwendung von CISC-Prozessoren ins Wanken und führte dazu, dass die Bedeutung der CISC-Prozessoren abnahm.

13.6.2 RISC

Im Lauf der 1980er Jahre wurden zahlreiche Studien zu Architekturen von Mikroprozessoren durchgeführt, die unter anderem zeigten, dass viele der komplexen Befehle eines CISC-Prozessors nur zu einem geringen Anteil in praktischen Programmen verwendet wurden. Die meisten Programme nutzen nur einen kleinen Anteil des Befehlssatzes, vorrangig die einfach strukturierten Befehle des Prozessors. Diese Beobachtung führte zu einem Architekturansatz, der als *RISC (Reduced Instruction Set Computer)* bezeichnet wird. Typische RISC-Prozessoren zeichnen sich durch die folgenden Eigenschaften aus:

Limitierter Befehlssatz
Es werden nur die am häufigsten benötigten Grundbefehle implementiert, wobei auf komplexe Adressierungsarten verzichtet wird. Dies ist sowohl für den Aufwand als auch im Hinblick auf die Taktfrequenz von Vorteil.

Instruktionspipelining
Durch die Reduktion des Befehlssatzes wird gleichzeitig der Einsatz von Instruktionspipelining vereinfacht. Hierbei wird angestrebt, in jedem Taktzyklus des Prozessors die Bearbeitung eines neuen Befehls zu beginnen.

Load/Store-Architektur
Zum Austausch von Daten mit dem Speicher oder Ein-/Ausgabekomponenten werden Befehle eingesetzt, die nur einen Transport der Daten zwischen Speicher und den Arbeitsregistern der CPU durchführen *(load, store)*. Auf die Möglichkeit, innerhalb eines Befehls sowohl den Datentransport als auch eine arithmetisch-logische Operation auszuführen, wird im Gegensatz zu typischen CISC-Prozessoren, verzichtet.

Relativ hohe Registeranzahl
RISC-Prozessoren besitzen meist deutlich mehr Register als CISC-Prozessoren. Die während der Abarbeitung eines Programms anfallenden Zwischenergebnisse können so innerhalb des Prozessors abgelegt werden. Die Anzahl für zusätzliche Befehle zum Ablegen der Zwischenergebnisse im Datenspeicher kann auf diese Weise reduziert werden.

Universell verwendbare Register
Die CPU-internen Register können sowohl für die Verarbeitung von Daten als auch zur Berechnung von Adressen verwendet werden. Eine Unterscheidung zwischen Daten- und Adressregistern, wie sie teilweise bei CISC-Prozessoren verwendet wurde, findet nicht statt.

Einfache Befehlscodierung
Um die Decodierung eines Befehls zu vereinfachen und damit zu beschleunigen, wird eine einheitliche Codierung der Befehle angestrebt. Hierbei wird das Befehlswort in der Regel in einzelne Felder unterteilt, in denen unabhängig vom Befehl, immer die gleiche Information (zum Beispiel die auszuführende Operation oder die für die Operation zu verwendenden Register) gespeichert ist.

13.6.3 RISC und Harvard-Architektur

Wie im vorigen Abschnitt beschrieben, ist eine wesentliche Eigenschaft von RISC-Prozessoren die Verwendung von Instruktionspipelining zur Verarbeitung von Befehlen. Der Einsatz von Instruktionspipelining ermöglicht eine Erhöhung des Befehlsdurchsatzes (Anzahl der verarbeiteten Befehle pro Taktzyklus), da in jedem Taktzyklus mehrere unterschiedliche Befehle in den einzelnen Stufen der Pipeline verarbeitet werden. Wird ein RISC-Prozessor auf Basis einer Von-Neumann-Architektur implementiert, ergibt sich ein Engpass, durch die Verwendung eines gemeinsamen Speichers für Befehle und Daten.

Dieser Engpass entsteht, da bei Verwendung von Instruktionspipelining in jedem Taktzyklus die Ausführung eines neuen Befehls gestartet werden kann. Dabei wird mit jedem Taktzyklus ein Zugriff auf den Speicher ausgeführt. Werden Befehle ausgeführt, die einen Zugriff auf den Datenspeicher ausführen, führt dies zu einem Konflikt: Innerhalb eines Taktzyklus müsste sowohl der Zugriff auf die Befehle des Programms als auch der Zugriff auf die im gemeinsamen Programm- und Datenspeicher abgelegten Daten erfolgen. Der gemeinsame Speicher für Daten und Befehle einer Von-Neumann-Architektur ermöglicht jedoch nur einen Zugriff, entweder auf Daten oder auf Befehle. Somit müssen die Zugriffe auf Daten und Befehle in unterschiedlichen Taktzyklen erfolgen. Es kann also nicht mehr in jedem Taktzyklus ein Zugriff auf die Befehle des Programms erfolgen und der Befehlsdurchsatz sowie die erzielbare Rechenleistung

werden reduziert. Der beschriebene Engpass der Von-Neumann-Architektur wird auch als *„Von-Neumann-Bottleneck“* bezeichnet.

Es ist möglich, einen RISC-Prozessor auf Basis einer Von-Neumann-Architektur zu realisieren, sofern die beschriebene Reduktion der Rechenleistung für das Anwendungsgebiet des Prozessors tolerierbar ist. Ist es dagegen das Ziel, einen möglichst hohen Befehlsdurchsatz zu erzielen, ist es sinnvoll, den Speicherkonflikt durch Realisierung getrennter Speicher für Befehle und Daten aufzulösen. Dieser architektonische Ansatz wird als *Harvard-Architektur* bezeichnet. Die Struktur eines Mikrorechnersystems auf Basis einer Harvard-Architektur ist in Abb. 13.14 dargestellt. Der Programmspeicher der in Abb. 13.14 Architektur kann beispielsweise als nichtflüchtiger Flashspeicher realisiert werden. Der Datenspeicher wird dagegen meist auf Basis eines flüchtigen SRAMs realisiert.

In der Regel benötigen Programme Konstanten, die beim Start des Programms definierte Werte enthalten. Einerseits handelt es sich bei diesen Konstanten um Daten, die somit im flüchtigen Datenspeicher abgelegt werden müssen, der jedoch nach dem Einschalten der Versorgungsspannung keine definierten Werte enthält. Daher werden die Konstanten zusammen mit dem Programm im Flashspeicher abgelegt und stehen sofort nach dem Einschalten des Systems zur Verfügung. Zu Beginn des Programms werden die Konstanten aus dem Flashspeicher in den Datenspeicher kopiert. Für diesen initialen Kopiervorgang muss der Programmspeicher jedoch wie ein Datenspeicher betrieben werden. Da dies nicht dem reinen Grundkonzept einer Harvard-Architektur entspricht, werden Architekturen mit getrennten Daten- und Programmspeichern, die einen

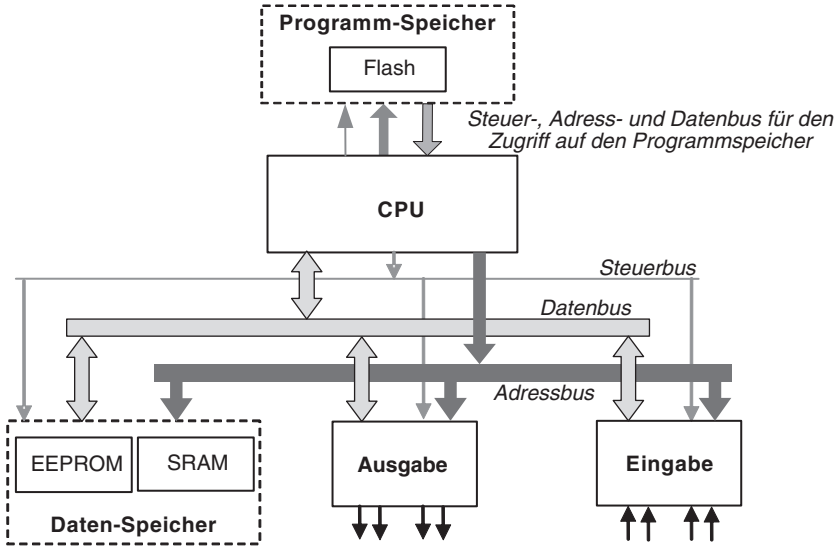


Abb. 13.14 Struktur eines Mikrorechners auf Basis einer Harvard-Architektur

datenorientierten Zugriff auf den Programmspeicher erlauben, auch als *modifizierte Harvard-Architektur* bezeichnet.

Die Idee, Daten und Befehle in getrennten Speichern abzulegen, um eine möglichst hohe Rechenleistung zu erzielen, mag einfach erscheinen. Allerdings wurde dieser Ansatz von ersten integrierten Mikrorechnersystemen nicht verwendet. Bei typischen CISC-Prozessoren, wie sie insbesondere in den 1970er bis 1990er Jahren realisiert wurden, tritt kein Zugriffskonflikt auf, da ein Befehl immer komplett abgearbeitet wird, bevor die Verarbeitung des nachfolgenden Befehls gestartet wird. Darüber hinaus ist die Realisierung getrennter Speicher aufwendiger und kann die Kosten des Systems erhöhen. Erst bei Einsatz von Instruktionspipelining, welches zuerst in Spezialprozessoren für die digitale Signalverarbeitung (Digitale Signalprozessoren, DSP) eingesetzt wurde, wurden getrennte Speicher für Daten und Befehle für die Realisierung von Mikrorechnersystemen eingesetzt. Später wurden die zunächst CISC-basierten Standardprozessoren mehr und mehr durch RISC-Prozessoren ersetzt. Als Folge des hierbei verwendeten Instruktionspipelinings bekam die Harvard-Architektur eine immer größere Bedeutung für die Realisierung integrierter Mikroprozessoren und Mikrorechnersysteme.

13.7 Mikrocontroller

Mikrocontroller sind integrierte Mikrorechnersysteme, die neben einer CPU auch Speicher, Ein-/Ausgabeeinheiten sowie weitere für den Betrieb des Systems notwendige Komponenten, beispielsweise die Takterzeugung, enthalten. Durch die Integration des Systems auf einem Mikrochip kann die Verwendung von externen Komponenten auf ein Minimum reduziert werden. Auf diese Weise lassen sich kostengünstige Mikrorechner realisieren.

Beim Entwurf und Einsatz von Mikrocontrollern stehen üblicherweise die Kosten des Controllers und die Verlustleistung im Vordergrund. Daher besitzen Mikrocontroller eine deutlich geringere Rechenleistung als sie zum Beispiel von Prozessoren für den PC-Markt zur Verfügung gestellt werden. Auch wenn dies auf den ersten Blick als ein Nachteil erscheinen mag, darf nicht vergessen werden, dass Mikrocontroller häufig für Anwendungen mit relativ geringen Rechenleistungen eingesetzt werden.

Sehr deutlich wird der Vorteil von Mikrocontrollern, wenn die Kosten eines Controllers mit dem eines PC-basierten Systems verglichen werden. Ein PC-basiertes System mit CPU, Speicher und Hauptplatine kostet mehrere hundert Euro, während Mikrocontroller für wenige Euro, teilweise sogar für Preise unterhalb eines Euros, erhältlich sind.

Mikrocontroller werden in vielen eingebetteten Systemen des Alltags eingesetzt. Sie übernehmen die Steuerung von Haushaltsgeräten, Fernsehgeräten, Kraftfahrzeugen, von industriellen Anlagen oder auch Medizingeräten.

Viele Halbleiterhersteller bieten Mikrocontroller mit unterschiedlichen Eigenschaften an. Anbieter von Mikrocontrollern sind (in alphabetischer Reihenfolge) die Firmen

Atmel, Fujitsu, Infineon, Microchip, NEC, NXP, Renesas, Texas Instruments und ST Microelectronics.

Einige der wichtigsten Unterscheidungskriterien, die bei der Auswahl eines Controllers zu beachten sind, werden im Folgenden vorgestellt:

Wortbreite des Rechenwerks
Typische Mikrocontroller des unteren Preissegmentes setzen Rechenwerke mit einer Wortbreite von 8 bit ein. Controller für höhere Rechenleistungen verwenden Rechenwerke mit einer Breite von 32 bit. Darüber hinaus werden auch 16-Bit-Mikrocontroller angeboten.

Verwendete CPU
Die Wahl des Prozessors stellt einen entscheidenden Faktor für die Leistungsfähigkeit des Systems dar. Darüber hinaus kann es von praktischer Bedeutung sein, dass die Controller für unterschiedliche Produkte eines Unternehmens die gleiche CPU verwenden. Auf diese Weise kann das einmal erworbene Know-how sowie Entwurfssoftware auch für Folgeprodukte effizient eingesetzt werden.

Taktfrequenz
Mikrocontroller arbeiten mit relativ geringen Taktfrequenzen, die sich im Bereich von einigen MHz bis hin zu einigen hundert MHz bewegen.

Größe des eingebetteten Speichers
Häufig wird der Programmspeicher als Flashspeicher und der Datenspeicher als SRAM zusammen mit der CPU integriert. Hierbei variiert die Größe dieser Speicher zwischen wenigen kByte bis zu mehreren hundert kByte.

Eingebettete Schnittstellen
Während alle Mikrocontroller Möglichkeiten zur einfachen programmgesteuerten digitalen Ein-/Ausgabe besitzen, werden darüber hinaus weitere sehr unterschiedliche Schnittstellen in Hardware zur Verfügung gestellt.

Der grundsätzliche Aufbau eines Mikrocontrollers ist in Abb. 13.15 dargestellt. Die Komponenten eines Mikrocontrollers umfassen einen Mikroprozessor (CPU), Speicher für Programme und Daten und Ein-/Ausgabeeinheiten.

Die Ein- und Ausgabe von digitalen Daten wird bei allen Mikrocontrollern mithilfe sogenannter *Ports* unterstützt. Ports sind digitale bidirektionale Anschlüsse des Controllers, die sowohl als Eingänge als auch als Ausgänge genutzt werden können. Die Auswahl, ob ein bestimmter Anschluss als Eingang oder Ausgang genutzt wird, erfolgt über das Programm, welches von der CPU ausgeführt wird. Darüber hinaus erfolgt auch die Ein-/Ausgabe durch die Software, sodass Ports sehr universell einsetzbar sind.

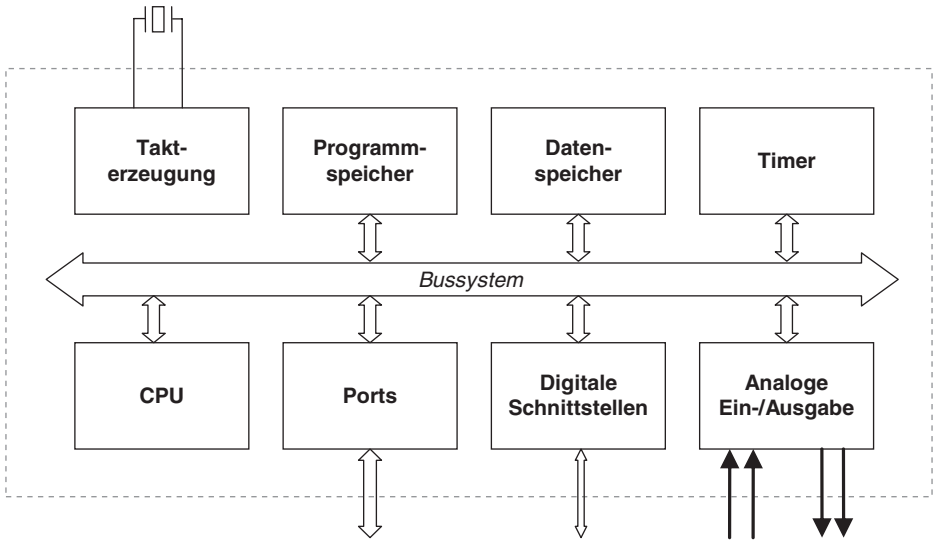


Abb. 13.15 Architektur eines Mikrocontrollers

In vielen Abwendungsfällen muss ein Mikrocontroller über standardisierte digitale Schnittstellen mit anderen Komponenten eines Systems oder externen Geräten kommunizieren. Grundsätzlich kann diese Kommunikation mithilfe von Ports realisiert werden, indem das jeweilige Schnittstellenprotokoll in Software implementiert wird. Durch diesen Ansatz wird jedoch ein bestimmter Anteil der CPU-Rechenleistung für die SW-basierte Implementierung des Schnittstellenprotokolls benötigt, sodass die zur Verfügung stehende Rechenleistung für die eigentliche Applikation reduziert wird. Um diesen Nachteil zu vermeiden, bieten Mikrocontroller verschiedene digitale Schnittstellen (zum Beispiel USB oder Ethernet) als integrierte Hardwaremodule an. Diese Schnittstellen implementieren das Protokoll zur Datenübertragung in HW und entlasten so die CPU des Controllers, die lediglich die zu sendende Daten bereitstellen beziehungsweise empfangene Daten von der Schnittstelle abholen muss.

Darüber hinaus enthalten viele Mikrocontroller Schnittstellen, die der Erweiterung des auf dem Controller integrierten Speichers dienen und mit externen SRAM- oder SDRAM-Speicherbausteinen kommunizieren können. Neben der digitalen Ein-/Ausgabe ermöglichen viele Mikrocontroller das Einlesen oder Ausgeben analoger Werte durch integrierte A/D- beziehungsweise D/A-Umsetzer.

Eine weitere typische Mikrocontrollerkomponente sind die sogenannten Timer, die im deutschen Sprachgebrauch zum Teil auch als Zeitgeber bezeichnet werden. Im Grunde handelt es sich bei Timern um integrierte Zähler, die entweder mit einem internen Takt des Controllers oder mit einem von außen zugeführten Takt betrieben werden können. In Abhängigkeit vom Zählerstand können verschiedene, programmierbare

Ereignisse ausgelöst werden. Zum Beispiel kann bei Erreichen eines vorprogrammierten Zählerstands der Ausgabewert einer der Controlleranschlüsse invertiert werden, wodurch sich ein Rechtecksignal erzeugen lässt. In der Regel lässt sich das erzeugte Signal in Frequenz und Tastverhältnis mit geringem Softwareaufwand modifizieren. Timer werden für praktische Anwendungen häufig eingesetzt. Sie erlauben unter anderem den regelmäßigen Aufruf von Unterprogrammen sowie die zeitliche Vermessung von Signalen.

Mikrocontroller verfügen darüber hinaus über eine integrierte Hardwareeinheit zur Takterzeugung, die das Taktsignal für den Betrieb des Controllers generiert. Die Auswahl der erzeugten Taktfrequenz erfolgt mithilfe weniger externer Komponenten, zum Beispiel mithilfe eines externen Quarzes oder eines RC-Gliedes. Die meisten Mikrocontroller besitzen daneben die Möglichkeit, den Systemtakt durch einen integrierten Oszillator zu erzeugen. In diesem Fall kann auf externe Komponenten völlig verzichtet werden.

Wird für eine Anwendung eine möglichst exakte Taktfrequenz benötigt, empfiehlt sich die Verwendung eines externen Quarzes. Die internen Oszillatoren können in der Regel eine Frequenzabweichung von einigen Prozent aufweisen und sind auch im Hinblick auf die Temperaturstabilität einem quarzbasierten Oszillator unterlegen.

Mikrocontroller sind also integrierte Schaltkreise, die alle notwendigen Komponenten eines Rechners beinhalten. Auf Basis von Mikrocontrollern lassen sich sehr einfach kostengünstige programmierbare Systeme realisieren, deren Einsatzgebiet nahezu unbegrenzt ist.

13.8 Übungsaufgaben

In den folgenden Aufgaben werden einige Themen dieses Kapitels aufgegriffen. Die Lösungen der Aufgaben finden Sie am Ende des Buches.

Sofern nicht anders vermerkt, ist nur eine Antwort richtig.

Aufgabe 13.1

Welche Aussagen zu Adressräumen sind korrekt? *(Mehrere Antworten sind richtig)*

a) Bei Memory-Mapped-Adressierung besitzen Speicher und Ein-/Ausgabe unterschiedliche Adressräume.
b) Bei Port-Mapped-Adressierung besitzen Speicher und Ein-/Ausgabe gemeinsame Adressräume.
c) Bei Memory-Mapped-Adressierung besitzen Speicher und Ein-/Ausgabe gemeinsame Adressräume.
d) Bei Port-Mapped-Adressierung besitzen Speicher und Ein-/Ausgabe unterschiedliche Adressräume.

Aufgabe 13.2
Wie wird die Adressierungsart bezeichnet, bei der die Speicheradresse direkt aus dem Befehlswort übernommen wird?

a) Unmittelbare Adressierung
b) Absolute Adressierung
c) Indirekte Adressierung
d) Indirekte Adressierung mit Verschiebung

Aufgabe 13.3
Welche Adressierungsarten verwenden einen „Adressspeicher“ (zum Beispiel CPU-Register)? *(Mehrere Antworten sind richtig)*

a) Unmittelbare Adressierung
b) Absolute Adressierung
c) Indirekte Adressierung
d) Indirekte Adressierung mit Verschiebung

Aufgabe 13.4
Mit welchen Maßnahmen kann die Rechenleistung eines Mikroprozessors gesteigert werden? *(Mehrere Antworten sind richtig)*

a) Erhöhung der Taktfrequenz
b) Spezialbefehle
c) Instruktions-Pipelining
d) VLIW

Aufgabe 13.5
Was ist der wesentliche Unterschied zwischen einer Von-Neumann- und einer Harvard-Architektur?

a) Die Harvard-Architektur kann nur für CISC-Prozessoren eingesetzt werden.
b) Die Von-Neumann-Architektur verwendet Flash als Instruktionsspeicher, die typische Harvard-Architektur dagegen SRAM
c) Die Harvard-Architektur besitzt getrennte Speicher für Instruktionen und Daten.
d) Die Harvard-Architektur unterstützt weniger Befehle als die Von-Neumann-Architektur.

Mikrocontroller

14

Mikrocontroller sind kleine Rechnersysteme, die in einem Baustein alles beinhalten, was zur Realisierung eines Computers benötigt wird: Sie enthalten eine CPU, Speicher und auch Ein-/Ausgabeeinheiten. Die Vorteile eines Mikrocontrollers sind die kompakte Bauform und der günstige Preis. Mikrocontroller werden in unterschiedlichsten, meist kostensensitiven, Anwendungen eingesetzt. Ein typisches Einsatzgebiet sind Steuerungs- und Regelungsanwendungen. In Ihrer Waschmaschine sorgt beispielsweise ein Mikrocontroller dafür, dass Sie ein Waschprogramm auswählen können. Er regelt die Wassertemperatur und steuert unter anderem die Elektronik für den Trommelmotor und die Pumpen an.

Die Hersteller von Mikrocontrollern bieten eine relativ große Produktpalette an. Meistens werden die Produkte eines Herstellers in Familien unterteilt. Die Produkte einer solchen Familie besitzen in der Regel die gleiche CPU, unterscheiden sich aber im Hinblick auf die Speicherkapazität oder die integrierten Ein-/Ausgabekomponenten. Aufgrund dieser Produktvielfalt kann der Anwender den Controller auswählen, der im Hinblick auf die technischen Eigenschaften und die Kosten optimal für das geplante Einsatzgebiet geeignet ist.

Dieses Kapitel bietet einen Einstieg in die Technik der Mikrocontroller. Der Mikrocontroller *ATmega32* aus der AVR-Familie des Herstellers Atmel wird exemplarisch vorgestellt. Die hier vorgestellten Grundprinzipien lassen sich auf zahlreiche andere Mikrocontrollern übertragen und sind nicht auf die AVR-Familie beschränkt.

14.1 Die Mikrocontroller-Familie AVR

Die AVR-Mikrocontroller sind relativ einfach strukturiert und eignen sich gut für erste Lernschritte im Bereich der Mikrorechnertechnik. Viele der AVR-Mikrocontroller werden in DIP-Gehäusen *(Dual-Inline-Package)* angeboten, die sich gut für vertiefende Experimente auf einem Steckbrett eignen.

W. Gehrke et al., *Digitaltechnik,* Springer-Lehrbuch,
DOI 10.1007/978-3-662-49731-9_14

Alle Controller der AVR-Familie basieren auf einem RISC-Mikroprozessor, der eine zweistufige Befehlspipeline besitzt. Das Rechenwerk der CPU hat eine Wortbreite von 8 bit. Die Firma Atmel unterteilt die Mikrocontroller der AVR-Familie in mehrere Gruppen, von denen drei im Folgenden kurz vorgestellt werden.

tinyAVR
Die tinyAVR-Mikrocontroller zeichnen sich durch eine kleine Gehäuseform mit wenigen Anschlüssen aus. Viele der im DIP-Gehäuse angebotenen Controller besitzen 8 oder 14 Anschlüsse, wovon jeweils 2 für die Spannungsversorgung verwendet werden. Die Controller können mit einer Spannung zwischen 1,8 und 5,5 V betrieben werden. Der Programmspeicher ist, wie bei allen Controllern der AVR-Familie, als Flashspeicher ausgeführt. Die Größe dieses Speichers liegt für tinyAVR-Controller meist zwischen 1 und 8 kByte. Zur Speicherung von Daten stehen ein SRAM und ein EEPROM zur Verfügung, deren Größe 64 bis 512 Byte beträgt. Alle Controller besitzen mindestens einen Timer und mindestens eine Schnittstelle zur seriellen digitalen Datenübertragung. Für das Einlesen analoger Werte stehen teilweise AD-Umsetzer zur Verfügung.

megaAVR
Die Mikrocontroller der megaAVR-Serie sind umfangreicher ausgestattet als die tinyAVR-Controller. Sie besitzen einen größeren Flash-Programmspeicher, dessen Größe zwischen 8 und 256 kByte liegt. Zur Speicherung von Daten stehen SRAM- oder EEPROM-Speicher mit einer Größe von bis zu 4 kByte zur Verfügung. Darüber hinaus besitzen die Controller der megaAVR-Serie mindestens zwei Timer und verfügen über eine größere Anzahl digitaler Schnittstellen als die tinyAVR-Controller.

Die in den megaAVR-Controllern verwendete CPU besitzt einen Hardware-Multiplizierer, der eine schnelle Multiplikation von 8 bit breiten Operanden ermöglicht.

Als ein Beispiel für die Controller der megaAVR-Serie ist die Pinbelegung des Mikrocontrollers *ATmega32* in Abb. 14.1 dargestellt. Dieser Controller besitzt einen Flash-Programmspeicher der Größe 32 kByte, 2 kByte SRAM und 1 kByte EEPROM-Speicher, sowie diverse eingebettete Peripheriekomponenten.

Der *ATmega32* besitzt 32 Portanschlüsse *(PA0-PA7, PB0-PB7, PC0-PC7 und PD0-PD7),* deren Funktion durch das ausgeführte Programm festgelegt wird. Die weiteren Anschlüsse dienen der Stromversorgung *(VCC, AVCC und GND)* oder können zur Erzeugung des Systemtaktes *(XTAL1, XTAL2)* oder zum Rücksetzen des Controllers in den Einschaltzustand *(/Reset)* verwendet werden. Die in Klammern angegebenen Pinbezeichnungen beziehen sich auf die sogenannten *alternativen Portfunktionen*. Per Software kann ausgewählt werden, ob die Anschlüsse direkt über die Software gesteuert werden sollen (Funktion als Ein-/Ausgabe-Ports) oder ob sie als Anschlüsse für eingebettete Peripheriekomponenten eingesetzt werden.

Aufgrund seines relativ großen Programmspeichers und einer großen Anzahl per Software steuerbarer Anschlüsse eignet sich der *ATmega32* gut für die Durchführung praktischer Experimente.

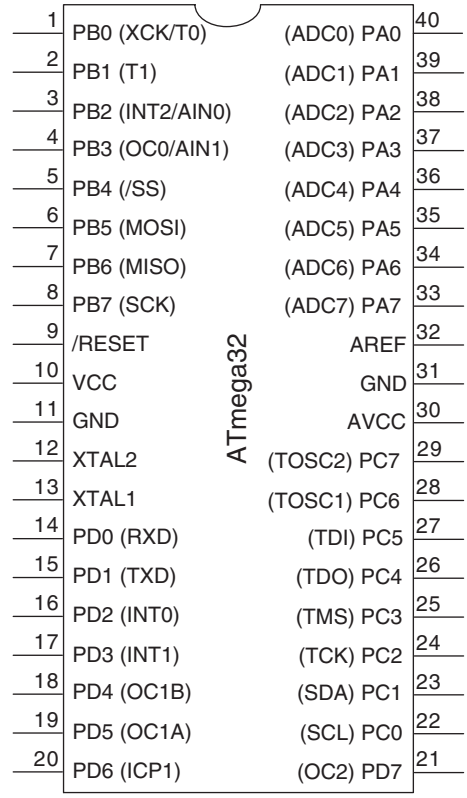


Abb. 14.1 Pinbelegung des Mikrocontrollers *ATmega32* im 40-poligen DIL-Gehäuse

AVR XMEGA

Einer der vielen Unterschiede der AVR-XMEGA-Serie zu den zuvor vorgestellten AVR-Serien ist der Einsatz eines DMA-Controllers *(Direct Memory Access)* in Kombination mit dem sogenannten Event-System. Diese Module ermöglichen unter anderem einen Datenaustausch zwischen den Komponenten des Systems, ohne die CPU mit dem eigentlichen Datentransfer zu belasten. Die Controller der XMEGA-Serie besitzen einen bis zu 384 kByte großen Flash-Programmspeicher und einen bis zu 32 kByte großen SRAM-Speicher, welcher bei einigen XMEGA-Controllern durch externen Speicher erweitert werden kann.

14.2 Programmierung von Mikrocontrollern

Die Programmierung von Mikrocontrollern kann in *Assembler* oder in einer Hochsprache erfolgen. Bei der Programmierung in Assembler besteht das Programm aus Befehlen, die genau wie im Programm angegeben, von der CPU ausgeführt werden. Diese Art der Programmierung hat verschiedene Vorteile: So kann zum Beispiel die

Ausführungszeit des Programms bereits während der Entwicklung exakt bestimmt werden. Darüber hinaus können Assemblerprogramme im Hinblick auf die Ausführungszeit und die Programmgröße (Flash-Speicherbedarf) optimiert werden.

Um Assembler-Programme schreiben zu können, ist eine genaue Kenntnis des Befehlssatzes der eingesetzten CPU erforderlich. Diese Notwendigkeit wird bei Einsatz einer Hochsprache vermieden, da der Compiler das Umsetzen des Quellcodes in die Befehle der CPU übernimmt. Der Einsatz einer Hochsprache vereinfacht daher die Programmierung und Programme können in kürzerer Zeit realisiert werden als bei einer Programmierung in Assembler. Die Optimierung der CPU-Befehle wird dann vom Compiler übernommen. Obwohl heutige Compiler eine gute Codeoptimierung durchführen, ist bei der Verwendung einer Hochsprache nicht gewährleistet, dass das Ergebnis das Optimum im Hinblick auf Rechenzeit und Speicherbedarf darstellt. Dennoch wird der Praxis die Programmierung in C/C++ in vielen Fällen der Programmierung in Assembler vorgezogen, da die Produktivität bei der Programmentwicklung im Vordergrund steht.

Im Rahmen dieses Kapitels wird am Beispiel des AVR-Mikrocontrollers *ATmega32* auf die Programmierung sowohl in Assembler als auch in C eingegangen. Die Beschäftigung mit der Programmierung in Assembler ermöglicht unter anderem ein tieferes Verständnis der Funktionsweise eines Mikroprozessors.

Grundsätzlich besitzen Mikrocontrollerprogramme die gleichen Elemente wie die Programme, die Sie vielleicht bereits auf einem PC entwickelt haben. Es gibt Funktionen, Verzweigungen, Schleifen usw. Einer der größten Unterschiede zwischen einem typischen PC-Programm und einem Mikrocontrollerprogramm ist, dass das Hauptprogramm des Controllers eine Endlosschleife enthält. Dass dies so sein muss, wird plausibel, wenn ein typisches Anwendungsgebiet eines Mikrocontrollers anschaut: Die Steuerung einer Waschmaschine.

Stellen Sie sich vor, Sie schalten Ihre Waschmaschine ein. Das Programm des Mikrocontrollers in der Steuereinheit wird gestartet und fragt die Bedienknöpfe ab. Das Programm ist aber wahrscheinlich schneller als Sie. Noch bevor Sie einen Taster des Bedienfeldes drücken können, stellt das Programm fest, dass offensichtlich nichts zu tun ist (es wurde ja kein Taster gedrückt) und wird beendet. Ihre Waschmaschine wäre mit einer solchen Steuerung nicht gut bedienbar.

Statt das Programm nach der ersten Abfrage des Bedienfeldes zu beenden, müssen die Taster und Schalter kontinuierlich abgefragt werden. Mit der Auswahl eines Waschprogramms wird der Mikrocontroller der Steuereinheit in ein entsprechendes Unterprogramm verzweigen, welches die Sensorik (Wasserstand, Wassertemperatur, usw.) abfragt und die Aktorik (Pumpe, Heizung, usw.) ansteuert. Nach dem Beenden des Unterprogramms wird wieder zur Abfrage des Bedienfeldes zurückgekehrt. Das Mikrocontroller-Programm wird also bis zum Abschalten der Waschmaschine laufen und muss damit eine Endlosschleife enthalten.

Die typische Grundstruktur eines Mikrocontrollerprogramms besteht aus zwei Teilen: Zu Beginn des Programms wird die Initialisierung des Systems ausgeführt und die Peripheriekomponenten initialisiert. Ist die Initialisierung abgeschlossen, werden die Eingangswerte des Controllers in einer Endlosschleife überprüft und gegebenenfalls

neue Ausgangswerte berechnet, die anschließend über die Ausgabeeinheiten ausgegeben werden. Dieser Grundstruktur folgen sowohl Assemblerprogramme als auch Hochsprachenprogramme.

14.2.1 Programmierung in Assembler

Im Gegensatz zu einem Hochsprachenprogramm darf in jeder Zeile eines Assemblerprogramms maximal ein CPU-Befehl stehen.

CPU-Befehle bestehen aus einer Bitkombination, die im Programmspeicher des Rechners abgelegt werden. Da jedoch niemand ein Programm schreiben möchte, das aus einer Textdatei mit Nullen und Einsen besteht, werden CPU-Befehle in einer für den Menschen lesbaren Form angegeben. Hierzu werden die Befehle als *Mnemonics* (Kürzel, die meist aus 1 bis 4 Buchstaben bestehen) angegeben. Nach dem Befehlskürzel werden zu verarbeitenden Operanden angeben.

Für die AVR-CPU kann man den Befehl zur Addition der Werte in den Arbeitsregistern *r5* und *r7* in binärer Form so schreiben:

```
0000110001010111
```

Deutlich besser lesbar ist diese Variante:

```
add r5, r7
```

Hier wird der Befehl als Mnemonic angegeben und sowohl die ausgeführte Operation als auch die verwendeten Operanden sind leicht erkennbar.

Neben den Mnemonics werden Ihnen in Assemblerprogrammen auch Label (Marken) begegnen. Mithilfe von Labeln wird eine Codezeile mit einem Symbol versehen, das im Programm eingesetzt werden kann. Im Verlauf dieses Kapitels werden Sie einige Beispiele für die Verwendung von Labels kennenlernen. Daher wird hier zunächst lediglich der Sprung in ein Unterprogramm als ein Beispiel für die Verwendung eines Labels dargestellt:

```
; Hier wird das Unterprogramm durch ein Label markiert
my_add_up:
   add r5, r7   ; Dieses einfache Unterprogramm führt eine Addition aus
   ret          ; Der "Return"-Befehl
                ; bewirkt die Rückkehr in das Hauptprogramm
; Das Hauptprogramm. In diesem Beispiel wird es auch markiert
main:
   …               ; Hier stehen weitere Befehle
   call my_add_up  ; Mit diesem Befehl wird das UP aufgerufen …
   …               ; und anschließend der hier stehende Code ausgeführt
```

Das Programm, welches die Umsetzung des Assemblercodes in die für den Rechner lesbare Form (also Nullen und Einsen) übersetzt, wird als *Assembler* bezeichnet. Eine Codeoptimierung, wie sie Hochsprachencompiler durchführen, findet bei der Übersetzung nicht statt.

14.2.2 Programmierung in C

Ein wesentlicher Aspekt der Mikrocontrollerprogrammierung ist der Zugriff auf die eingebettete Peripherie. Um beispielsweise einen digitalen Wert an einem Mikrocontrolleranschluss ausgeben zu können, muss die CPU den Ausgabewert in Registern der Peripheriekomponenten ablegen. Zwei wichtige Aspekte, die für die Mikrocontroller-Programmierung in C wichtig sind, werden in diesem Abschnitt vorgestellt.

14.2.2.1 Zugriff auf Peripheriekomponenten

Viele Mikrocontroller verwenden Memory-Mapped-I/O (vgl. Kapitel 13) um Zugriffe auf die Komponenten, zum Beispiel Ein-/Ausgabe-Einheiten, zu ermöglichen. Auf die Peripherie kann dann genauso wie auf den Datenspeicher zugegriffen werden. Auf welche Komponente zugegriffen wird, ergibt sich aus der verwendeten Adresse. Während es für „normale" Variablen völlig egal ist, an welcher Stelle sie im Speicher abgelegt werden, ist es für einen Peripheriezugriff essenziell, genau die richtige Adresse anzusprechen. Man muss also, im Gegensatz zu typischen PC-Programmen, dem Compiler vorschreiben, auf welche Adresse er zugreifen soll. Dies lässt sich relativ einfach mithilfe von Zeigern realisieren.

Nehmen wir an, Sie möchten auf eine Peripheriekomponente zugreifen, die unter der Adresse 234 erreichbar. Ein entsprechender Programmausschnitt, welcher der Peripheriekomponente den Wert 7 übergibt, kann dann wie folgt aussehen.

```
// Zeiger definieren und initialisieren
// Anschließend verweist der Zeiger auf die gewünschte Adresse
volatile char *periph_ptr = (char *) 234;

// Der Peripheriekomponente einen Wert übergeben
// Hierzu wird an die Adresse, auf die der Zeiger verweist,
// der gewünschte Wert abgelegt
*periph_ptr = 7;
```

In dem Programm wird ein Zeiger angelegt und mit der gewünschten Adresse initialisiert. Der Zugriff auf die Peripheriekomponente erfolgt dann durch die Dereferenzierung des Zeigers im unteren Teil des Beispielcodes. Mithilfe des Schlüsselwortes *volatile* wird der Compiler angewiesen, bei Verwendung des Zeigers keine Optimierung anzuwenden.

Warum dies wichtig ist, kann anhand eines einfachen Beispiels erläutert werden. Nehmen wir an, Ihr Mikrocontroller besitzt eine Peripheriekomponente, mit welcher der Ausgangswert eines Portanschlusses festgelegt werden kann. Schreibt man in ein Register der

Komponente den Wert 1, wird eine 1 am Ausgang ausgegeben; wird eine 0 geschrieben erscheint am Ausgang der Wert 0. Mithilfe dieser Komponente möchten Sie nun einen kurzen Impuls ausgeben. Ein entsprechender Programmausschnitt würde so aussehen:

```
*output_ptr = 0; // Wir gehen ganz sicher: Erst mal eine 0 ausgeben
*output_ptr = 1; // Eine 1 erscheint am Ausgang
*output_ptr = 0; // Aber nicht lange: Wir setzen den Ausgang wieder
auf 0
```

Aus Sicht des Compilers gibt es nur Speicherstellen. Der Compiler kennt keine Peripherie. Was würde also ein optimierender Compiler mit diesem Codeausschnitt tun?

Nun, der Compiler würde annehmen, dass die ersten beiden Zeilen überflüssig sind, da am Ende in der (vermeintlichen) Speicherstelle, auf die der Zeiger *output_ptr* verweist, eine Null stehen wird. Die ersten beiden Zeilen müssten aus Sicht des Compilers also nicht ausgeführt werden. Daher wird der Compiler diese Zeilen ignorieren und so die Rechenzeit des Programms reduzieren.

Für einen Datenspeicher wäre dieses Verhalten des Compilers korrekt und wünschenswert. Für den Zugriff auf eine Peripheriekomponente muss die Optimierung dagegen unterbunden werden, da andernfalls kein 1-Impuls am Ausgang des Controllers erscheint. Daher werden Zeiger auf Peripheriekomponenten stets mit dem C-Schlüsselwort *volatile* definiert.

In der Praxis muss man die Zeiger nicht selbst definieren. Die Hersteller von Mikrocontrollern stellen in der Regel Header-Dateien bereit, in denen die entsprechenden Definitionen bereits enthalten sind. Bei dem in diesem Kapitel vorgestellten AVR-Mikrocontroller ist dies die Datei *io.h*.

14.2.2.2 Setzen und Löschen von Bits

Häufig sind in einem Register einer Peripheriekomponente mehrere unterschiedliche Informationen zusammengefasst. Die einzelnen Bits des Registers besitzen also eine unterschiedliche Wirkung. In vielen Fällen möchte man daher nur einzelne Bits eines Registers modifizieren.

Nehmen wir zum Beispiel an, dass über den oben verwendeten Zeiger *output_ptr* der Ausgangswert von 8 Mikrocontrolleranschlüssen festgelegt werden kann. Jedem Bit des Peripherieregisters, auf das *output_ptr* verweist, ist genau ein Portausgang des Controllers zugeordnet.

Nehmen wir an, Sie möchten am Anschluss 3 eine 1 ausgeben. Hierzu muss also das Bit 3 des Registers gesetzt werden. Nehmen wir darüber hinaus an, dass die anderen Ausgabewerte unverändert bleiben sollen. Es darf also nur das Bit 3 des Registers modifiziert werden.

Dies lässt sich durch eine bitweise ODER-Verknüpfung, in C/C++ der Operator |, des Registerwertes mit dem Wert 8 (Bit 3 ist gesetzt, alle anderen Bits sind Null) erreichen. In C kann dies so erfolgen:

```
// Beispiel für das Setzen eines Bits
char tmp;
tmp = *output_ptr; // aktuellen Registerwert holen
tmp = tmp | 8; // Bit 3 setzen -- Achtung! Dies ist ein bitweises
ODER
// Nicht mit dem logischen ODER verwechseln: ||
*output_ptr = tmp; // und wieder in das Register schreiben
```

In C/C++ kann man dies auch in einer Zeile schreiben:

```
*output_ptr |= 8; // Bit 3 setzen - die kurze Variante
```

Um das beeinflusste Bit noch deutlicher im Code sichtbar zu machen wird häufig eine andere Variante für die Angabe der verwendeten Konstante gewählt:

```
*output_ptr |= 1<<3; // Hier sieht man besser welches Bit gesetzt wird
```

Die Schreibweise *1<<3* mag auf den ersten Blick ungewöhnlich aussehen. Vielleicht wirkt es umständlich, eine 1 um 3 Stellen nach links zu schieben, um so die Konstante 8 zu erhalten. Dennoch wird diese Schreibweise bevorzugt bei der Mikrocontrollerprogrammierung eingesetzt, da das modifizierte Bit explizit angegeben wird. Der Code ist besser lesbar.

Um einzelne Bits zu löschen wird die bitweise UND-Verknüpfung (Operator &) verwendet. Die UND-Verknüpfung mit einer Konstanten, die nur an einer Bitposition eine Null enthält uns ansonsten Einsen, löscht genau ein Bit und lässt die anderen unangetastet.

Möchte man das Bit 3 löschen, benötigt man den invertierten Wert von `1<<3`. In C wird die bitweise Invertierung durch den Operator ~ realisiert. Der Code für das Löschen des Bits 3 sieht also so aus:

```
*output_ptr &= ~(1<<3); // Die Klammern sind wichtig, da sonst zuerst
                        // die Invertierung und dann das Schieben
                        // ausgeführt wird - und das wäre falsch
```

Im Fall des AVR sind in der Headerdatei *io.h* viele Konstanten definiert, welche die Bitposition einzelner Peripherieregister enthalten. Im Datenblatt des Controllers findet man beispielsweise ein Register mit der Abkürzung TCCR1B. Unter anderem enthält dieses Register ein Bit, das mit der Bezeichnung WGM12 abgekürzt wird (was diese Bit bewirkt, wird später vorgestellt). Nach Einbinden der Headerdatei io.h kann dieses Bit mit der folgenden Zeile gesetzt werden:

```
TCCR1B |= 1<<WGM12; // Setzen des Bits WGM12 im Register TCCR1B
                    // so man muss nicht die genaue Position
                    // dieses Bits im Kopf haben
```

Es können auch mehrere Bits mit einer einzelnen Zuweisung gesetzt werden. Mit der nachfolgenden Codezeile werden beispielsweise die Bits *TWINT, TWSTA* und *TWEN* im Register *TWCR* gesetzt. Alle anderen Bits des Registers bleiben unverändert.

```
TWCR |= (1<<TWINT)|(1<<TWSTA)|(1<<TWEN);
```

14.3 Die AVR-CPU

Der Mikroprozessor der AVR-Controller ist eine RISC-CPU, die auf einer Harvard-Architektur basiert. Der Prozessor beinhaltet 32 Arbeitsregister mit einer Wortbreite von 8 bit, die sowohl für arithmetisch-logische Operationen als auch für Adressberechnungen eingesetzt werden können. Diese Register werden als *r0,r1,..., r30, r31* bezeichnet. Da die CPU auf einer Load/Store-Architektur basiert, können arithmetisch-logische Operationen nur mit Daten ausgeführt werden, die sich in den Arbeitsregistern befinden. Für das Laden der Register beziehungsweise das Abspeichern von Registerwerten stehen entsprechende Transferbefehle zur Verfügung.

Neben den Arbeitsregistern enthält der Mikroprozessor der AVR-Controller die folgenden Register:

Programmzähler (Program Counter, PC)
Der Programmzähler enthält die Adresse des als nächsten auszuführenden Befehls und besitzt eine Wortbreite, die es ermöglicht, den gesamten Programmspeicher des jeweiligen Controllers zu adressieren.

Statusregister
Das Statusregister besitzt eine Wortbreite von 8 bit. Jedes dieser Bits wird auch als Flag bezeichnet. Die Flags enthalten unter anderem Informationen über die ausgeführten Operationen (zum Beispiel Auftreten arithmetischer Überläufe).

Stackpointer
Der Stackpointer (Stapelzeiger) ist ein Register, welches die aktuelle SRAM-Adresse des Stapels enthält.

Befehlsregister
Das Befehlsregister dient der Zwischenspeicherung des aus dem Programmspeicher ausgelesenen Befehls. Das Befehlsregister ist bei der Programmierung nicht sichtbar und der Inhalt kann nicht durch Befehle modifiziert werden.

Der Mikroprozessor enthält darüber hinaus ein Steuerwerk, welches die Decodierung der Befehle vornimmt und CPU-interne Steuersignale zur Verarbeitung eines Befehls generiert. Das Rechenwerk des AVR-Prozessors enthält eine 8-Bit-ALU, welche die in

den Befehlen codierten arithmetischen und logischen Operationen ausführt. Die prinzipielle Struktur des AVR-Mikroprozessors ist in Abb. 14.2 dargestellt.

Die Befehle der CPU werden in 16 bit breiten Worten des Programmspeichers abgelegt. Der Mikroprozessor der AVR-Controller arbeitet die Befehle mithilfe einer zweistufigen Befehlspipeline ab. In der ersten Pipelinestufe wird ein Befehl aus dem Programmspeicher ausgelesen. In der zweiten Stufe wird der Befehl ausführt.

Die Mehrheit der arithmetischen Befehle benötigt für die Ausführung der zweiten Pipelinestufe lediglich einen Taktzyklus. Da der nachfolgende Befehl bereits während der Ausführung des aktuellen Befehls eingelesen wird, kann ein Befehlsdurchsatz von bis zu einem Befehl pro Taktzyklus erreicht werden.

Die meisten Load- und Storebefehle sowie die Sprungbefehle benötigen für die Verarbeitung mehrere Taktzyklen. Hierbei wird die Befehlspipeline des AVR angehalten, sodass der Befehlsdurchsatz bei Verwendung dieser Befehle absinkt.

Bei der Ausführung eines Befehls wird in vielen Fällen der Inhalt des Statusregisters berücksichtigt. Aus diesem Grund wird im Folgenden zunächst das Statusregister der CPU betrachtet. Im Anschluss daran werden die Befehle der AVR-CPU vorgestellt.

Das Statusregister (vgl. Tab. 14.1) besitzt eine Wortbreite von 8 bit und beinhaltet die nachfolgend erläuterten Flags.

I-Flag

Mithilfe des Interrupt-Flags können Interrupts freigegeben ($I = 1$) oder gesperrt werden ($\mathrm{I} = 0$).

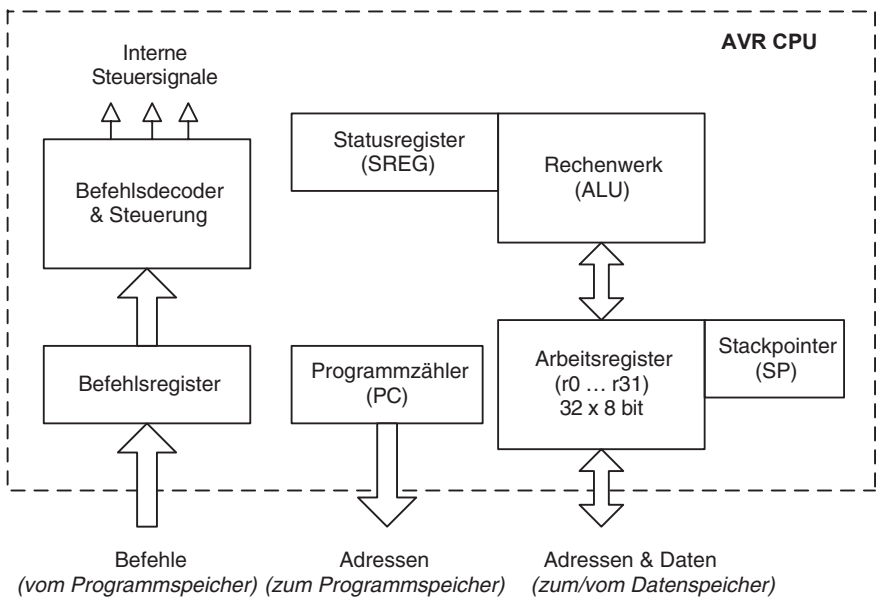


Abb. 14.2 Struktur der AVR-CPU

Tab. 14.1 Statusregister der AVR-CPU

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
I	T	H	S	V	N	Z	C
Interrupt-Flag	Bit-Transfer-Flag	Half-Carry-Flag	Sign-Flag	Overflow-Flag	Negative-Flag	Zero-Flag	Carry-Flag

T-Flag
Das T-Flag kann als Bitspeicher aufgefasst werden. Es kann durch Befehle gelöscht und gesetzt werden. Darüber hinaus kann es für die Ausführung von bedingten Sprüngen abgefragt werden.

Im Gegensatz zum I-Flag und T-Flag beziehen sich alle weiteren Flags auf arithmetische Operationen.

Z-Flag
Ist das Ergebnis einer Operation Null, wird dies durch ein gesetztes Zero-Flag ($Z = 1$) signalisiert.

N-Flag
Das Negative-Flag ist die Kopie des höchstwertigen Bits des Ergebnisses, da dieses Bit bei Zahlen in Zweierkomplementdarstellung das Vorzeichen repräsentiert.

C-Flag
Mithilfe des Carry-Flags wird gekennzeichnet ($C = 1$), ob bei einer vorzeichenlosen Operation ein Überlauf, also ein Verlassen des darstellbaren Zahlenbereichs aufgetreten ist. Darüber hinaus wird das C-Flag bei Schiebe- oder Rotationsbefehlen eingesetzt.

V-Flag
Das Overflow-Flag signalisiert mit $V = 1$ einen Überlauf bei vorzeichenbehafteten Operationen wie der Addition oder der Subtraktion.

S-Flag
Ein Überlauf bei einer Zweierkomplementoperation führt dazu, dass das höchstwertige Ergebnisbit nicht das korrekte Vorzeichen enthält (vgl. Kapitel 2). Somit kann durch das N-Flag in diesem Fall nicht das Vorzeichen des Ergebnisses bestimmt werden. Aus diesem Grund bietet die AVR-CPU ein weiteres Flag an: Mithilfe des Sign-Flags wird das wahre Vorzeichen, auch bei einem aufgetretenen Zweierkomplementüberlauf, angegeben. Das S-Flag ergibt sich aus der Exklusiv-Oder-Verknüpfung des N- und des V-Flags. Durch diese Verknüpfung enthält das S-Flag im Fall eines Überlaufs (V = 1) das invertierte N-Flag, während es eine Kopie des N-Flags enthält, wenn kein Überlauf aufgetreten ist.

H-Flag
Das Half-Carry-Flag wird gesetzt ($H = 1$), wenn bei einer vorzeichenlosen Operation ein Überlauf aus dem niederwertigen in das höherwertige Halbbyte auftritt. Das H-Flag ist zum Beispiel für Rechenoperationen mit Zahlen in BCD-Darstellung sinnvoll.

14.4 Der AVR-Befehlssatz

Dieser Abschnitt gibt eine Übersicht über den Befehlssatz der AVR-CPU. Mithilfe von arithmetischen und logischen Befehlen werden Arbeitsregister der CPU modifiziert. Transferbefehle werden genutzt um Daten aus den Registern in die Peripheriekomponenten oder den Speicher zu übertragen, beziehungsweise um Daten aus den Systemkomponenten in die Arbeitsregister zu transferieren. Eine weitere Gruppe sind die Befehle, die zur Steuerung des Programmablaufs genutzt werden. Mithilfe dieser Befehle können Sprünge, Verzweigungen und Schleifen realisiert werden.

14.4.1 Arithmetische und logische Befehle

Als Operanden für die arithmetischen und logischen Befehle können die Arbeitsregister verwendet werden. Die AVR-CPU verwendet ein sogenanntes *Zwei-Adress-Format*. Das bedeutet, dass maximal zwei Operanden durch einen Befehl adressiert werden. Das Ergebnis der ausgeführten Operation wird hierbei in einem der beiden Operandenregister abgelegt und der darin gespeicherte Wert wird überschrieben. Exemplarisch kann dies anhand des Additionsbefehls verdeutlich werden. Der Befehl *add r7,r12* führt eine Addition der Inhalte der Register *r7* und *r12* aus. Die Summe der beiden Operanden wird anschließend im erstgenannten Register *r7* abgelegt.

Darüber hinaus kann für einige Befehle auch die unmittelbare (engl. *immediate*) Adressierung verwendet werden. In diesem Fall ist der zweite Operand eine 8 bit breite Konstante, welche im Befehlswort abgelegt wird. Soll beispielsweise die Konstante 17 vom Inhalt des Registers *r23* subtrahiert werden, kann dies mithilfe des Befehls *subi r23,17* erfolgen. Der Buchstabe *i* ist hierbei das Kürzel für *immediate*.

Für alle Befehle, die eine unmittelbare Adressierung verwenden, gilt die Einschränkung, dass sie nur mit der oberen Hälfte des Arbeitsregistersatzes, also mit *r16* bis *r31*, verwendet werden können. Der Grund hierfür ist die Beschränkung der Befehlswortbreite auf 16 bit. Da die Konstante bereits 8 Bit belegt und für die Codierung der auszuführenden Operation weitere 4 Bit benötigt werden, verbleiben lediglich 4 Bit zur Codierung des Arbeitsregisters, womit nicht alle 32 Register adressiert werden können.

Bei vielen Mikrocontrollern der AVR-Serie werden auch einige Befehle mit Operanden der Wortbreite 16 bit unterstützt. Da die Arbeitsregister der AVR-CPU eine Wortbreite von 8 bit besitzen, werden die Operanden aus zwei aufeinanderfolgenden Registern

(Registerpaare) gebildet. Das Register mit dem niedrigeren Index enthält hierbei die unteren 8 Bit, das Register mit dem höheren Index die oberen 8 Bit des Operanden.

In Tab. 14.2 sind die wichtigsten arithmetischen und logischen Befehle der AVR-CPU zusammengefasst.

Die Flags des Statusregisters können auch direkt durch Befehle gesetzt oder gelöscht werden (Tab. 14.3).

14.4.2 Transferbefehle

Die bisher vorgestellten arithmetischen und logischen Befehle dienen der Verarbeitung von Daten, die in den Arbeitsregistern der CPU abgelegt sind. Für einen Datenaustausch zwischen den Arbeitsregistern und anderen Komponenten des Systems werden weitere Befehle, die sogenannten Transferbefehle, benötigt. Es existieren Befehle zum Kopieren von Daten zwischen Arbeitsregistern und zum Datenaustausch zwischen CPU und Peripheriekomponenten und dem Speicher.

Für den Datenaustausch mit Peripheriekomponenten (zum Beispiel Speicher oder Schnittstellen) werden Load- und Storebefehle bereitgestellt. Für die Adressierung bietet die AVR-CPU die Adressierungsarten *direkt, indirekt, indirekt mit Post-Inkrement, indirekt mit Pre-Inkrement* und *indirekt mit Verschiebung* an (vgl. Kapitel 13).

Im Fall der indirekten Adressierung wird eine 16-Bit-Adresse aus Registerpaaren geholt. Als mögliche Registerpaare stehen die Paare *r26:r27, r28:r29* und *r30:r31* zur Verfügung. Auf diese Weise kann ein Adressraum mit einer Adresswortbreite von 16 bit angesprochen werden. Zur Vereinfachung können diese Registerpaare auch mit neuen Symbolen *(X, Y* und *Z)* angesprochen werden. Die Register *X*, *Y* und *Z* stellen keine zusätzlichen Register dar, sondern sind lediglich andere Bezeichnungen für Registerpaare, die bereits im Arbeitsregistersatz enthalten sind. Für die Zuordnung der Registerbezeichnungen gilt Tab. 14.4.

Während alle Komponenten des Mikrocontrollers memory-mapped adressiert werden können, ist für einige häufig verwendete Komponenten auch ein Zugriff über eine io-mapped-basierte Adressierung mithilfe der Befehle *in* und *out* möglich. Diese Befehle benötigen weniger Programmspeicherplatz und werden schneller ausgeführt als die entsprechenden memory-mapped arbeitenden Load-/Storebefehle.

In Tab. 14.5 sind die wichtigsten Transferbefehle der AVR-CPU zusammengestellt. Grundsätzlich werden durch die Transferbefehle keine Flags des Statusregisters beeinflusst.

14.4.3 Befehle zur Programmablaufsteuerung

Zur Steuerung des Programmablaufs besitzen die Mikrocontroller der AVR-Familie verschiedene Sprungbefehle, mit denen unbedingte oder bedingte Sprünge ausgeführt

Tab. 14.2 Arithmetische und logische Befehle der AVR-CPU

ADD Rd,Rr	**Add**		
	Die Summe von Rd und Rr wird Rd zugewiesen		
	Flags: HSVZNC	Taktzyklen: 1	Befehlsworte: 1
ADC Rd,Rr	**Add with Carry**		
	Die Summe von Rd, Rr und dem Carry-Flag wird Rd zugewiesen		
	Flags: HSVZNC	Taktzyklen: 1	Befehlsworte: 1
ADIW Rd,K	**Add Immediate to Word**		
	Die Summe des Registerpaares Rd+1:Rd und der Konstanten K wird Rd+1:Rd zugewiesen[1]. ($0 \leq K \leq 63$)		
	Flags: SVZNC	Taktzyklen: 2	Befehlsworte: 1
AND Rd,Rr	**And**		
	Das Ergebnis der bitweisen Und-Verknüpfung von Rd und Rr wird Rd zugewiesen		
	Flags: SVNZ[2]	Taktzyklen: 1	Befehlsworte: 1
ANDI Rd,K	**And Immediate**		
	Das Ergebnis der bitweisen Und-Verknüpfung von Rd und der Konstanten K wird Rd zugewiesen. Der Befehl kann nur für die Register R16 bis R31 durchgeführt werden.		
	Flags: SVNZ[2]	Taktzyklen: 1	Befehlsworte: 1
ASR Rd	**Arithmetic Shift Right**		
	Rd wird um 1 bit nach rechts geschoben, wobei das höchstwertige Bit seinen Wert beibehält. Das niederwertigste Bit in das Carry-Flag übertragen.		
	Flags: SVNZ[3]	Taktzyklen: 1	Befehlsworte: 1
CLR Rd	**Clear Register**		
	Rd wird auf 0 gesetzt		
	Flags: SVNZ[4]	Taktzyklen: 1	Befehlsworte: 1
DEC Rd	**Decrement**		
	Rd wird um 1 erniedrigt		
	Flags: SVNZ	Taktzyklen: 1	Befehlsworte: 1
CBR Rd,K	**Clear Bit(s) in Register**		
	Das Ergebnis der bitweisen Und-Verknüpfung von Rd und der invertierten Konstante K wird Rd zugewiesen. Der Befehl kann nur für die Register R16 bis R31 durchgeführt werden.		
	Flags: SVNZ[2]	Taktzyklen: 1	Befehlsworte: 1
COM Rd	**One's Complement**		
	Rd wird invertiert		
	Flags: SVNZ[5]	Taktzyklen: 1	Befehlsworte: 1
CP Rd,Rr	**Compare**		
	Die Flags werden wie bei der Verwendung des Subtraktionsbefehls SUB gesetzt. Rd wird jedoch nicht überschrieben		
	Flags: HSVZNC	Taktzyklen: 1	Befehlsworte: 1

(Fortsetzung)

Tab 14.2 (Fortsetzung)

CPC Rd,Rr	**Compare with Carry**
	Die Flags werden wie bei der Verwendung des Subtraktionsbefehls SBC gesetzt. Rd wird jedoch nicht überschrieben Flags: HSVZNC Taktzyklen: 1 Befehlsworte: 1
CPI Rd,K	**Compare Immediate**
	Die Flags werden wie bei der Verwendung des Subtraktionsbefehls SUBI gesetzt. Rd wird jedoch nicht überschrieben. Der Befehl kann nur für die Register R16 bis R31 durchgeführt werden. Flags: HSVZNC Taktzyklen: 1 Befehlsworte: 1
EOR Rd,Rr	**Exclusive Or**
	Das Ergebnis der bitweisen Exklusiv-Oder-Verknüpfung von Rd und Rr wird Rd zugewiesen Flags: SVNZ[2] Taktzyklen: 1 Befehlsworte: 1
INC Rd	**Increment**
	Rd wird um 1 erhöht Flags: SVNZ Taktzyklen: 1 Befehlsworte: 1
LSL Rd	**Logical Shift Left**
	Das höchstwertigste Bit von Rd in das Carry-Flag übertragen. Anschließend wird Rd um 1 bit nach links geschoben, wobei das niederwertigste Bit auf 0 gesetzt wird. Flags: HSVNZC[3] Taktzyklen: 1 Befehlsworte: 1
LSR Rd	**Logical Shift Right**
	Das niederwertigste Bit von Rd in das Carry-Flag übertragen. Anschließend wird Rd um 1 bit nach rechts geschoben, wobei das höchstwertigste Bit auf 0 gesetzt wird. Flags: SVNZC[3] Taktzyklen: 1 Befehlsworte: 1
MUL Rd,Rr	**Multiply Unsigned**
	Das Produkt aus Rd und Rr wird im Registerpaar R1:R0 abgelegt. Beide Operanden werden als vorzeichenlose Zahlen behandelt. Rd wird nicht überschrieben. Flags: ZC[6] Taktzyklen: 2 Befehlsworte: 1
MULS Rd,Rr	**Multiply Signed**
	Das Produkt aus Rd und Rr wird im Registerpaar R1:R0 abgelegt. Beide Operanden werden als Zweierkomplement-Zahlen behandelt. Rd wird nicht überschrieben. Flags: ZC[6] Taktzyklen: 2 Befehlsworte: 1
MULSU Rd,Rr	**Multiply Unsigned/Signed**
	Das Produkt aus Rd und Rr wird im Registerpaar R1:R0 abgelegt. Rd wird als vorzeichenlos, Rr als Zweierkomplement-Zahl behandelt. Rd wird nicht überschrieben. Flags: ZC[6] Taktzyklen: 2 Befehlsworte: 1

(Fortsetzung)

Tab 14.2 (Fortsetzung)

NEG Rd	**Two's Complement**
	Rd wird negiert. Die ausgeführte Operation entspricht „0 - Rd". Flags: HSVNZC Taktzyklen: 2 Befehlsworte: 1
NOP	**No Operation**
	Dieser Befehl führt keine Operation aus und es werden somit keine Register modifiziert. Der Befehl kann u.a. für Verzögerungsschleifen verwendet werden. Flags: keine Taktzyklen: 1 Befehlsworte: 1
OR Rd,Rr	**Or**
	Das Ergebnis der bitweisen Oder-Verknüpfung von Rd und Rr wird Rd zugewiesen. Flags: SVNZ[2] Taktzyklen: 1 Befehlsworte: 1
ORI Rd,K	**Or Immediate**
	Das Ergebnis der bitweisen Oder-Verknüpfung von Rd und der Konstanten K wird Rd zugewiesen. Der Befehl kann nur für die Register R16 bis R31 durchgeführt werden. Flags: SVNZ[2] Taktzyklen: 1 Befehlsworte: 1
ROL Rd	**Rotate Left**
	Das höchstwertigste Bit von Rd in das Carry-Flag übertragen. Anschließend wird Rd um 1 bit nach links geschoben, wobei das niederwertigste Bit auf den Wert des Carry-Flags vor Ausführung des Befehls gesetzt wird. Flags: HSVNZC[3] Taktzyklen: 1 Befehlsworte: 1
ROR Rd	**Rotate Right**
	Das niederwertigste Bit von Rd in das Carry-Flag übertragen. Anschließend wird Rd um 1 bit nach rechts geschoben, wobei das höchstwertigste Bit auf den Wert des Carry-Flags vor Ausführung des Befehls gesetzt wird. Flags: SVNZC[3] Taktzyklen: 1 Befehlsworte: 1
SBR Rd,K	**Set Bit(s) in Register**
	Das Ergebnis der bitweisen Oder-Verknüpfung von Rd und der invertierten Konstante K wird Rd zugewiesen und ist somit identisch mit dem Befehl ORI. Der Befehl kann nur für die Register R16 bis R31 durchgeführt werden. Flags: SVNZ[2] Taktzyklen: 1 Befehlsworte: 1
SBC Rd,Rr	**Subtract with Carry**
	Die Differenz von Rd und (Rr+Carry-Flag) wird Rd zugewiesen. Flags: HSVZNC Taktzyklen: 1 Befehlsworte: 1
SBCI Rd,K	**Subtract with Carry Immediate**
	Die Differenz von Rd und (K+Carry-Flag) wird Rd zugewiesen. Der Befehl kann nur für die Register R16 bis R31 durchgeführt werden. Flags: HSVZNC Taktzyklen: 1 Befehlsworte: 1
SBIW Rd,K	**Subtract Immediate from Word**
	Die Different des Registerpaares Rd+1:Rd und der Konstanten K wird Rd+1:Rd zugewiesen[1]. ($0 \leq K \leq 63$) Flags: SVZNC Taktzyklen: 2 Befehlsworte: 1

(Fortsetzung)

Tab 14.2 (Fortsetzung)

TST Rd	Test Register (for Zero or Minus)
	Die bitweise Und-Verknüpfung von Rd mit sich selbst wird ausgeführt, wobei Rd nicht modifiziert wird. Flags: SVNZ[2] Taktzyklen: 1 Befehlsworte: 1

Anmerkungen:

[1] Das Register Rd+1 enthält die oberen 8 bit und Rd die unteren 8 bit des verarbeiteten 16-bit-Wertes. Der Befehl kann nur für die Registerpaare R25:R24, R27:R26, R29:R28 und R31:R30 durchgeführt werden.

[2] V = 0, unabhängig von den Operanden

[3] V = N⊕C

[4] S = 0, N = 0, V = 0, Z = 1

[5] V = 0, C = 1 unabhängig vom Operanden

[6] C = Bit 15 des Produktes

Tab. 14.3 Statusregister der AVR-CPU

Flag	Befehl zum Setzen	Befehl zum Löschen
C	SEC	CLC
N	SEN	CLN
Z	SEZ	CLZ
V	SEV	CLV
S	SES	CLS
H	SEH	CLH
T	SET	CLT
I	SEI	CLI

Tab. 14.4 Alternative Bezeichnungen für Register der AVR-CPU

Registerpaar	Synonyme Bezeichnung
r27:r26	X
r29:r28	Y
r31:r30	Z

werden können. Während die bedingten Sprungbefehle nur eine relative Adressierung zur Bestimmung des Sprungziels unterstützen, stehen die unbedingten sowohl mit relativer als auch mit absoluter und indirekter Adressierung zur Verfügung.

Der Aufruf von Unterprogrammen wird beim AVR durch die *CALL*-Befehle unterstützt. Diese Befehle entsprechen einem unbedingten Sprung, wobei zusätzlich die Rücksprungadresse, an der das Programm nach Beenden des Unterprogramms fortgesetzt werden soll, auf dem Stack abgelegt wird. Für das Beenden eines Unterprogramms wird der Befehl *RET* verwendet. Dieser lädt die auf dem Stack abgelegte Rücksprungadresse in den Program Counter und setzt somit das aufrufende Programm an der Stelle fort, die dem Einsprung in das Unterprogramm folgt.

Tab. 14.5 Transferbefehle der AVR-CPU

LDI Rd,K	**Load Immediate**
	Das Register Rd wird mit der Konstanten K geladen. Der Befehl kann nur für die Register R16 bis R31 durchgeführt werden. Taktzyklen: 1 Befehlsworte: 1
LDS Rd,K	**Load Direct (from data space)**
	Die 16-bit-Konstante K wird zur absoluten Adressierung verwendet. Der so adressierte Wert wird in das Register Rd übertragen. Taktzyklen: 2 Befehlsworte: 2
LD Rd,*P*	**Load Indirect**
	Das Registerpaar *P* wird für die indirekte Adressierung verwendet. Der so adressierte Wert wird in das Register Rd übertragen. Für das Symbol *P* ist entweder X, Y oder Z einzusetzen. Taktzyklen: 2 Befehlsworte: 1
LD Rd,*P*+	**Load Indirect with Post-Increment**
	Das Registerpaar *P* wird für die indirekte Adressierung verwendet. Der so adressierte Wert wird in das Register Rd übertragen. Anschließend wird *P* inkrementiert. Für das Symbol *P* ist entweder X, Y oder Z einzusetzen. Taktzyklen: 2 Befehlsworte: 1
LD Rd,-*P*	**Load Indirect with Pre-Decrement**
	P wird dekrementiert. Anschließend wird das Registerpaar *P* für die indirekte Adressierung verwendet. Der so adressierte Wert wird in das Register Rd übertragen. Für das Symbol *P* ist entweder X, Y oder Z einzusetzen. Taktzyklen: 2 Befehlsworte: 1
LDD Rd,*P*+q	**Load Indirect with Displacement**
	Das Registerpaar *P* und die Konstante q werden addiert und die Summe wird für die indirekte Adressierung verwendet. Der so adressierte Wert wird in das Register Rd übertragen. Für das Symbol *P* ist entweder Y oder Z einzusetzen. Für q gilt: $0 \leq q \leq 63$ Taktzyklen: 2 Befehlsworte: 2
LPM Rd **LPM Rd,Z**	**Load from Program Memory**
	Das Register *Z* wird für die indirekte Adressierung des Programmspeichers verwendet. Der so adressierte Wert wird in das Register Rd übertragen. Taktzyklen: 3 Befehlsworte: 1
LPM Rd,*Z*+	**Load from Program Memory with Post-Increment**
	Das Register *Z* wird für die indirekte Adressierung des Programmspeichers verwendet. Der adressierte Wert wird in das Register Rd übertragen und Z wird inkrementiert. Taktzyklen: 3 Befehlsworte: 1
LD Rd,*P*+	**Load Indirect with Post-Increment**
	Das Registerpaar *P* wird für die indirekte Adressierung verwendet. Der so adressierte Wert wird in das Register Rd übertragen. Anschließend wird *P* inkrementiert. Für das Symbol *P* ist entweder X, Y oder Z einzusetzen. Taktzyklen: 2 Befehlsworte: 1

(Fortsetzung)

Tab 14.5 (Fortsetzung)

STS K,Rr	**Store Direct (to data space)**
	Die 16-bit-Konstante K wird zur absoluten Adressierung verwendet. In die adressierte Speicherstelle wird der Wert des Registers Rr übertragen. Taktzyklen: 2 Befehlsworte: 1
ST *P*,Rr	**Store Indirect**
	Das Registerpaar *P* wird für die indirekte Adressierung verwendet. In die adressierte Speicherstelle wird der Wert des Registers Rr übertragen. Für das Symbol *P* ist entweder X, Y oder Z einzusetzen. Taktzyklen: 2 Befehlsworte: 1
ST *P*+,Rr	**Store Indirect with Post-Increment**
	Das Registerpaar *P* wird für die indirekte Adressierung verwendet. In die adressierte Speicherstelle wird der Wert des Registers Rr übertragen. Anschließend wird *P* inkrementiert. Für das Symbol *P* ist entweder X, Y oder Z einzusetzen. Taktzyklen: 2 Befehlsworte: 1
ST -*P*,Rr	**Store Indirect with Pre-Increment**
	P wird dekrementiert. Anschließend wird das Registerpaar *P* für die indirekte Adressierung verwendet. In die adressierte Speicherstelle wird der Wert des Registers Rr übertragen. Für das Symbol *P* ist entweder X, Y oder Z einzusetzen. Taktzyklen: 2 Befehlsworte: 1
ST *P*+q,Rr	**Store Indirect with Pre-Increment**
	Das Registerpaar *P* und die Konstante q werden addiert und die Summe wird für die indirekte Adressierung verwendet. In die adressierte Speicherstelle wird der Wert des Registers Rr übertragen. Für das Symbol *P* ist entweder Y oder Z einzusetzen. Taktzyklen: 2 Befehlsworte: 1
IN Rd,K	**Input (from I/O space)**
	Die Konstante K wird zur Adressierung des I/O-Adressraums verwendet. Der adressierte Wert wird in das Register Rd übertragen. Taktzyklen: 1 Befehlsworte: 1
OUT K,Rr	**Output (to I/O space)**
	Die Konstante K wird zur Adressierung des I/O-Adressraums verwendet. In die adressierte Speicherstelle wird der Wert des Registers Rr übertragen. Taktzyklen: 1 Befehlsworte: 1
MOV Rd,Rr	**Move Register**
	Das Register Rr wird in das Register Rd übertragen. Taktzyklen: 1 Befehlsworte: 1
MOVW Rd,Rr	**Move Register Pair**
	Das Registerpaar Rr+1:Rr wird in das Registerpaar Rd+1:Rd übertragen. Taktzyklen: 1 Befehlsworte: 1
PUSH Rr	**Push Register on Stack**
	Das Register Rr wird auf dem Stack abgelegt. Anschließend wird der Stapelzeiger (Stack Pointer, SP) dekrementiert. Taktzyklen: 2 Befehlsworte: 1
POP Rd	**Pop Register from Stack**
	Der Stapelzeiger (Stack Pointer, SP) wird inkrementiert. Anschließend wird Rd mit dem durch SP adressierten Wert geladen. Taktzyklen: 2 Befehlsworte: 1

In Tab. 14.6 sind die wichtigsten Sprungbefehle der AVR-CPU zusammengestellt.

Neben den oben genannten Sprungbefehlen besitzt die AVR-CPU Befehle, mit denen ein einzelner nachfolgender Befehl übersprungen werden kann. Diese sogenannten Skipbefehle sind in Tab. 14.7 zusammengefasst.

Für die Ausführung von bedingten Sprüngen können die Befehle *BRBC* und *BRBS* verwendet werden. Hierbei ist die Angabe des abzufragenden Flags des Statusregisters erforderlich. Da dies das Programm unübersichtlicher machen kann, stehen für jedes Bit des Statusregisters spezielle Sprungbefehle zur Verfügung. Diese Befehle stellen keine zusätzlichen Befehle dar, sondern sind lediglich synonyme Bezeichnungen für die entsprechenden Varianten des *BRBC*- beziehungsweise *BRBS*-Befehls. Tab. 14.8 fasst die Synonyme für bedingte relative Sprungbefehle zusammen.

14.5 Verwendung der AVR-Befehle

In diesem Abschnitt wird die Verwendung des Befehlssatzes anhand einiger Beispiele verdeutlicht. Hierzu werden Programmfragmente vorgestellt, die auch in größeren AVR-Programmen eingesetzt werden können.

14.5.1 Arithmetische und logische Grundfunktionen

Die Mikroprozessoren der AVR-Familie unterstützen Befehle zur Verarbeitung von Byteoperanden. Sollen Operanden mit einer größeren Wortbreite verarbeitet werden, müssen hierzu mehrere aufeinander folgende Befehle verwendet werden. Im Folgenden wird dies für einige arithmetische und logische Grundfunktionen vorgestellt.

14.5.1.1 Setzen und Löschen einzelner Bits

Zum Setzen oder Löschen einzelner oder auch mehrerer Bits stehen die Befehle *sbr* und *cbr* zur Verfügung. Alternativ können hierfür auch die logischen Befehle *and, andi* beziehungsweise *or* oder *ori* eingesetzt werden. Sowohl für die Befehle *sbr* und *cbr* als auch für alle Befehle mit unmittelbarer Adressierung wie *andi* oder *ori* dürfen nur die Register *r16* bis *r31* verwendet werden.

```
; Setzen des Bits 4 im Register r20
sbr   r20,16
ori   r20,0x10   ; Alternative mit identischer Funktion
; Löschen des Bits 2 im Register r23
cbr   r23,4
andi  r23,0xFB   ; Alternative mit identischer Funktion
```

Tab. 14.6 Sprungbefehle der AVR-CPU

BRBC k,s	**Branch if Bit in Status Register Cleared**
	Dieser Befehl führt einen relativen bedingten Sprung aus. Ist das Bit s im Statusregister gelöscht, wird das Programm an der um k Programmspeicherworte verschobenen Position fortgesetzt. Andernfalls wird das Programm mit dem nachfolgenden Befehl fortgesetzt. Wird der Sprung ausgeführt, benötigt der Befehl zur Ausführung 2 Taktzyklen, andernfalls 1 Taktzyklus. Für k bzw. s gilt: $-64 \leq k \leq 63$ und $0 \leq s \leq 7$ Taktzyklen: 1/2 Befehlsworte: 1
BRBS k,s	**Branch if Bit in Status Register Set**
	Dieser Befehl führt einen relativen bedingten Sprung aus. Ist das Bit s im Statusregister gesetzt, wird das Programm an der um k Programmspeicherworte verschobenen Position fortgesetzt. Andernfalls wird das Programm mit dem nachfolgenden Befehl fortgesetzt. Wird der Sprung ausgeführt, benötigt der Befehl zur Ausführung 2 Taktzyklen, andernfalls 1 Taktzyklus. Für k bzw. s gilt: $-64 \leq k \leq 63$ und $0 \leq s \leq 7$ Taktzyklen: 1/2 Befehlsworte: 1
CALL k	**Call to a Subroutine**
	Aufruf eines Unterprogramms an der Programmspeicheradresse k. Die Rücksprungadresse wird auf dem Stack abgelegt. Taktzyklen[1]: 4 Befehlsworte: 2
ICALL	**Indirect Call to a Subroutine**
	Aufruf eines Unterprogramms an der Programmspeicheradresse, die durch das Z-Register indirekt adressiert wird. Taktzyklen[1]: 3 Befehlsworte: 1
IJMP	**Indirect Jump**
	Sprung zu der Programmspeicheradresse, die durch das Z-Register adressiert wird. Taktzyklen: 2 Befehlsworte: 1
JMP k	**Jump**
	Sprung zu der Programmspeicheradresse k. Taktzyklen: 3 Befehlsworte: 2
RCALL k	**Relative Call to a Subroutine**
	Aufruf eines Unterprogramms mit relativer Adressierung. Das Programm wird an der Position PC+k+1 fortgesetzt. Die Rücksprungadresse wird auf dem Stack abgelegt. Für k gilt: $-64 \leq k \leq 63$ Taktzyklen[1]: 3 Befehlsworte: 1
RET	**Return from Subroutine**
	Rücksprung aus einem Unterprogramm. Die Rücksprungadresse wird vom Stack gelesen. Taktzyklen[1]: 4 Befehlsworte: 1
RETI	**Return from Interrupt**
	Rücksprung aus einer Interrupt-Service-Routine mit gleichzeitigem Setzen des I-Flags. Die Rücksprungadresse wird vom Stack gelesen. Flags: I Taktzyklen[1]: 2 Befehlsworte: 1
RJMP k	**Relative Jump**
	Das Programm wird an der Position PC+k+1 fortgesetzt. Für k gilt: $-64 \leq k \leq 63$ Taktzyklen: 2 Befehlsworte: 1

Anmerkungen:
[1] Bei AVR-Controllern mit einer Programmspeichergröße von mehr als 128kB dauert die Ausführung des Befehls einen Taktzyklus länger

Tab. 14.7 Überblick über die Skipbefehle

CPSE Rd,Rr	**Compare and Skip if Equal**
	Sind die Werte der beiden Register Rd und Rr identisch, wird der nachfolgende Befehl nicht ausgeführt.
	Taktzyklen: 1/2/3 Befehlsworte: 1
SBRC Rr,b	**Skip if Bit in Register Cleared**
	Ist das Bit b im Register Rr gelöscht, wird der nachfolgende Befehl nicht ausgeführt.
	Taktzyklen: 1/2/3 Befehlsworte: 2
SBRS Rr,b	**Skip if Bit in Register Set**
	Ist das Bit b im Register Rr gesetzt, wird der nachfolgende Befehl nicht ausgeführt.
	Taktzyklen: 1/2/3 Befehlsworte: 1
SBIC K,b	**Skip if Bit in I/O Register Cleared**
	Ist das Bit b im der Adresse K im I/O-Adressraum gelöscht, wird der nachfolgende Befehl nicht ausgeführt.
	Taktzyklen: 1/2/3 Befehlsworte: 1
SBIS K,b	**Skip if Bit in I/O Register Set**
	Ist das Bit b im der Adresse K im I/O-Adressraum gesetzt, wird der nachfolgende Befehl nicht ausgeführt.
	Flags: I Taktzyklen: 1/2/3 Befehlsworte: 1

Tab. 14.8 Überblick über bedingte relative Sprungbefehle

Flag		Bedingter Sprungbefehl	
C	0	BRCC, BRSH	Branch if Carry Cleared, Branch if Same or Higher
	1	BRCS, BRLO	Branch if Carry Set, Branch if Lower
N	0	BRPL	Branch if Plus
	1	BRMI	Branch if Minus
Z	0	BRNE	Branch if Not Equal
	1	BREQ	Branch if Equal
V	0	BRVC	Branch if Overflow Cleared
	1	BRVS	Branch if Overflow Set
S	0	BRGE	Branch if Greater or Equal (signed)
	1	BRLT	Branch if Less Than (signed)
H	0	BRHC	Branch if Half-Carry Cleared
	1	BRHS	Branch if Half-Carry Set
T	0	BRTC	Branch if T-Flag Cleared
	1	BRTS	Branch if T-Flag Set
I	0	BRID	Branch if Interrupt Disabled
	1	BRIE	Branch if Interrupt Enabled

14.5.1.2 Addition und Subtraktion

Im folgenden Programmfragment wird davon ausgegangen, dass die zu addierenden 16 bit breiten Operanden in den Registerpaaren *r25:r24* und *r27:r26* stehen. Die Summe wird im Registerpaar *r25:r24* abgelegt.

```
; Addition zweier 16-Bit-Operanden
add   r24,r26   ; untere 8 Bit der Operanden addieren
adc   r25,r27   ; obere 8 Bit der Operanden addieren
```

Mithilfe des ersten Befehls werden die beiden unteren Bytes der Operanden addiert. Der Übertrag dieser Operation wird im Carry-Flag des Statusregisters gespeichert. Mit dem zweiten Befehl werden die beiden oberen Bytes der Operanden addiert, wobei das im Carry-Flag gespeicherte Übertragsbit durch den Befehl *adc* (add with carry) berücksichtigt wird.

Liegen die Operanden nicht in Registern sondern im Speicher, müssen die Operanden zunächst durch geeignete Load-Befehle in die CPU übertragen werden. Dies kann mit absoluter oder indirekter Adressierung geschehen. Das folgende Beispiel benutzt die absolute Adressierung für den ersten Operanden, während der zweite Operand mit indirekter Adressierung unter Verwendung des Y-Registers (Registerpaar *r29:r28*) in den Prozessor übertragen wird.

```
; Addition zweier 16-Bit-Operanden im Speicher
lds   r24,0x100   ; untere 8 Bit des 1. Operanden holen
lds   r25,0x101   ; obere 8 Bit des 1. Operanden holen
ldi   r28,0x02    ; Adresse des 2. Operanden in das …
ldi   r29,0x01    ; … Y-Register übertragen
ld    r24,Y+      ; unteres Byte des 2. Operanden holen (Adresse: 0x102)
ld    r25,Y       ; oberes Byte des 2. Operanden holen (Adresse: 0x103)
add   r24,r26     ; Addition durchführen
adc   r25,r27
sts   0x100,r24   ; unteres Byte des Ergebnisses speichern
sts   0x101,r25   ; oberes Byte des Ergebnisses speichern
```

Analog zur Addition kann die Subtraktion ausgeführt werden:

```
; Subtraktion zweier 16-Bit-Operanden
sub   r24,r26   ; untere Bytes der Operanden subtrahieren
sbc   r25,r27   ; obere Bytes der Operanden subtrahieren
```

Wie im Fall der Addition wird in diesem Beispiel ein möglicher Übertrag, der sich bei der Subtraktion der unteren Operandenbytes ergibt, durch den Befehl *sbc* (subtract with carry) berücksichtigt. Im Fall der Subtraktion ist dieser Übertrag negativ zu gewichten. Daher wird mithilfe des *sbc*-Befehls von der Differenz der Operanden *r25* und *r27* zusätzlich der Wert des Carry-Flags subtrahiert.

14.5.1.3 Arithmetische und logische Schiebeoperationen

Für logische oder arithmetische Schiebeoperationen stehen die Befehle *lsr, lsl, ror, rol* und *asr* zur Verfügung, die einen Wert um ein Bit nach rechts beziehungsweise links verschieben. Das jeweils „herausgeschobene" Bit, beim Rechtsschieben beispielsweise das unterste Bit des Operanden, wird im Carry-Flag abgelegt. Die Rotationsbefehle *ror* und *rol* übertragen den Wert des Carry-Flags in das frei gewordenen Bit des Arbeitsregisters. In Abb. 14.3 ist die Funktionsweise der Schiebe- und Rotationsbefehle veranschaulicht.

Die Schiebebefehle arbeiten mit 8-Bit-Operanden. Durch mehrfache Anwendung von Schiebebefehlen können auch breitere Operanden verarbeitet werden. Exemplarisch wird dies anhand eines 16-Bit-Wertes gezeigt, welcher in zwei Arbeitsregistern abgelegt ist.

```
; Schieben nach links
lsl   r24   ; unteres Byte schieben
rol   r25   ; oberes Byte schieben
; Schieben nach rechts (logisch bzw. vorzeichenlos)
lsr   r25   ; oberes Byte schieben
ror   r24   ; unteres Byte schieben
```

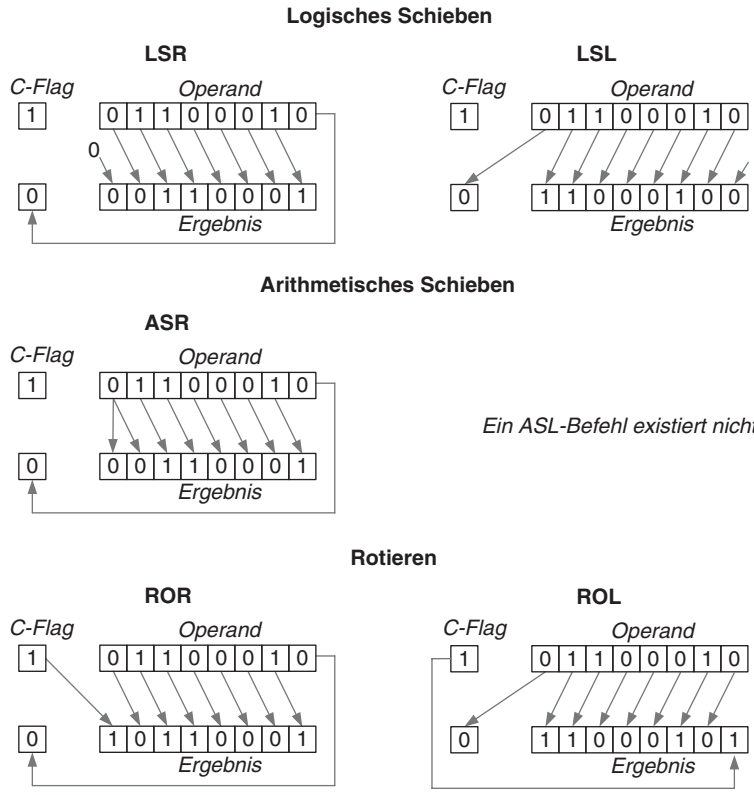


Abb. 14.3 Schiebe- und Rotationsbefehle

Die gezeigten Befehle können auch als arithmetische Schiebeoperation, also eine Multiplikation mit 2 beziehungsweise eine Division durch 2 aufgefasst werden, sofern der Operand als vorzeichenlose Zahl aufgefasst wird. Bei der Schiebeoperation nach links tritt hierbei ein Überlauf auf, wenn das höchstwertige Bit des Operanden gesetzt ist. Soll dieser Fall abgefangen werden, kann nach dem Schieben der Wert des Carry-Flags abgefragt werden. Ist dieses gesetzt, ist ein Überlauf aufgetreten. Im Fall des Rechtsschiebens tritt dagegen nie ein Überlauf auf.

Stellt der Operand dagegen eine Zweierkomplementzahl dar, kann ein Überlauf im Fall des Linksschiebens durch ein gesetztes V-Flag detektiert werden.

Für das arithmetische Rechtsschieben von vorzeichenbehafteten Zahlen muss der Befehl *asr* verwendet werden. Dieser sorgt im Gegensatz zu den anderen Schiebebefehlen dafür, dass das höchstwertige Bit des Operanden in das Ergebnis kopiert wird.

```
; Arithmetisches Schieben nach rechts (vorzeichenbehaftet)
asr   r25   ; oberes Byte schieben
ror   r24   ; unteres Byte schieben
```

14.5.1.4 Multiplikation

Viele der AVR-CPUs unterstützen die Multiplikation zweier 8-Bit-Werte. Das Ergebnis der Multiplikationsbefehle ist ein 16-Bit-Wert, der im Registerpaar *r1:r0* abgelegt wird. Sollen 16-Bit-Operanden multipliziert werden, müssen insgesamt vier Multiplikationsbefehle verwendet werden. Die hierbei entstehenden Teilergebnisse werden anschließend entsprechend ihrem Gewicht addiert. Diese Vorgehensweise wird in Abb. 14.4 verdeutlicht.

Eine entsprechende Umsetzung des Prinzips ist im folgenden Programmfragment dargestellt. Die 16-Bit-Operanden stehen in den Registerpaaren *r25:r24* und *r27:r26*. Das 32 bit breite Ergebnis wird in den Registern *r16* bis *r19* abgelegt, wobei *r16* die untersten 8 Bit und *r19* die obersten 8 Bit des Produktes enthält. Die Register *r17* und *r18* enthalten die Produktbits 15 bis 8 beziehungsweise 23 bis16. Die Berücksichtigung des Gewichts der Teiloperanden erfolgt durch die Auswahl der Produktregister.

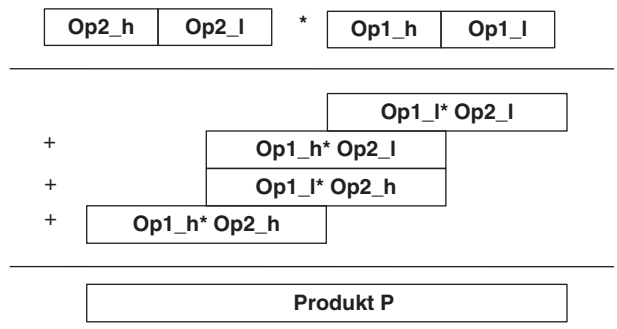


Abb. 14.4 Prinzip der 16x16-Multiplikation mithilfe von 8x8-Multiplikationsbefehlen

```
; Vorzeichenlose 16x16 Multiplikation
clr   r18          ; Produktbits (23:16) auf 0 setzen
clr   r19          ; Produktbits (31:24) auf 0 setzen
mul   r24,r26      ; op1_l * op2_l (1. Teilergebnis mit Gewicht 16:0)
mov   r16,r0       ; Ergebnis in die Produktbits (7:0) …
mov   r17,r1       ; … und (15:8) kopieren
mul   r25,r26      ; op1_h * op2_l (2. Teilergebnis mit Gewicht 24:8)
add   r17,r0       ; Ergebnis zu den Produktbits (15:8) …
adc   r18,r1       ; … und (23:16) addieren
mul   r24,r27      ; op1_l * op2_h (3. Teilergebnis, mit Gewicht 24:8)
add   r17,r0       ; Ergebnis zu den Produktbits (15:8) …
adc   r18,r1       ; … und (23:16) addieren
mul   r24,r27      ; op1_h * op2_h (4. Teilergebnis mit Gewicht 31:16)
add   r18,r0       ; Ergebnis zu den Produktbits (23:16) …
adc   r19,r1       ; … und (31:24) addieren
```

14.5.1.5 Division

Die Division wird von der AVR-CPU nicht durch einen entsprechenden Befehl unterstützt. Stattdessen kann diese Operation mithilfe eines Algorithmus durchgeführt werden, der, wie die schriftliche Division, auf einer sukzessiven Berechnung der Quotientenbits basiert. Abb. 14.5 veranschaulicht das Vorgehen bei einer vorzeichenlosen Division für 8-Bit-Operanden.

Ein entsprechendes AVR-Programm kann wie folgt realisiert werden:

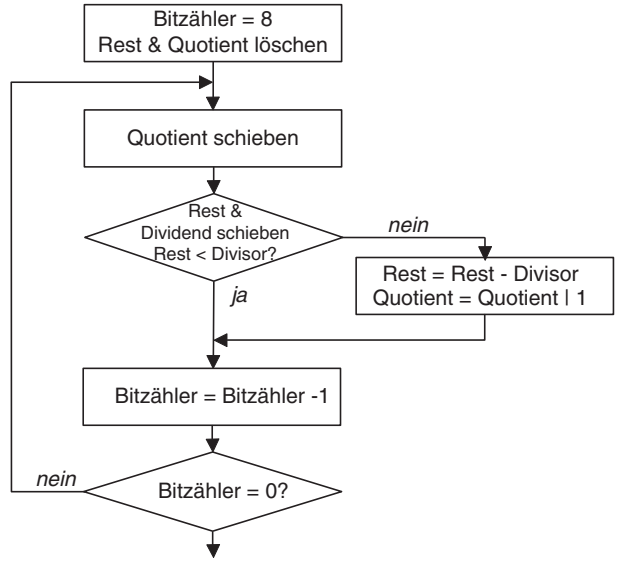


Abb. 14.5 Flussdiagramm für die 8-Bit-Division

```
; Vorzeichenlose 8-Bit-Division
; r16=Bitzähler, r24=Dividend, r25=Divisor
; r26=Rest, r27=Quotient
ldi r16,8          ; Bitzähler = 8
clr r26            ; Rest löschen
clr r27            ; Quotient löschen
div_loop:
   lsl r27         ; Quotient schieben
   lsl r24         ; oberstes Dividendenbit in C-Flag
   rol r26         ; Dividendenbit in Rest verschieben
   cp r26,r25      ; Rest mit Divisor vergleichen
   brcs dec_cnt    ; falls Rest kleiner: springen
   sub r26,r25     ; Divisor von Rest subtrahieren
   ori r27,1       ; Quotientenbit setzen
dec_cnt:
   dec r16         ; Bitzähler dekrementieren
   brne div_loop   ; falls noch nicht 0: nächste Iteration
```

14.5.2 Befehle für den Zugriff auf Speicher und Peripheriekomponenten

In diesem Abschnitt werden die Befehle zum Transfer von Daten zwischen der CPU und dem Speicher beziehungsweise den eingebetteten Peripheriekomponenten näher erläutert. Da die AVR-CPU eine Load-Store-Architektur besitzt, müssen alle Daten, die durch ein Programm verarbeitet werden sollen, zunächst durch einen Ladebefehl *(load)* in ein Arbeitsregister der CPU übertragen werden. Anschließend können die Daten mit arithmetischen oder logischen Befehlen verarbeitet werden. Die Ergebnisse dieser Operationen können für weitere arithmetisch-logische Befehle im Register verbleiben oder werden mithilfe eines Speicherbefehls *(store)* in den Speicher oder die Peripheriekomponenten übertragen

Für die Adressierung des SRAMs im Controller stehen mehrere Load- und Storebefehle zur Verfügung, welche sich durch die verwendete Adressierungsart unterscheiden. Diese Befehle sind in Tab. 14.9 zusammengefasst.

Tab. 14.9 Adressierung des SRAMs über Lade- und Speicherbefehle

Addressierungsart	Load-Befehl	Storebefehl
Absolut	lds	sts
Indirekt	ld	st
Indirekt mit Pre-Drekrement oder Post-Inkrement	ld	st
Indirekt mit Verschiebung	ldd	std
Absolut	lds	sts

Als Zielarbeitsregister der Load-Befehle beziehungsweise Quellarbeitsregister für die Store-Befehle können alle 32 Arbeitsregister der CPU verwendet werden. Für die Adressierung des Speichers im Fall der indirekten Adressierung können nur die Register X, Y und Z verwendet werden. Der Offset bei indirekter Adressierung mit Verschiebung darf nur positiv sein und den Wert 63 nicht überschreiten. Darüber hinaus wird diese Adressierungsart nur für die Register Y und Z unterstützt.

Im Folgenden sind Codebeispiele zur Verwendung der Load- und Storebefehle angegeben. Alle Beispiele führen die gleiche Operation aus: Das Kopieren des Wertes in der SRAM-Speicherstelle 228 in die SRAM-Speicherstelle 254.

```
; Speicherstelle kopieren mit absoluter Adressierung
lds   r7,228      ; Wert aus SRAM laden
sts   254,r7      ; Wert in SRAM speichern
; Speicherstelle kopieren mit indirekter Adressierung
ldi   r30,228     ; Low-Byte des Z-Registers laden
ldi   r31,0       ; High-Byte des Z-Registers laden
ldi   r26,255     ; Low-Byte des X-Registers laden
ldi   r27,0       ; High-Byte des X-Registers laden
ld    r3,Z        ; Wert indirekt aus SRAM laden
st    -X,r3       ; mit Pre-Dekrement speichern
; Kopieren mit indirekter Adressierung mit Verschiebung
ldi   r28,220     ; Low-Byte des Y-Registers laden
ldi   r29,0       ; High-Byte des Y-Registers laden
ldd   r5,Y+8      ; Wert aus SRAM laden
std   Y+34,r5     ; Wert in SRAM speichern
```

Für einen Zugriff auf den Programmspeicher wird der Befehl *lpm* (load from program memory) zur Verfügung gestellt. Dieser Befehl ermöglicht es auf im Programmspeicher abgelegte Konstanten zuzugreifen. Hierbei wird nur die indirekte Adressierung oder die indirekte Adressierung mit Post-Inkrement unterstützt. Wird bei Verwendung des *lpm*-Befehls kein Operand angeben, erfolgt die Berechnung der Programmspeicheradresse mithilfe des Registerpaares *Z* und der gelesene Wert wird im Register *r0* abgespeichert.

```
; Beispiele für die Verwendung des Befehls lpm
lpm               ; r0 mit Wert aus Programmspeicher laden
lpm   r0,Z        ; identisch zu voriger Zeile
lpm   r8,Z        ; hier wird r8 überschrieben
lpm   r17,Z+      ; r17 laden, anschließend Z inkrementieren
```

Der Zugriff auf die Peripheriekomponenten kann sowohl memory-mapped (mithilfe der zuvor beschriebenen Load-/Storebefehle) als auch port-mapped erfolgen. Für einen port-mapped-basierten Zugriff stellt die AVR-CPU die Befehle *in* und *out* zur Verfügung.

Die Befehle *in* und *out* unterstützen nur eine absolute Adressierung. Soll beispielsweise ein Wert auf den digitalen Ausgängen *(PORTA)* eines *ATmega32* ausgegeben werden, kann dies mithilfe des Befehls

```
out   27,r7          ; r7 auf den PORTA-Anschlüssen ausgeben
```

geschehen. Die Adressen der einzelnen Peripheriekomponenten können in den Datenblättern der AVR-Controller nachgeschlagen werden. Allerdings wird der Programmcode durch eine Angabe der Adresse als Zahlenwert wie im obigen Beispiel recht unübersichtlich. Die Firma Atmel stellt daher sowohl für Assembler- als auch für C-Programme Include-Dateien zur Verfügung, in denen symbolische Konstanten für den Zugriff auf die Peripheriekomponenten definiert sind. Hiermit kann das obige Beispiel auch wie folgt formuliert werden.

```
.include "m32def.inc"    ; Include Datei einbinden
out   PORTA,r7           ; r7 auf den PORTA-Anschlüssen ausgeben
```

Mithilfe der Befehle *in* und *out* kann auch auf CPU-interne Register wie das Statusregister oder den Stackpointer zugegriffen werden, wie die folgenden Codebeispiele zeigen:

```
in    r14,SREG       ; Statusregister nach r14 kopieren
in    r30,SPL        ; niederwertiges Byte des SP nach r30
in    r31,SPH        ; höherwertiges Byte des SP nach r31
```

14.5.3 Programmverzweigungen

Im Rahmen dieses Abschnitts wird anhand von einfachen Beispielen verdeutlicht, wie die Sprungbefehle der AVR-CPU eingesetzt werden können. Hierzu wird zunächst auf Programmverzweigungen eingegangen. Anschließend werden der Aufruf von Unterprogrammen und Möglichkeiten der Übergabe von Parametern an Unterprogramme vorgestellt. Eine Einführung in die Verarbeitung von Interrupts schließt den Abschnitt ab.

14.5.3.1 Bedingte Verzweigungen und Programmschleifen

Soll eine bedingte Verzweigung oder eine Programmschleife realisiert werden, können hierzu die bedingten Sprungbefehle oder auch die Skip-Befehle verwendet werden. Exemplarisch kann die Verwendung der AVR-Sprungbefehle anhand zweier einfacher Beispiele verdeutlich werden, die in Abb. 14.6 als Flussdiagramme dargestellt sind.

Die bedingte Ausführung eines Befehls (Abb. 14.6a) lässt sich mithilfe eines Vergleichs und eines bedingten Sprungbefehls realisieren.

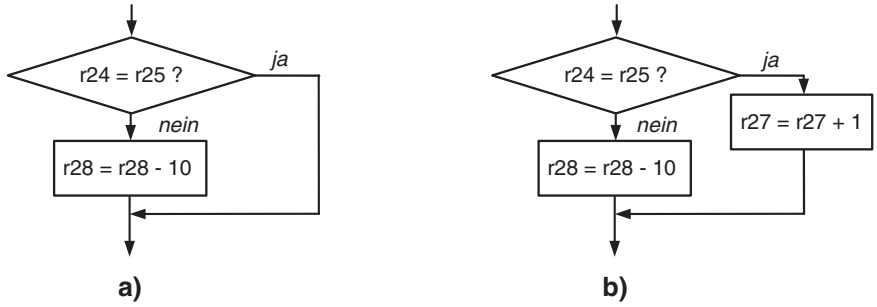


Abb. 14.6 Beispiele für Programmverzeigungen

```
; Verzweigung mit bed. Sprung (if)
cp    r24,r25    ; R24 und R25 vergleichen (Flags wie SUB)
brne  weiter     ; falls ungleich: springen
subi  r28,10     ; Subtraktion ausführen
weiter:
```

Ebenso könnte die gleiche Funktionalität erreicht werden, wenn das Programmfragment mit Skip-Befehlen realisiert würde. Der Skip-Befehl überprüft eine Bedingung (zum Beispiel, ob zwei Register identische Werte besitzen) und überspringt nachfolgenden Befehl falls die Bedingung erfüllt ist. Im obigen Beispiel bietet sich die Verwendung des Befehls *cpse (compare and skip if equal)* an:

```
; Verzweigung mit Skip-Befehl
cpse  r24,r25     ; Vergleich -- Nachf. Befehl evtl. überspringen
subi r28,10       ; wird nicht ausgeführt falls r24=r25
```

Bei diesem Beispiel besitzt der Skip-Befehl gegenüber dem Branch-Befehl Vorteile, da Programmcode eingespart und gleichzeitig die Ausführungszeit des Programmabschnitts um einen Taktzyklus reduziert wird.

Ein Programmfragment, welches das Beispiel aus Abb. 14.6b umsetzt, könnte wie folgt formuliert werden:

```
; Verzweigung mit bed. Sprung (if-else)
cp     r24,r25   ; R24 und R25 vergleichen (Flags wie SUB)
brne  do_inc     ; falls ungleich: springen
subi  r28,10     ; Subtraktion ausführen
rjmp  weiter     ; alternativen Code überspringen
do_inc:
   inc   r27     ; R27 inkrementieren
weiter:
```

Der Einsatz von Skip-Befehlen ist in diesem Beispiel nicht vorteilhafter. Auf ein entsprechendes Beispiel wird daher verzichtet.

14.5.3.2 Unterprogrammaufrufe

Unterprogramme werden mithilfe der Befehle *call*, *rcall* oder *icall* aufgerufen. Während der Befehl *call* die Adresse des Unterprogramms absolut angibt, verwendet der Befehl *rcall* eine relative Adressierung. Da der Befehl *rcall* einen geringeren Programmspeicherbedarf besitzt und schneller ausgeführt wird, ist dieser Befehl in der Regel zu bevorzugen. Hierbei ist jedoch zu berücksichtigen, dass die Differenz der Einsprungadresse des Unterprogramms und des *rcall*-Befehls auf den Wertebereich von −2048 bis +2047 begrenzt ist. Ist die Differenz größer, muss auf den *call*-Befehl zurückgegriffen werden. Der Befehl *icall* ermöglicht die Angabe der Unterprogrammadresse mithilfe des Z-Registers und verwendet somit eine indirekte Adressierung.

Die CPU legt mit dem Unterprogrammaufruf die Rücksprungadresse merken. Dies ist die Programmspeicheradresse, bei der das Programm nach Beenden des Unterprogramms fortgesetzt werden soll. Für die Speicherung der Rücksprungadresse wird der Stack verwendet. Da Programmspeicheradressen im Fall des AVR eine Wortbreite von 16 bit besitzen, werden hierdurch zwei Speicherplätze des Stacks belegt.

Ein Unterprogramm wird mit dem Befehl *ret* beendet. Dieser Befehl lädt die auf dem Stack abgelegte Programmspeicheradresse des aufrufenden Programms in den Program Counter (PC). Der Program Counter adressiert somit anschließend die Befehle des aufrufenden Programms, welches nach Ausführung des *ret*-Befehls fortgesetzt wird.

Werden in einem Unterprogramm Zwischenergebnisse erzeugt, können diese temporär in Arbeitsregistern abgelegt werden. Da die Werte dieser Register durch das Unterprogramm verändert werden, ist es sinnvoll, die betroffenen Arbeitsregister zu Beginn des Unterprogramms auf dem Stack zu sichern. Hierzu wird der Befehl *push* verwendet. Vor Verlassen des Unterprogramms werden die ursprünglichen Werte der Arbeitsregister mithilfe des *pop*-Befehls vom Stack in die Register zurückgeladen. Nach Verlassen des Unterprogramms besitzen somit alle Arbeitsregister den Wert, den sie beim Aufruf des Unterprogramms besaßen.

Die Arbeitsregister können auch zur Übergabe von Parametern oder Rückgabewerten verwendet werden. Ein Beispiel hierfür zeigt das nachfolgende Programm, welches aus einem Hauptprogramm *haupt_prg* und einem Unterprogramm *up_add* besteht.

```
; Beispiel für Unterprogrammaufrufe mit
; registerbasierter Parameterübergabe
haupt_prg:
ldi   r24,42        ; 1. Beispielwert laden
ldi   r25,37        ; 2. Beispielwert laden
rcall up_add        ; Unterprogramm aufrufen
…                   ; weitere Befehle zur Verarbeitung des
```

```
…                   ; Ergebnisses (steht in r16)
up_add:
push  r24           ; r24 auf dem Stack sichern
add   r24,r25       ; Parameter addieren
mov   r16,r24       ; Ergebnis nach r16 schreiben
pop   r24           ; r24 wieder herstellen
ret                 ; Unterprogramm verlassen
```

Als Alternative zur registerbasierten Parameterübergabe kommt auch die Übergabe von Parametern mithilfe des Stacks infrage. Dieses Vorgehen ist insbesondere dann sinnvoll, wenn eine große Anzahl von Parametern an ein Unterprogramm übergeben werden soll. Im folgenden Beispiel wird die Parameterübergabe exemplarisch verdeutlicht.

Das Unterprogramm sichert zunächst die verwendeten Registerwerte auf dem Stack, und es wird der aktuelle Wert des Stackpointers in das Z-Register geladen. Anschließend erfolgt die indirekte Adressierung der Daten mithilfe einer indirekten Adressierung mit Verschiebung. Nach der Verarbeitung der Daten, in diesem Beispiel die Addition der übergebenen Parameter, wird das Ergebnis auf dem Stack gesichert, wobei der erste Übergabeparameter überschrieben wird. Nach dem Wiederherstellen der gesicherten Registerwerte wird das Unterprogramm verlassen.

Das Hauptprogramm stellt den ursprünglichen Wert des Stackpointers nach Rückkehr aus dem Unterprogramm wieder her, indem die zuvor mit push-Befehlen auf dem Stack abgelegten Werte mit zwei *pop*-Befehlen vom Stack entfernt werden. Da der erste Übergabeparameter mit dem Ergebnis des Unterprogramms überschrieben wurde, befindet sich nach Ausführung beider *pop*-Befehle das Ergebnis des Unterprogramms im Arbeitsregister *r24*.

Wird eine Parameterübergabe mithilfe des Stacks durchgeführt, ist es sinnvoll, die Belegung des Stacks tabellarisch festzuhalten. Hierzu wird in einer zweispaltigen Tabelle in der linken Spalte die Adresse der Speicherstelle (relativ zum aktuellen Stackpointer) und in der rechten Spalte der Wert der Speicherstelle eingetragen.

Für die Realisierung des Codes wird angenommen, dass das Hauptprogramm zwei Parameter auf dem Stack ablegt und anschließend in das Unterprogramm verzweigt. Zu Beginn des Unterprogramms werden die Register *r30, r31, r24* und *r25* auf dem Stack gesichert. Die anschließende Belegung des Stacks ist in Tab. 14.10 dargestellt. Anhand der Tabelle kann nachvollzogen werden, dass die Parameter an den Adressen *Stackpointer+7* und *Stackpointer+8* zu finden sind, woraus sich direkt der benötigte Offset für die Verschiebung zur Adressierung der Übergabeparameter ergibt.

Auf Basis der dokumentierten Stackbelegung kann das Programm realisiert werden. Im Folgenden ist der Code für das Hauptprogramm *haupt_prg* und das Unterprogramm *up_add* dargestellt.

Tab. 14.10 Belegung des Stacks für das Beispielprogramm

Adresse	Wert
SP+8	1. Parameter (42)
SP+7	2. Parameter (37)
SP+6	Rücksprungadresse
SP+5	Rücksprungadresse
SP+4	r30
SP+3	r31
SP+2	r24
SP+1	r25
SP	*„unbelegt“*

```
; Beispiel für Unterprogrammaufrufe mit
; stackbasierter Parameterübergabe
haupt_prg:
ldi   r24,42      ; 1. Beispielwert laden
push  r24         ; 1. Wert auf dem Stack ablegen
ldi   r24,37      ; 2. Beispielwert laden
push  r24         ; 2. Wert auf dem Stack ablegen
rcall up_add      ; Unterprogramm aufrufen
pop   r24         ; Stackpointer durch pop-Befehle
pop   r24         ; wieder herstellen
                  ; Das Ergebnis steht nun in r24
…                 ; weitere Befehle zur Verarbeitung des Ergebnisses
up_add:
push  r30         ; r30, r31 (= Z-Register)
push  r31         ; auf dem Stack sichern
push  r24         ; temporär verwendete Register sichern
push  r25
in     r30,SPL    ; untere 8 Bit des Stackpointers nach r30
in     r31,SPL    ; obere 8 Bit des Stackpointers nach r31
ldd   r24,Z+8     ; 1. Wert vom Stack nach r24 kopieren
ldd   r25,Z+7     ; 2. Wert vom Stack nach r25 kopieren
add   r24,r25     ; Parameter addieren
std   Z+8,r24     ; Ergebnis anstelle des 1. Wertes auf dem Stack ablegen
pop   r25         ; gesicherte Register wieder herstellen
pop   r24         ; Aufgrund der Struktur des Stapelspeichers
pop   r31         ; geschieht dies in umgekehrter Reihenfolge
pop   r30         ; (Beispiel: Das zuerst gesicherte Register wird
                  ; zuletzt vom Stack geholt)
ret               ; Unterprogramm verlassen
```

14.6 Grundlagen der Interruptverarbeitung

Eine wichtige Aufgabe eines Mikrorechnersystems ist es, auf Ereignisse reagieren zu können. Derartige Ereignisse können zum Beispiel Eingaben des Benutzers oder auch das Bereitstellen von Daten eines Sensors sein. Ebenso könnten eingebettete Peripheriekomponenten wie Timer oder Kommunikationsschnittstellen Ereignisse auslösen, auf die das Programm reagieren muss. Eine besondere Eigenschaft dieser Ereignisse ist, dass sie asynchron zum laufenden Programm auftreten. Dies heißt, dass man bei der Erstellung eines Programms nicht weiß, welcher Teil des Programms gerade abgearbeitet wird, wenn das Ereignis auftritt.

Es existieren zwei grundlegende Alternativen, um auf diese Ereignisse zu reagieren. Diese Alternativen werden im Folgenden mit dem englischen Fachbegriff *Polling* (deutsch: Abfragen) und als Interruptverarbeitung oder kurz *Interrupts* bezeichnet.

Eine Analogie aus dem täglichen Leben kann helfen, die Grundprinzipien dieser beiden Strategien zu verdeutlichen: Sie haben Gäste eingeladen, wissen aber nicht genau, wann die Gäste erscheinen werden. Zur Bewirtung Ihrer Gäste müssen Sie noch Getränke kalt stellen.

Eine denkbare Strategie wäre es, auf dem Flur der Wohnung im Kreis zu laufen. Jedes Mal bei Erreichen der Wohnungstür wird diese geöffnet, um nachzuschauen, ob die Gäste schon eingetroffen sind. Um die Wartezeit sinnvoller zu nutzen, könnte auch ein Weg durch die Küche gewählt werden, um mit jedem Durchlauf eine Getränkeflasche in den Kühlschrank zu stellen. Diese Vorgehensweise entspricht in etwa dem Prinzip des Pollings: Die Abfrage des Ereignisses („Gäste sind da“) wird wiederholt (in einer Warteschleife) ausgeführt ohne zu wissen, ob das Ereignis wirklich eingetreten ist. Zusätzlich zur Abfrage des Ereignisses kann in der Warteschleife ein Teil der sonst noch anstehenden Aufgaben („Getränke kalt stellen“) abgearbeitet werden.

In der Realität würden die meisten Menschen vermutlich eine andere Strategie wählen, da sie eine Türklingel besitzen: Sie arbeiten die Aufgabe „Getränke kalt stellen“ ab und unterbrechen diese Arbeit sobald die Klingel läutet. Die Gäste werden hereingelassen und die unterbrochene Arbeit wird wieder aufgenommen. Diese Strategie entspricht der Interruptverarbeitung: Das Ereignis („Gäste sind da“) wird durch eine besondere Hardware („Klingel“) signalisiert. Solange das Ereignis nicht eintritt, werden andere Aufgaben abgearbeitet.

Obwohl die oben dargestellte Analogie nicht überstrapaziert werden sollte, kann sie einige Konsequenzen der unterschiedlichen Strategien zur Verarbeitung von Ereignissen verdeutlichen:

- Für die interruptbasierte Verarbeitung wird zusätzliche Hardware benötigt, die zur Unterbrechung einer von der CPU abgearbeiteten Aufgabe führt.
- Potenziell könnte Polling zu kürzeren Reaktionszeiten führen. („Öffnen der Tür genau in dem Moment, in dem die Gäste die Tür erreichen“).

- Treten Ereignisse nur kurzzeitig auf, besteht insbesondere bei Polling die Gefahr, dass diese Ereignisse verpasst werden („die Gäste gehen wieder, weil der Gastgeber gerade zu lange in der Küche beschäftigt ist“)

In den folgenden Abschnitten werden weitere Aspekte der Interruptverarbeitung durch ein Mikrorechnersystem diskutiert.

14.6.1 Interruptfreigabe

In typischen Mikrorechnersystemen können prinzipiell mehrere Ereignisse auftreten, auf die ein Programm reagieren könnte. Nicht alle dieser möglichen Ereignisse sind für eine konkrete Anwendung relevant. Daher ist es sinnvoll, dass nur die relevanten Ereignisse zu einer Unterbrechung des Programms führen.

Die Möglichkeit, dass man festlegt welche Ereignisse zu Programmunterbrechungen führen, wird als Interruptfreigabe bezeichnet.

Die Interruptfreigabe erfolgt häufig hierarchisch. Hierbei kann eine globale und eine lokale Interruptfreigabe unterschieden werden. Die globale Interruptfreigabe dient dazu, Programmunterbrechungen grundsätzlich zuzulassen. Zusätzlich ist es für jedes Ereignis möglich, das Auslösen eines Interrupts zu erlauben oder zu sperren. Erst wenn die globale Interruptfreigabe und die lokale Freigabe mindestens eines Ereignisses erfolgt sind, können Unterbrechungen auftreten. Damit ein Ereignis (beispielsweise eine Flanke an einem Interrupteingang) zu einer Programmunterbrechung führt, muss also sowohl die lokale Freigabe des jeweiligen Ereignisses als auch die globale Interruptfreigabe erfolgt sein.

Die globale Freigabe von Interrupts im Fall der AVR-CPU erfolgt durch das Setzen des Interrupt-Flags (I-Flag) im Statusregister der CPU. Hierfür kann für Assemblerprogramme der Befehl *sei* beziehungsweise für C-Programme die Funktion *sei()* verwendet werden. Das Löschen des Flags, und damit das globale Sperren aller Interrupts, erfolgt mit dem Befehl *cli* oder der C-Funktion *cli()*.

Die lokale Interruptfreigabe erfolgt durch eine entsprechende Programmierung der einzelnen eingebetteten Peripheriekomponenten, die in den nachfolgenden Abschnitten näher vorgestellt werden. Exemplarisch für die lokale Interruptfreigabe werden hier externe Interrupts behandelt.

Die Controller der AVR-Familie besitzen die Möglichkeit einer Programmunterbrechung, wenn ein äußeres Signal einen bestimmten Wert annimmt. Hierzu besitzt beispielsweise der Controller *ATmega32* drei Anschlüsse, die mit *INT0*, *INT1* und *INT2* gekennzeichnet sind. Für die lokale Freigabe der zugehörigen Interrupts besitzt der *ATmega32* das Global Interrupt Control Register *(GICR),* welches die in Tab. 14.11 dargestellte Belegung hat.

Durch Setzen des Bits 7 dieses Registers erfolgt beispielsweise die lokale Freigabe des Interrupts, der dem Controlleranschluss *INT1* zugeordnet ist. Entsprechendes gilt für die Bits 6 und 5, mit denen die Interrupts der Anschlüsse *INT0* bzw. *INT2* freigeschaltet werden können.

Tab. 14.11 Belegung des GICR-Registers

GICR								
Bit	7	6	5	4	3	2	1	0
Name	INT1	INT0	INT2	–	–	–	IVSEL	IVCE

Für die Auswahl, welches konkrete Ereignis (Low-Pegel, High-Pegel, steigende oder fallende Flanke des an dem Anschluss zugeführten Signals) zu einer Programmunterbrechung führt, existieren weitere Register, wie das Microcontroller Unit Control Register *(MCUCR)* und das Microcontroller Unit Control and Status Register *(MCUCSR)*.

14.6.2 Interrupt-Service-Routinen

Nachdem ein freigegebenes Ereignis zur Auslösung einer Unterbrechung geführt hat, muss der Programmteil aufgerufen werden, der zur Verarbeitung dieses Interrupts vorgesehen ist. Dieser Programmteil wird als *Interrupt-Service-Routine (ISR)* bezeichnet.

Die Unterbrechung des laufenden Programms und das Verzweigen in die ISR erfolgen durch die Hardware des Mikrorechners. Daher muss der CPU vor dem Auslösen eines Interrupts bekannt sein, an welcher Position im Programmspeicher die zugehörige ISR zu finden ist. Hierzu wird ein Zeiger auf die entsprechende Programmspeicher-Position benötigt. Dieser Zeiger, welcher auch als *Interrupt-Vektor* bezeichnet wird, kann bereits mit dem Entwurf des Prozessors festgelegt werden. In diesem Fall liegt die Position der ISR fest und kann nicht nachträglich modifiziert werden. Alternativ finden auch programmierbare Interrupt-Vektoren Anwendung. In diesem Fall kann durch eine entsprechende Programmierung die Einsprungadresse der ISR durch das Programm bestimmt werden.

Im Fall der AVR-CPU wird der erste Weg beschritten, wobei die Interrupt-Service-Routinen in den ersten Speicherstellen des Programmspeichers abgelegt werden. Für jede ISR sind im unteren Teil des Programmspeichers zwei Programmspeicherworte reserviert. Dieser Speicherplatz reicht natürlich nicht für die Aufnahme einer kompletten ISR aus. Allerdings ist der reservierte Bereich ausreichend, um einen Sprungbefehl (zum Beispiel mithilfe des *jmp*-Befehls) aufzunehmen, mit welcher der Code der eigentlichen ISR aufgerufen wird.

In den Datenblättern der AVR-Controller ist die Zuordnung von Ereignissen und Interrupt-Vektoren zu finden. Exemplarisch fasst Tab. 14.12 die Interrupt-Vektoren des *ATmega32* zusammen.

Bei einem Ereignis an *INT1* springt ein Interrupt beispielsweise an Adresse 4 und dort wird ein Sprung in die ISR hinterlegt. Da die Sprungbefehle immer 16 Bit des Programmspeichers belegen, beginnen alle Interruptvektoren an geraden Programmspeicheradressen.

Die folgenden Beispielprogramme verdeutlichen die Verwendung von Interrupt-Service-Routinen in Assembler beziehungsweise C. Um die Programme möglichst

Tab. 14.12 Zuordnung von Ereignissen und Interruptvektoren

Interrupt-vektor (hex.)	Interruptquelle		
	Kurzbezeichnung	Erläuterungen	Gruppe
00	RESET	Reset des Systems (nicht sperrbar)	Reset
02	INT0	Ereignis an Anschluss INT0	Externe Interruptquellen
04	INT1	Ereignis an Anschluss INT1	
06	INT2	Ereignis an Anschluss INT2	
08	TIMER2_COMP	Timer2-Vergleichs-Interrupt	Timer
0A	TIMER2_OVF	Timer2-Überlauf-Interrupt	
0C	TIMER1_CAPT	Timer1-Capture-Interrupt (ICU)	
0E	TIMER1_COMPA	Timer1-Vergleichs-Interrupt A	
10	TIMER1_COMPB	Timer1-Vergleichs-Interrupt B	
12	TIMER1_OVF	Timer1-Überlauf-Interrupt	
14	TIMER0_COMP	Timer0-Vergleichs-Interrupt	
16	TIMER0_OVF	Timer0-Überlauf-Interrupt	
18	SPI_STC	SPI: Übertragung abgeschlossen	Eingebettete Schnittstellen
1A	USART_RXC	USART: Datenempfang abgeschlossen	
1C	USART_UDRE	USART: Sendedatenspeicher leer	
1E	USART_TXC	USART: Senden eines Datums abgeschlossen	
20	ADC	Umsetzung des Analogwertes fertig	
22	EE_RDY	EEPROM-Bereit-Interrupt	EEPROM
24	ANA_COMP	Analog-Komparator	Eingebettete Schnittstellen
26	TWI	I^2C/TWI-Interrupt	
28	SPM_RDY	Programmspeicher-Interrupt	Programmspeicher

übersichtlich zu halten, beschränkt sich die Aufgabe der ISR auf das Zählen der steigenden Flanken am Controller-Anschluss *INT1*.

In diesem Programm werden einige Assembler-Direktiven verwendet, die an dem vorangestellten Punkt zu erkennen sind. Assembler-Direktiven sind Anweisungen, die während der Übersetzung des Programms ausgewertet werden. Sie werden nicht in ausführbare Befehle umgesetzt und belegen daher auch keinen Platz im Programmspeicher. Die Direktive *.org* bewirkt, dass nachfolgende Befehle an einer definierten Position im Programmspeicher abgelegt werden. Im obigen Beispiel wird sie verwendet, um den nachfolgenden Befehl *(jmp isr_int1)* an die Adresse des Interruptvektors (0x04) abzulegen. Nach Einsatz der Direktive *.dseg* beziehen sich alle Befehle auf das SRAM. Die im Beispielprogramm angegebene Folge aus den Direktiven *.dseg, .org* und *.db* dienen dazu, im SRAM ein Byte an der Adresse 0x200 zu reservieren. Mithilfe des Labels *icnt* kann

auf dieses Byte ähnlich wie auf eine Variable in einem Hochsprachenprogramm zugegriffen werden.

```
; Beispiel für Unterprogrammaufrufe mit Interrupt-Service-Routinen
; Beispiel für die Interruptverarbeitung in Assembler
; Zählen der steigenden Flanken am Anschluss INT1
.include "m32def.inc" ; Controllerspezifische Definitionen einbinden
jmp   main            ; nach Reset: Sprung ins Hauptprogramm
.org  0x04            ; Assemblierung bei INT-Vektor fortsetzen
jmp   isr_int1        ; Sprung in eigentliche ISR für INT1
…                     ; hier möglicherweise weitere ISRs
…                     ; oder Unterprogramme
isr_int1:
push  r24             ; Register auf Stack retten
lds   r24,icnt        ; aktuellen Zählerwert holen
inc   r24             ; Zähler inkrementieren
sts   icnt,r24        ; in SRAM abspeichern
reti                  ; ISR verlassen
main:
; Initialisierung
clr   r24             ; r24 auf null setzen
sts   icnt,r24        ; Zählvariable löschen
lds   r16,MCUCR
ori   r16,(1<<ISC10)  ; INT1 so programmieren, dass eine Unter-
ori   r16,(1<<ISC11)  ; brechung mit einer steig. Flanke auftritt
sts   MCUCR,r16
lds   r16, GIFR
ori   r16,(1<< INT1)  ; lokale Interruptfreigabe für INT1
sts   MCUCR,r16
sei                   ; globale Interruptfreigabe
; Endlosschleife des Hauptprogramms
main_lp:
lds  r24,icnt         ; aktuellen Zähler nach r24
call  ausgabe         ; Wert ausgeben -- Das Unterprogramm
                      ; „ausgabe" ist hier nicht angegeben
rjmp  main_lp         ; Endlosschleife des Hauptprogramms
.dseg                 ; Assemblierung auf SRAM umschalten
(„Datensegment")
.org  0x200           ; SRAM-Adresse auswählen
icnt:
.db  0                ; 1 Byte reservieren
```

Ein äquivalentes Programm kann auch in der Sprache C formuliert werden. Hierzu werden sowohl die controller-spezifischen Definitionen aus der Include-Datei *io.h* als auch die speziellen Definitionen für Interruptverarbeitung *(interrupt.h)* eingebunden.

In der Include-Datei *interrupt.h* sind unter anderem auch Makros definiert, die zu einer einfachen Definition von ISRs verwendet werden können. Hierzu wird das Makro *ISR()* aufgerufen. Als Parameter wird der zugehörige Interruptvektor verwendet, der durch die Kurzbezeichnung des Vektors mit nachgestelltem *„_vect“* gekennzeichnet wird.

```
// Programmbeispiel zur Verwendung von ISRs in C
#include <avr/io.h>          // Controller-spezifische Definitionen
#include <avr/interrupt.h>   // Header-Datei für Interrupts
volatile unsigned char icnt;
// Hauptprogramm
void main()
{
   // Initialisierung
   icnt = 0;
   // Interrupt bei steigender Flanke an INT1
   MCUCR |= (1<<ISC11)| (1<<ISC10);
   GICR |= 1<<INT1;     // Lokale Interruptfreigabe
   sei();               // Globale Interruptfreigabe
   while (1) {          // Endlosschleife
        Ausgabe(icnt);
   }
}
// INT1 ISR
ISR (INT1_vect)
{
   icnt++;             // Diese ISR inkrementiert icnt
}
```

Insbesondere bei Verwendung der Hochsprache C wird deutlich, dass Interrupt-Service-Routinen grundsätzlich keine Argumente und auch keine Rückgabewerte besitzen. Diese Eigenschaft ergibt sich aus der Tatsache, dass man nicht wissen kann, welcher Programmteil gerade ausgeführt wird, wenn eine ISR aufgerufen wird. Somit können auch keine Parameter in Arbeitsregistern oder auf dem Stack abgelegt werden, die dann von einer ISR verarbeitet werden könnten. Die einzige Möglichkeit eine Kommunikation zwischen Hauptprogramm und ISR zu realisieren, ist die Verwendung gemeinsamer Speicherplätze, zum Beispiel im SRAM.

14.7 Eingebettete Peripheriekomponenten

Mikrocontroller sind universell einsetzbare digitale Systeme, die auf einem Chip integriert sind. Neben einer CPU enthalten Sie eine Vielzahl von verschiedenen Peripheriekomponenten für sehr unterschiedliche Aufgaben. Die Hersteller von Mikrocontrollern

bieten meist eine große Anzahl von unterschiedlichen Mikrocontrollern an, die sich auch im Hinblick auf die in dem System eingebetteten Komponenten unterscheiden.

Im Folgenden werden am Beispiel der Mikrocontroller der AVR-Familie typische Peripheriekomponenten und ihre Programmierung vorgestellt. Hierbei wird der AVR-Mikrocontroller *ATmega32* zugrunde gelegt. Anhand dieses Beispielcontrollers wird die Funktionsweise ausgewählter Peripheriekomponenten diskutiert. Auf diese Weise werden konkrete Kenntnisse der AVR-Mikrocontrollerserie vermittelt, die es ermöglichen, einfache AVR-basierte Systeme zu realisieren. Außerdem werden Sie in die Lage versetzt, die anhand der AVR-Serie vermittelten Grundprinzipien auf andere eingebettete digitale Systeme zu übertragen.

14.7.1 Ports

Jeder Mikrocontroller besitzt sogenannte *Ports*. Ports sind Anschlüsse des Mikrocontrollers, die durch eine entsprechende Programmierung als digitale Eingänge oder Ausgänge verwendet werden können.

Häufig werden die einzelnen Anschlüsse zu Gruppen zusammengefasst und erhalten einen logischen Namen, der sowohl im Datenblatt referenziert als auch im Rahmen der Softwareentwicklung in den Programmen verwendet wird. So besitzt der *ATmega32* beispielsweise vier Ports, die mit *PORTA, PORTB, PORTC* und *PORTD* bezeichnet werden. Jedem dieser Ports sind acht Anschlüsse des Controllers zugeordnet. Die Portanschlüsse des Controllers werden durch eine entsprechende Nummerierung unterschieden. So werden beispielsweise die acht Anschlüsse des Ports *PORTA* als *PA0* bis *PA7* bezeichnet. Für die anderen Ports gelten entsprechende Zuordnungen.

Um eine hohe Flexibilität beim Einsatz der Ports zu erzielen, ist es möglich, jeden einzelnen Anschluss eines Ports, unabhängig von den anderen Anschlüssen dieses Ports, als Ausgang oder Eingang zu programmieren.

Ist ein Portanschluss als Ausgang konfiguriert, wird durch das laufende Programm festgelegt, ob an diesem Anschluss eine logische 0 oder 1 ausgegeben wird. Entsprechend kann mithilfe eines als Eingang programmierten Ports ein digitaler Wert eingelesen und durch die Software des Controllers ausgewertet werden.

Ports stellen somit die universellste Peripheriekomponente dar, da sie für die Verbindung eines Mikrocontrollers mit beliebigen anderen digitalen Bausteinen eingesetzt werden können. Aus diesem Grund wird statt des Begriffs *Port* häufig auch der Begriff *General Purpose Input/Output (GPIO)* verwendet.

Die Grenzen der Einsetzbarkeit von Ports wird im Wesentlichen durch die Leistungsfähigkeit der CPU des Controllers bestimmt: Je häufiger ein Portbit pro Zeiteinheit umprogrammiert werden muss, desto höher ist die hierfür benötigte Rechenleistung. Im ungünstigsten Fall übersteigt die zur Bedienung der Ports benötigte Rechenleistung die durch die CPU zur Verfügung gestellte Rechenleistung, sodass eine konkrete Aufgabe, wie die Kommunikation mit einem anderen Baustein, nicht realisiert werden kann. Es

muss daher im Einzelfall geprüft werden, ob eine angestrebte digitale Ein-/Ausgabefunktion durch eine entsprechende Portprogrammierung erfolgen kann, oder ob der Einsatz eines Controllers sinnvoll ist, der die gewünschte Funktion durch spezialisierte Hardware-Komponenten zur Verfügung stellt.

Zur Programmierung von Peripheriekomponenten werden sogenannte I/O-Register verwendet. Entsprechend der Grundfunktion eines Ports müssen mindestens zwei I/O-Register vorhanden sein: Ein Register zur Auswahl, ob ein Anschluss als Eingang oder als Ausgang betrieben werden soll und ein weiteres Register, welches zum Einlesen oder Ausgeben der eigentlichen Daten dient.

Entsprechend ihrer Funktion findet man in nahezu allen Mikrocontrollern zur Programmierung von Ports sogenannte *Datenrichtungsregister* und *Datenregister*. Mithilfe des Datenrichtungsregisters wird die *Datenrichtung,* also ob ein Portbit als Eingang oder als Ausgang arbeitet, programmiert. Die Datenregister dienen der eigentlichen Ein-/Ausgabe digitaler Werte. Darüber hinaus können einem Port weitere I/O-Register zugeordnet sein, mit denen spezielle Portfunktionen aktiviert werden können.

Die im Rahmen dieses Kapitels betrachtete AVR-Familie ordnet jedem Port drei I/O-Register zu:

Datenrichtungsregister (Data Direction Register, DDR)
Wird ein Bit im Datenrichtungsregister auf 0 gesetzt, arbeitet der zugehörige Anschluss als Eingang. Ist das dem Anschluss zugehörige Bit dagegen auf 1 gesetzt, wird der entsprechende Anschluss als digitaler Ausgang betrieben.

Dateneingangsregister (Port Input Register, PIN)
Mithilfe dieses Registers können die an einem Port anliegenden digitalen Eingangswerte eingelesen werden.

Datenausgaberegister (Port Output Register, PORT)
Ist ein Portanschluss als Ausgang programmiert, kann mithilfe des PORT-Registers der ausgegebene logische Wert festgelegt werden. Ist ein Portanschluss als Ausgang programmiert, wird durch Setzen des zugehörigen Bits des PORT-Registers eine 1 oder durch Löschen des Bits eine logische 0 ausgegeben.

Wird ein Portanschluss als Eingang verwendet, kann mithilfe des PORT-Registers ein sogenannter Pull-up-Widerstand durch Setzen des Portbits aktiviert werden. Der Porteingang wird dann über einen Widerstand (im Bereich mehrerer Kiloohm) mit der Versorgungsspannung verbunden. Auf diese Weise liegt an dem Eingang eine „schwache Eins“ an, die durch die äußere Beschaltung mit einer „starken Null“, also einer relativ niederohmigen Verbindung zu Masse, überschrieben werden kann.

Ist das zugehörige Bit im PORT-Register gelöscht, arbeitet der Eingang in einem hochohmigen Modus (s. Tab. 14.13).

Die Portprogrammierung kann anhand eines einfachen Schaltungsbeispiels verdeutlicht werden: An einen *ATmega32*-Controller ist ein Taster und eine LED mit

Vorwiderstand anschlossen. Der Taster ist mit dem Portanschluss *PA2* und die LED mit dem Anschluss *PA6* verbunden. Ein entsprechender Schaltplan ist in Abb. 14.7 dargestellt. Die Aufgabe des Controllers besteht darin, die LED einzuschalten, wenn der Taster gedrückt wird.

Tab. 14.13 Funktionen der Portanschlüsse bei AVR-Mikrocontrollern

Bit im IO-Register		Funktion des Portanschlusses
DDR	PORT	
0	0	Eingang, hochohmig
0	1	Eingang, Pull-up-Widerstand aktiviert
1	0	Ausgang, Ausgabe einer 0
1	1	Ausgang, Ausgabe einer 1

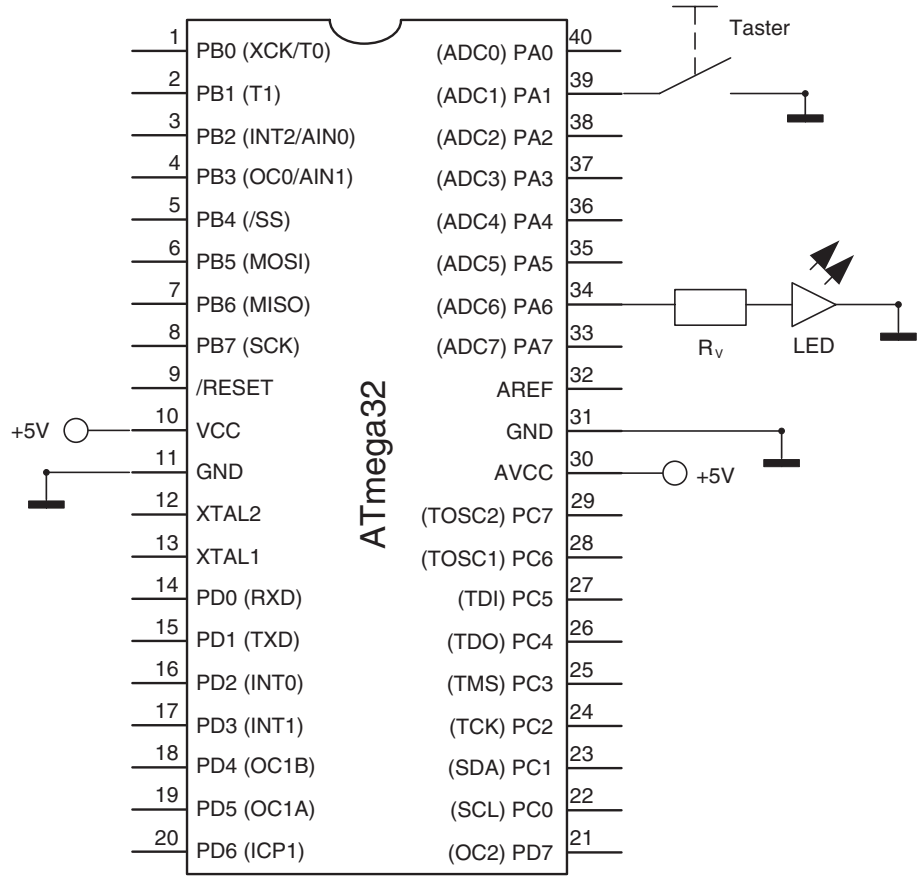


Abb. 14.7 Einfache Mikrocontrolleranwendung mit LED und Taster

Aufgrund der Beschaltung muss der Anschluss *PA1* als Eingang und der Anschluss *PA6* als Ausgang betrieben werden. Über die Verwendung der anderen Anschlüsse des Ports *PORTA* wurde keine Aussage getroffen.

```
#define __AVR_ATmega32__ // Auswahl des Controllers
#include <avr/io.h> // Definitionen etc. für den Controller einbinden
void main()
{
   // Initialisierung
   DDRA  |= 0x40; // Bit 6 im Datenrichtungsregister A setzen
   DDRA  &= 0xFD; // Bit 1 im Datenrichtungsregister A löschen
   PORTA |= 0x02; // Bit 1 im PORTA-Register setzen und so den
                  // internen Pull-Up-Widerstand aktivieren

   // Endlosschleife
   while (1) {
        if (PINA & 0x02)      // PA2 = 1 => Taster nicht gedrückt
            PORTA &= 0xBF;    // PORTA(6) löschen (LED aus)
        else                  // PA2 = 0 => Taster gedrückt
            PORTA |= 0x40;    // PORTA(6) setzen (LED an)
   }
}
```

Neben der Portprogrammierung verdeutlicht dieses einfache Beispielprogramm einige weitere wichtige Aspekte der Programmierung von Mikrocontrollern der AVR-Serie. Im Folgenden werden die einzelnen Zeilen des Programms näher erläutert:

```
#define __AVR_ATmega32__ // Auswahl des Controllers
```

Mithilfe dieser Zeile wird der Controller ausgewählt, für den das Programm geschrieben wird. Bei Verwendung einer Entwicklungsumgebung wie *Atmel Studio* geschieht diese Auswahl in der Regel über die Einstellungen des in der Entwicklungsumgebung angelegten Projektes. In diesem Fall kann die explizite Auswahl des Controllers im Programm entfallen.

```
#include <avr/io.h> // Definitionen etc. für den Controller einbinden
```

Diese Zeile inkludiert eine Header-Datei, welche unter anderem die Definitionen der im Controller vorhandenen I/O-Register enthält. Anschließend kann auf die Register des Controllers wie auf Variablen eines C-Programms zugegriffen werden, was die Programmierung und die Lesbarkeit des Programms wesentlich vereinfacht.

```
void main()
```

Diese Zeile leitet das Hauptprogramm ein und entspricht der üblichen Programmierung in der Programmiersprache C. Anders als bei manchen Programmen, die für einen PC entwickelt werden, besitzt das Hauptprogramm keine Argumente und keine Rückgabewerte. Da das Hauptprogramm bei einfachen Controllern häufig direkt nach dem Einschalten des Systems, ohne Zuhilfenahme eines Betriebssystems gestartet wird, existiert kein aufrufendes Programm (zum Beispiel das Betriebssystem), welches Argumente übergeben könnte oder Rückgabewerte erwartet.

Manchmal wird dennoch für das Hauptprogramm eines AVRs ein Rückgabewert angegeben. Der Grund hierfür ist in dem verwendeten Compiler zu suchen, welcher eine Warnmeldung ausgibt, falls das Hauptprogramm keinen Rückgabewert besitzt. Diese Warnmeldung kann durch die Definition eines Rückgabewertes vermieden werden.

```
// Initialisierung
DDRA  |= 0x40; // Bit 6 im Datenrichtungsregister A setzen
DDRA  &= 0xFD; // Bit 1 im Datenrichtungsregister A löschen
PORTA |= 0x02; // Bit 1 im PORTA-Register setzen und so den
               // internen Pull-Up-Widerstand aktivieren
```

Die korrekte Programmierung des Datenrichtungsregisters des verwendeten Ports geschieht mithilfe dieser Zeilen. In der ersten Zeile wird der aktuelle Wert des Datenrichtungsregisters gelesen und mithilfe einer bitweisen ODER-Verknüpfung mit der hexadezimalen Konstanten 0x40 (binär: 0100 0000) verknüpft. Das Ergebnis der Verknüpfung wird im Datenrichtungsregister des Ports A, *DDRA,* abgelegt. Durch diese Form der Programmierung des Datenrichtungsregisters wird sichergestellt, dass das Bit 6 des Datenrichtungsregisters *DDRA,* welches die Datenrichtung des Portanschlusses *PA6* festlegt, auf 1 gesetzt wird. Alle anderen Bits des Datenrichtungsregisters behalten ihren Wert. Analog wird in der zweiten Zeile das Löschen des Bits 1 im Datenrichtungsregister durch die Verwendung einer bitweisen UND-Verknüpfung durchgeführt.

Da bei geöffnetem Taster kein eindeutiger logischer Pegel an dem Anschluss *PA1* anliegt, wird mithilfe der dritten Zeile der interne Pull-Up-Widerstand dieses Portanschlusses aktiviert. Bei geöffnetem Taster würde an dem Portanschluss über den Pull-Up-Widerstand eine logische 1 anliegen, während bei gedrücktem Taster eine logische 0 anliegt.

```
while (1)
```

Mit dem Setzen der für die Anwendung relevanten Bits der Register *DDRA* und *PORTA* ist die Initialisierung für dieses einfache Programmbeispiel abgeschlossen und es folgt der Code für den normalen Betrieb des Controllers. Dieser wird in den meisten Fällen in eine Endlosschleife eingebettet, da das Programm kontinuierlich auf Eingaben reagieren soll. Würde man auf die Endlosschleife verzichten, würde das Programm bereits nach wenigen Taktzyklen beendet sein. In diesem Fall wird in eine vom Compiler

erzeugte leere Endlosschleife verzweigt. Die gewünschte Reaktion des Controllers auf den Tastendruck würde also nur einmalig, kurz nach dem Einschalten des Controllers erfolgen.

```
if (PINA & 0x02)  // PA2 = 1 => Taster nicht gedrückt
```

Mithilfe dieser Zeile wird der Anschluss *PA2* des Controllers abgefragt. Der Lesezugriff auf *PINA* liefert den momentanen Wert aller Portanschlüsse des Ports *PA* zurück. Von diesen 8 Bit ist jedoch nur eines für die Ausführung der gewünschten Funktion relevant. Daher werden die nicht relevanten Bits durch die UND-Verknüpfung mit der Konstanten 0x02 (binär: 0000 0010) ausgeblendet, und es ergeben sich zwei mögliche Werte für den Ausdruck *PINA&0x02*: Liegt an dem Anschluss *PA2* eine logische 1 an (Taster offen), ergibt der Ausdruck den Wert 2. Ist der Taster gedrückt und liegt eine logische 0 am Anschluss *PA2* an, ergibt der Ausdruck den Wert 0. Da in der Programmiersprache C alle Ausdrücke, die einen Wert ungleich Null ergeben, als wahr interpretiert werden, kann der Ausdruck *PINA&0x02* zur Auswahl herangezogen werden. Ist der Taster nicht gedrückt (Ausdruck ungleich Null), würde der If-Zweig ausgeführt werden. Im anderen Fall der Else-Zweig.

```
PORTA &= 0xBF;      // PORTA(6) löschen (LED aus)
PORTA |= 0x40;      // PORTA(6) setzen (LED an)
```

Diese beiden Zeilen setzen beziehungsweise löschen das Bit 6 des I/O-Registers PORTA. Ist das Bit gelöscht, liegt an dem Portanschluss eine niedrige Spannung (nahe 0 V) an und über die LED fällt keine Spannung ab; die LED leuchtet nicht. Ist das Bit dagegen gesetzt, wird am Anschluss eine hohe Spannung (nahe der Versorgungsspannung des Controllers) ausgegeben und die LED leuchtet.

Die Programmierung der Ports kann ebenso in Assembler erfolgen. Ein Programm, welches die oben beschriebene Funktion ausführt, könnte wie folgt realisiert werden:

```
.include "m32def.inc"
in    r24,DDRA     ; Aktuellen Wert des DDRA-Registers holen
ori   r24,0x40     ; relevante Bits setzen
andi  r24,2        ; bzw. löschen
out   DDRA,r24     ; Ergebnis abspeichern
in    r24,PORTA    ; PORTA holen
ori   r24,0x2      ; Pull-Up für Eingang PA1 aktivieren
out   PORTA,r24    ; PORTA setzen
main_loop:
in    r16,PINA     ; Eingabewert holen
andi  r16,0x02     ; Bit 1 maskieren
breq  led_on       ; falls Ergebnis = 0, springen
in    r24,PORTA
```

```
andi   r24,0xBF
out    PORTA,r24    ; LED aus
rjmp   main_loop
led_on:
in     r24,PORTA
ori    r24,0x40
out    PORTA,r24    ; LED an
rjmp   main_loop
```

Den Portanschlüssen eines Mikrocontrollers können neben der softwaregesteuerten Ein-/Ausgabe digitaler Daten auch andere Funktionen zugeordnet werden. Die entsprechenden alternativen Portfunktionen (engl. *alternate port functions*) werden in den Anschlussdiagrammen des Controllers häufig in Klammern angegeben. So können die Anschlüsse *PA0* bis *PA7* eines *ATmega32* beispielsweise als analoge Eingänge verwendet werden. Andere Anschlüsse wie *PD0, PD1* oder *PC0, PC1* können dagegen mit eingebetteten Peripheriekomponenten zur seriellen Datenübertragung verbunden werden. Ist eine alternative Portfunktion aktiviert worden, ist die ursprüngliche Portfunktion in der Regel nicht mehr zugänglich, da die Peripheriekomponente die Steuerung der Anschlüsse übernimmt. Durch die Mehrfachbelegung der Anschlüsse eines Mikrocontrollers wird erreicht, dass die Anzahl der Anschlüsse gering gehalten wird. Der Controller kann somit in kleinere Gehäuse mit relativ wenigen Anschlüssen eingebaut werden, was neben der kleineren Bauform auch zu einer Verringerung der Herstellungskosten beiträgt. In den folgenden Abschnitten werden einige der wichtigsten Peripheriekomponenten anhand des Beispiels eines *ATmega32* beschrieben. Die zugehörigen alternativen Portfunktionen werden in Zusammenhang mit der jeweiligen Peripheriekomponente beschrieben.

14.7.2 Timer

Timer sind ebenso wie die zuvor beschriebenen Ports Standardkomponenten eines Mikrocontrollers. Sie können für sehr unterschiedliche Aufgaben eingesetzt werden. Hierzu zählen unter anderem die Erzeugung von Signalen, die zeitliche Vermessung von Signalen (zum Beispiel Frequenzzähler) oder auch die regelmäßige Unterbrechung des CPU-Programms durch Interrupts.

Die Kernkomponente eines Timers ist ein digitaler Zähler, der häufig eine Wortbreite von 8, 16 oder 32 bit besitzt. Der Zähler wird entweder mit einem aus dem Systemtakt abgeleiteten Takt oder mit einem von außen angelegten Taktsignal betrieben (Abb. 14.8). Der aktuelle Zählerstand wird durch eine nachgeschaltete Einheit ausgewertet, welche in Abhängigkeit vom Zählerstand ein *Timer-Ereignis* auslösen kann. Auf Basis des Timer-Ereignisses können weitere Aktionen abgeleitet werden. Dies kann zum Beispiel das Invertieren eines Mikrocontroller-Anschlusses oder die Unterbrechung des laufenden Programms durch einen Interrupt sein.

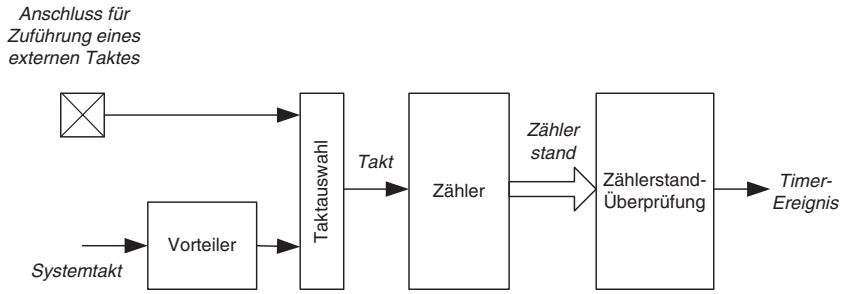


Abb. 14.8 Prinzipieller Aufbau eines einfachen Timers

Auf den ersten Blick mag die Kernfunktion eines Timers simpel erscheinen, sodass die Schlussfolgerung naheliegen könnte, dass die Funktion eines Timers auch mithilfe der CPU durch ein entsprechendes Programm nachgebildet werden kann. Diese Schlussfolgerung würde jedoch vernachlässigen, dass die CPU die wertvollste und universell einsetzbarste Komponente ist. Es ist daher nicht sinnvoll, diese Komponente für eine so einfache Aufgabe wie das Zählen von Impulsen einzusetzen, da die hierfür benötigte Rechenleistung nicht mehr für andere Aufgaben genutzt werden kann. Neben der Entlastung der CPU bietet die Auslagerung häufig genutzter Funktionen in eine eigenständige Hardwarekomponente einen weiteren entscheidenden Vorteil: Wird eine Funktion in Form einer spezialisierten Hardware implementiert, kann die Implementierung so erfolgen, dass die Ausführung dieser Funktion in wenigen Taktzyklen (häufig in einem einzelnen Taktzyklus) erfolgt. Eine entsprechende Realisierung als CPU-Programm benötigt dagegen in der Regel eine deutlich höhere Ausführungszeit, was sich insbesondere dann negativ auswirken würde, wenn auf äußere Ereignisse, wie zum Beispiel der Wechsel des Wertes eines Eingangssignals reagiert werden muss. Diese Tatsache wird in den folgenden Abschnitten am Beispiel der Funktion eines Timers verdeutlicht.

Die im Rahmen dieses Kapitels betrachteten Timer des Mikrocontrollers *ATmega32* können in verschiedenen Modi betrieben werden. Die Modi werden als der „Normale Modus (normal mode)“, der „CTC-Modus“ sowie als „PWM-Modi“ bezeichnet. In den folgenden Abschnitten werden diese Modi näher beschrieben.

14.7.2.1 Normal Mode

Der als *Normal Mode* bezeichnete Modus eines AVR-Timers stellt den einfachsten Betriebsmodus dar. In diesem Modus zählt der Zähler des Timers nur aufwärts. Bei Erreichen des Zähler-Endwertes (zum Beispiel 255 bei einem 8-Bit-Timer) wird der Zählerstand auf 0 gesetzt und erneut aufwärts gezählt. Das Erreichen des Endwertes stellt ein Ereignis dar, welches zum Beispiel zur Invertierung eines Ausgangssignals verwendet werden kann. Das zugehörige zeitliche Verhalten des Zählerstandes und des Ausgangssignals ist in Abb. 14.9 dargestellt.

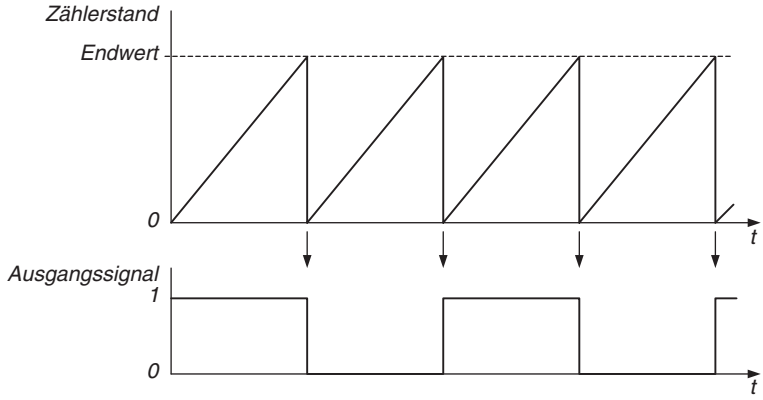


Abb. 14.9 Verlauf des Zählerstandes und eines Ausgangssignals im Normal Mode

Wird der Timer mit der Taktfrequenz des Controllers f_{sys} betrieben, ergibt sich die Rate der Überlaufereignisse R_{OV} beziehungsweise die Frequenz des erzeugten Signals f_{out} zu:

$$R_{OV} = \frac{f_{\text{sys}}}{256}$$

beziehungsweise

$$f_{\text{out}} = \frac{R_{OV}}{2} = \frac{f_{\text{sys}}}{256}$$

Die Überlaufrate R_{OV} hängt in diesem Fall nur von der Systemtaktfrequenz ab. Für eine grobe Einstellung der Überlaufrate wird bei Timern in der Regel ein Vorteiler eingesetzt. Im Fall des *ATmega32* kann der Vorteiler des Timers auf fünf verschiedene Werte zwischen 1 (keine Vorteilung) und 1024 (der Zähler des Timers wird mit 1/1024 der Systemfrequenz betrieben) eingestellt werden. Für die Einstellung des Vorteilers werden I/O-Register (Timer/Counter Control Register, *TCCR*) zur Verfügung gestellt. Wird für den Vorteiler der Wert N_{Vor} verwendet, gilt für die Überlaufrate

$$R_{OV} = \frac{f_{\text{sys}}}{N_{\text{Vor}} \cdot 256}$$

Durch die Verwendung des Vorteilers lässt sich somit eine grobe Einstellung der Überlaufrate und damit der Frequenz der Timerereignisse vornehmen.

14.7.2.2 CTC Modus

Eine feinere Einstellung des zeitlichen Verlaufs der Timerereignisse lässt sich erzielen, wenn der Zählerstand, dem ein Ereignis zugeordnet ist, frei programmiert werden kann. Hierzu besitzt ein Timer ein Register, dessen Inhalt kontinuierlich mit dem aktuellen

Zählerstand verglichen wird. Im Fall der AVR-Timer wird dieses Register als *OCR (Output Compare Register)* bezeichnet. Erreicht der Zählerstand des Timers den im *OCR*-Register programmierten Wert, kann dies als ein Timerereignis gewertet werden, welches analog zum Timer-Überlauf im Nomal Mode behandelt wird. Dieser Betriebsmodus wird als *Clear Timer on Compare match (CTC)* bezeichnet. Der Verlauf des Zählerstandes im CTC-Modus ist in Abb. 14.10 dargestellt.

Mithilfe des CTC-Modus wird eine relativ feine Einstellung der Rate der Timerereignisse beziehungsweise der Frequenz des Ausgangssignals ermöglicht. Es gilt:

$$R_{CTC} = \frac{f_{\text{sys}}}{N_{\text{Vor}} \cdot (OCR + 1)}$$

14.7.2.3 PWM-Modi

Bei den beiden zuvor vorgestellten Timermodi lässt sich die Frequenz eines erzeugten Signals einstellen, das Tastverhältnis ist dagegen mit 0,5 festgelegt und kann in diesen Modi nicht geändert werden. Die Erzeugung eines Signals mit programmierbarem Tastverhältnis lässt sich mithilfe der sogenannten PWM-Modi realisieren. Der Name PWM-Modi ergibt sich aus der typischen Anwendung dieser Modi, nämlich die Erzeugung eines pulsweiten-modulierten Signals (PWM-Signal).

Bei Verwendung der PWM Modi zählt der Timer immer bis zum Erreichen des Endwertes. Es kann allerdings sowohl das Erreichen des Endwertes als auch das Durchlaufen des Vergleichswertes als interrupt-auslösendes Timerereignis genutzt werden.

Grundsätzlich werden zwei PWM-Modi unterschieden. Im ersten Fall des *Fast-PWM-Modus,* der auch als *Single-Slope-PWM* bezeichnet wird, zählt der Zähler des Timers nur aufwärts. Nach dem Erreichen des Endwertes beginnt der Zähler von 0 an zu zählen. Dabei findet die Invertierung des Ausgangssignals sowohl bei Erreichen des

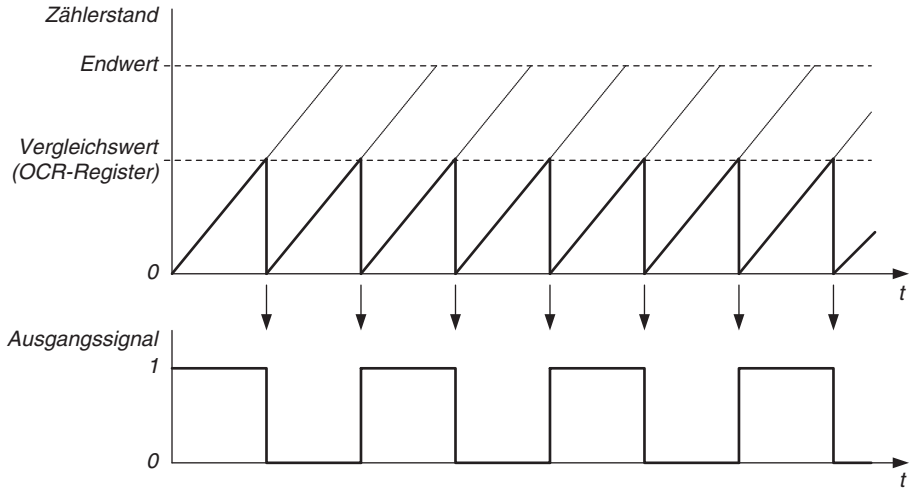


Abb. 14.10 Verlauf des Zählerstandes und eines Ausgangssignals im CTC Mode

Vergleichswertes als auch bei Erreichen des Endwertes statt. Abb. 14.11 zeigt das Zeitverhalten des Timers im Fast-PWM-Modus.

Die Frequenz des erzeugten PWM-Signals ist abhängig von dem gewählten Timerendwert TOP und der Eingangsfrequenz des Zählers, welche sich aus der Eingangsfrequenz f_{in} und dem gewählten Vorteilerwert N_{Vor} ergibt. Das erzeugte Signal besitzt die Frequenz f_{FPWM} mit einem Tastverhältnis V_{FPWM}

$$f_{FPWM} = \frac{f_{\text{in}}}{N_{\text{Vor}} \cdot (TOP + 1)}$$

$$V_{FPWM} = \frac{OCR}{TOP}$$

Im zweiten Fall des *Phase-Correct-PWM-Modus*, der auch als *Dual-Slope-PWM-Modus* bezeichnet wird, zählt der Timer zunächst aufwärts und nach Erreichen des Endwertes abwärts. Nach Erreichen des Wertes 0 zählt der Zähler wiederum aufwärts. Dabei findet ein Wechsel der Polarität des Ausgangssignals nur dann statt, wenn der Vergleichswert erreicht wird.

Hieraus ergeben sich die folgenden Gleichungen für die Frequenz beziehungsweise das Tastverhältnis des erzeugten Signals:

$$f_{PCPWM} = \frac{f_{\text{in}}}{2 \cdot N_{\text{Vor}} \cdot (TOP + 1)}$$

$$V_{PCPWM} = V_{FPWM} = \frac{OCR}{TOP}$$

Das zugehörige Zeitverhalten ist in Abb. 14.12 dargestellt.

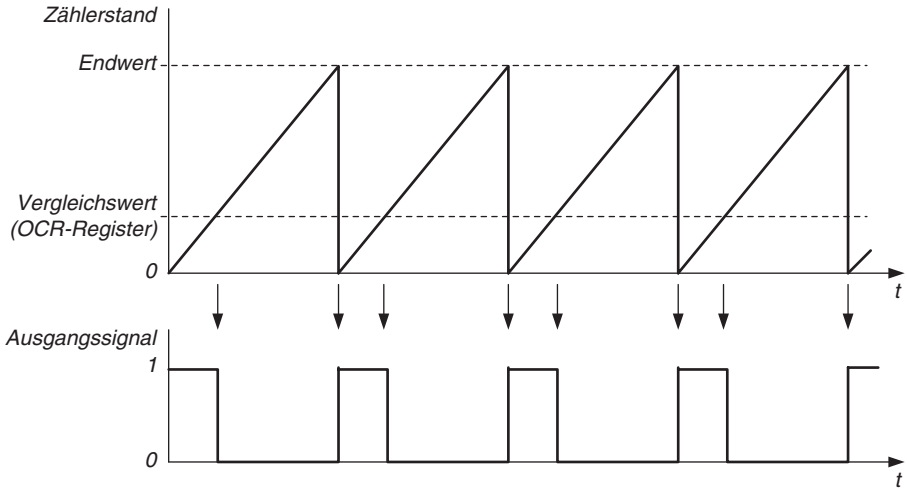


Abb. 14.11 Verlauf des Zählerstandes und eines Ausgangssignals im Fast-PWM-Mode

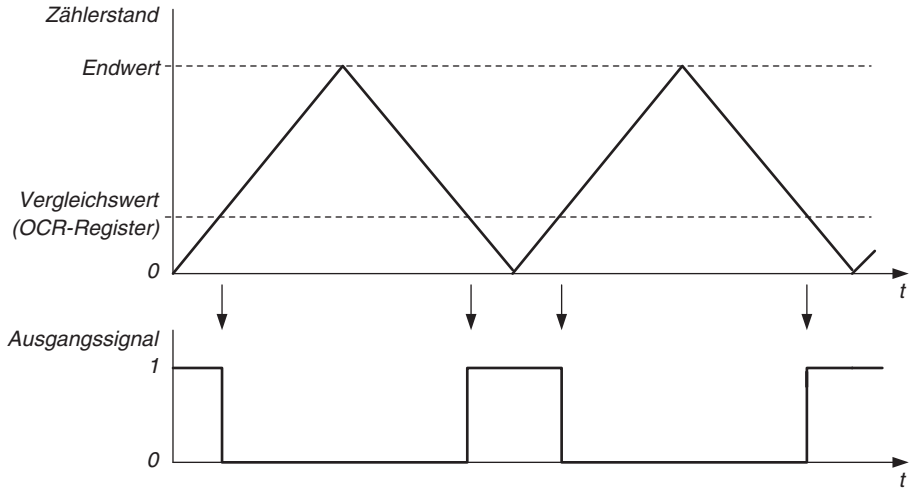


Abb. 14.12 Signalerzeugung im Phase-Correct-PWM-Mode (Dual-Slope-PWM-Mode)

14.7.2.4 Die Timer des ATmega32

Der Mikrocontroller *ATmega32* besitzt drei Timer, welche als Timer 0, Timer 1 und Timer 2 bezeichnet werden.

Timer 0 ist ein 8-Bit-Timer. Er besitzt einen Vergleichswert (I/O-Register: *OCR0*) und kann mit dem internen Systemtakt oder mit einem externen Takt (Anschluss: *T0*) betrieben werden.

Der Timer 1 ist ein 16-Bit-Timer, der ebenso mit dem internen Systemtakt oder einem externen Takt (Anschluss: *T1*) betrieben werden kann. Er besitzt zwei Vergleichswerte *(OCR1A, OCR1B)* und kann gleichzeitig zwei verschiedene Signale an den Controlleranschlüssen *OC1A* und *OC1B* ausgeben.

Timer 1 besitzt darüber hinaus eine sogenannte *Input-Capture-Unit (ICU)*. Die Aufgabe der ICU ist es, den aktuellen Zählerwert bei Auftreten eines zuvor programmierten Ereignisses in ein spezielles Register (I/O-Register *ICR1*) zu übertragen. Dieser Wert bleibt bis zum nächsten Auftreten des Ereignisses im ICP-Register gespeichert und kann von der CPU ausgelesen werden. Die ICU kann unter anderem zum zeitlichen Vermessen von digitalen Signalen verwendet werden. Wird beispielsweise als ICU-Ereignis das Auftreten einer steigenden Flanke des Eingangssignals am Anschluss ICP1 ausgewählt, ist es möglich, die Periodendauer des Eingangssignals durch zeitliches Vermessen zweier Taktflanken zu bestimmen. Hierzu muss lediglich der Wert des ICR1-Registers bei Auftreten der zweiten Taktflanke von dem ICR1-Wert bei Auftreten der ersten Taktflanke subtrahiert werden. Die Periodendauer T_{in} des Signals ergibt sich dann zu:

$$T_{\text{in}} = \frac{(ICR_{\text{Flanke2}} - ICR_{\text{Flanke2}}) \cdot N_{\text{Vor}}}{f_{\text{in}}}$$

Der Timer 2 des *ATmega32* ist ebenso wie der Timer 0 mit einem 8-Bit-Zähler ausgestattet, und es kann ein Vergleichswert *(OCR2)* programmiert werden. Die Besonderheit des Timers 2 ist der zugeordnete eingebettete Quarzoszillator, welcher unabhängig von den anderen Komponenten des Controllers betrieben werden kann. Der Timer 2 kann mithilfe dieses Oszillators mit einem Taktsignal versorgt werden, selbst wenn der Controller sich in einem Stromsparmodus befindet und wesentliche Systemkomponenten abgeschaltet sind.

Zur Programmierung der Timer des *ATmega32* stehen mehrere Register zur Verfügung, deren Funktion in den folgenden Abschnitten beschrieben wird.

14.7.2.5 Register des Timers 0

Der aktuelle Zählerstand des Timers 0 kann durch einen Zugriff auf das Register *TCNT0* gelesen oder auch geschrieben werden. Über das Register *OCR0* wird auf den Vergleichswert des Timers 0 zugegriffen.

Die Auswahl des Betriebsmodus des Timers 0 erfolgt über ein Steuerregister (Timer/Counter Control Register, *TCCR0*). Neben dem Betriebsmodus werden mithilfe dieses Registers auch der Wert des Vorteilers und die Quelle des Timertaktes festgelegt (Tab. 14.14).

Das Bit *FOC0* dient dem softwarebasierten Auslösen eines Compare-Match-Ereignisses (Zählerstand = Vergleichswert) unabhängig vom aktuellen Wert des *OCR0*-Registers oder des Zählerstandes. Die Bedeutung der anderen Bits dieses Registers ist in Tab. 14.15 und 14.16 zusammengefasst.

Tab. 14.14 Belegung des Registers TCCR0

TCCR0								
Bit	7	6	5	4	3	2	1	0
Name	FOC0	WGM00	COM01	COM00	WGM01	CS02	CS01	CS00

Tab. 14.15 Bedeutung der Bits CS00, CS01 und CS02

CS02	CS01	CS00	Bedeutung
0	0	0	Keine Taktauswahl, Timer ist abgeschaltet
0	0	1	Systemtakt mit Vorteiler = 1
0	1	0	Systemtakt mit Vorteiler = 8
0	1	1	Systemtakt mit Vorteiler = 64
1	0	0	Systemtakt mit Vorteiler = 256
1	0	1	Systemtakt mit Vorteiler = 1024
1	1	0	Externer Takt an Anschluss T0, aktive Flanke = fallende Flanke
1	1	1	Externer Takt an Anschluss T0, aktive Flanke = steigende Flanke

Tab. 14.16 Bedeutung der Bits WGM00, WGM01, COM00 und COM01

WGM00	COM01	COM00	WGM01	Bedeutung
0	0	0	0	*Normal Mode,* Signalerzeugung aus
0	0	0	1	*CTC-Modus,* Signalerzeugung aus
0	0	1	0	*Normal Mode,* Invertierung des OC0-Ausgangs bei Erreichen des Vergleichswertes
0	0	1	1	*CTC-Modus,* Invertierung des OC0-Ausgangs bei Erreichen des Vergleichswertes
0	1	0	0	*Normal Mode,* Löschen des OC0-Ausgangs bei Erreichen des Vergleichswertes
0	1	0	1	*CTC-Modus,* Löschen des OC0-Ausgangs bei Erreichen des Vergleichswertes
0	1	1	0	*Normal Mode*, Setzen des OC0-Ausgangs bei Erreichen des Vergleichswertes
0	1	1	1	*CTC-Modus*, Setzen des OC0-Ausgangs bei Erreichen des Vergleichswertes
1	0	0	0	*Phase-Correct-PWM-Modus,* Signalerzeugung aus
1	0	0	1	*Fast-PWM-Modus,* Signalerzeugung aus
1	0	1	0	Reserviert (ungültige Konfiguration)
1	0	1	1	Reserviert (ungültige Konfiguration)
1	1	0	0	*Phase-Correct-PWM-Modus,* Löschen des OC0-Ausgangs bei Erreichen des Vergleichswertes während des Aufwärtszählens, Setzen während des Abwärtszählens
1	1	0	1	*Fast-PWM-Modus,* Löschen des OC0-Ausgangs bei Erreichen des Vergleichswertes während des Aufwärtszählens, Setzen nach Zählerüberlauf
1	1	1	0	*Phase-Correct-PWM-Modus,* Setzen des OC0-Ausgangs bei Erreichen des Vergleichswertes während des Aufwärtszählens, Löschen während des Abwärtszählens
1	1	1	1	*Fast-PWM-Modus,* Setzen des OC0-Ausgangs bei Erreichen des Vergleichswertes während des Aufwärtszählens, Löschen nach Zählerüberlauf

14.7.2.6 Register des Timers 1

Da der Timer 1 auf einem Zähler mit einer Wortbreite von 16 bit basiert, sind Register, die sich auf den Zählerstand beziehen in Form von zwei 8 bit breiten Registern implementiert. So kann beispielsweise der Zählerstand des Timers über die Register *TCNT1L* (niederwertiges Byte) und *TCNT1H* (höherwertiges Byte) geschrieben und gelesen werden. Analog kann auf die Vergleichswerte mithilfe der Register *OCR1AL* und *OCR1AH* sowie *OCR2AL* und *OCR2AH* zugegriffen werden. Entsprechendes gilt für den ICR-Wert

der Input-Capture-Unit, auf welchen über die Register *ICRL* und *ICRH* zugegriffen werden kann.

Zur Steuerung werden für den Timer 1 zwei Register zur Verfügung gestellt, *TCCR1A* und *TCCR1B* (Tab. 14.17).

Die Bedeutung der Bits *CS12, CS11, CS10* und *FOC1A, FOC1B* besitzen eine zu den entsprechenden Bits des Timers 0 analoge Funktion (vgl. Tab. 14.18).

Die Bits *ICNC1* und *ICES1* sind der Input-Capture-Unit zugeordnet. Wird *ICNC1* gesetzt, wird damit ein Rauschfilter in der ICU aktiviert, welches kurzzeitige Signalwechsel am *ICP1*-Anschluss ausfiltert. Mit dem Bit *ICES1* kann die aktive Flanke des *ICP1*-Signals festgelegt werden: Ist *ICES1* gesetzt, reagiert die ICU auf eine steigende Flanke; ist *ICES1* gelöscht, reagiert die ICU auf eine fallende Flanke.

Tab. 14.17 Belegung der Register TCCR1A und TCCR1B

TCCR1A								
Bit	7	6	5	4	3	2	1	0
Name	COM1A1	COM1A0	COM1B1	COM1B0	FOC1A	FOC1B	WGM11	WGM10
TCCR1B								
Bit	7	6	5	4	3	2	1	0
Name	ICNC1	ICES1	-	WGM13	WGM12	CS12	CS11	CS10

Tab. 14.18 Bedeutung der Bits COM1A1, COM1B1, COM1A0 und COM1B0

COM1A1 COM1B1	COM1A0 COM1B0	Bedeutung
0	0	Signalerzeugung aus
0	1	*Normal, CTC:* Invertieren des OC-Ausgangs bei Erreichen des Vergleichswertes *PWM-Modi:* Signalerzeugung aus. Ausnahme WGM1 = 1001,1110 oder 1111: Invertieren des OC1A-Ausgangs bei Erreichen des Vergleichswertes
1	0	*Normal, CTC:* Löschen des OC-Ausgangs bei Erreichen des Vergleichswertes *Fast PWM:* Löschen des OC-Ausgangs bei Erreichen des Vergleichswertes während, Setzen nach Erreichen von TOP *PC-PWM:* Löschen des OC-Ausgangs bei Erreichen des Vergleichswertes während des Aufwärtszählens, Setzen während des Abwärtszählens
1	1	*Normal, CTC:* Setzen des OC-Ausgangs bei Erreichen des Vergleichswertes *Fast PWM:* Setzen des OC-Ausgangs bei Erreichen des Vergleichswertes während, Löschen nach Erreichen von TOP *PC-PWM:* Setzen des OC-Ausgangs bei Erreichen des Vergleichswertes während des Aufwärtszählens, Löschen während des Abwärtszählens

Mithilfe der Bits *WGM13* bis *WGM10* wird der Betriebsmodus des Timers festgelegt. Die 16 möglichen Modi des Timers 1 sind in Tab. 14.19 zusammengefasst.

14.7.2.7 Register des Timers 2

Die Register des Timers 2 entsprechen den Registern des Timers 0. Zur Unterscheidung der Timer werden die Register des Timers 2 mit der Ziffer 2 (statt 0) gekennzeichnet, also *TCNT2* statt *TCNT0, OCR2* statt *OCR0,* usw. Entsprechendes gilt für die einzelnen Bits der Timer-Register (zum Beispiel *WGM21* statt *WGM01*). Eine Ausnahme bilden die Bits *CS22* bis *CS20* zur Steuerung des Vorteilers (s. Tab. 14.20).

Als Taktquelle kann neben dem Systemtakt auch ein separater Quarzoszillator verwendet werden, welcher mit einem externen Quarz (typischerweise mit einem 32-kHz-Uhrenquarz) betrieben wird. Zur Steuerung dieser Funktion besitzt der Timer 2 ein weiteres Register, welches als Asynchronous Status Register *(ASSR)* bezeichnet wird (Tab. 14.21).

Ist das Bit *AS2* gesetzt, wird der Timer 2 über den separaten Quarzoszillator mit einem Taktsignal versorgt. Ist das Bit dagegen gelöscht, wird dem Timer der Systemtakt zugeführt. Die drei anderen Bits des *ASSR*-Registers dienen der Synchronisation zwischen der CPU und dem Timer: Wird beispielsweise ein Schreibzugriff auf das *OCR2*-Register ausgeführt, wird das Bit *OCR2UB* (OCR2 Update Busy) gesetzt. Erst wenn

Tab. 14.19 Bedeutung der Bits WGM13, WGM12, WGM11 und WGM10

WGM13	GM12	WGM11	WGM10	Bedeutung
0	0	0	0	*Normal Mode,* TOP = 0xFFFF
0	0	0	1	*Phase-Correct-PWM-Modus,* TOP = 0x00FF
0	0	1	0	*Phase-Correct-PWM-Modus,* TOP = 0x01FF
0	0	1	1	*Phase-Correct-PWM-Modus,* TOP = 0x03FF
0	1	0	0	*CTC-Modus,* TOP = OCR1A
0	1	0	1	*Fast-PWM-Modus,* TOP = 0x00FF
0	1	1	0	*Fast-PWM-Modus,* TOP = 0x01FF
0	1	1	1	*Fast-PWM-Modus,* TOP = 0x03FF
1	0	0	0	*Phase-and-Frequency-Correct-PWM,* TOP = ICR1
1	0	0	1	*Phase-and-Frequency-Correct-PWM,* TOP = OCR1A
1	0	1	0	*Phase-Correct-PWM-Modus,* TOP = ICR1
1	0	1	1	*Phase-Correct-PWM-Modus,* TOP = OCR1A
1	1	0	0	*CTC-Modus,* TOP = ICR1
1	1	0	1	Reserviert
1	1	1	0	*Fast-PWM-Modus,* TOP = ICR1
1	1	1	1	*Fast-PWM-Modus,* TOP = OCR1A

Tab. 14.20 Bedeutung der Bits CS00, CS01 und CS02

CS22	CS21	CS20	Bedeutung
0	0	0	Timer ist abgeschaltet
0	0	1	Vorteiler = 1
0	1	0	Vorteiler = 8
0	1	1	Vorteiler = 32
1	0	0	Vorteiler = 64
1	0	1	Vorteiler = 128
1	1	0	Vorteiler = 256
1	1	1	Vorteiler = 1024

Tab. 14.21 Belegung des Registers ASSR

ASSR								
Bit	7	6	5	4	3	2	1	0
Name	–	–	–	–	AS2	TCN2UB	OCR2UB	TCR2UB

der geschriebene Wert vom Timer übernommen wurde, wird das Bit von der Timer-HW zurückgesetzt. Entsprechendes gilt für die Register *TCNT2* und *TCCR2,* denen die Bits *TCN2UB* und *TCR2UB* zugeordnet sind.

14.7.2.8 Timer als Interruptquellen

Die Timer des *ATmega32* können auch als Interruptquellen genutzt werden. Es können mithilfe aller Timer Interrupts ausgelöst werden, wenn ein Timer-Überlauf aufgetreten ist oder der Zählwert des Timers den Vergleichswert erreicht hat. Zusätzlich kann für den Timer 1 das Auftreten eines „Input-Capture-Ereignisses“ als Interruptquelle genutzt werden. Die Freigabe der jeweiligen Interrupts geschieht durch Setzen des zugehörigen Bits im Timer/Counter-Interrupt-Mask-Register *(TIMSK)*.

Darüber hinaus ermöglichen die Timer das Abfragen des jeweiligen Interrupt-Status durch das Timer/Counter-Interrupt-Flag-Register *(TIFR)*. Mithilfe dieses Registers ist es zum Beispiel möglich, das Auftreten einer der oben genannten Interruptbedingungen durch die CPU anzufragen, ohne eine interruptbasierte Verarbeitung zu nutzen (Tab. 14.22).

Die Bits *OCIE*x und *OCF*x sind den Vergleichsereignissen (Zählerstand = Vergleichswert) und die Bits *TOIE*x und *TOV*x den Überlaufereignissen zugeordnet, während die Bits *TICIE1* und *ICF1* der Input Capture Unit des Timers 1 zugeordnet sind.

Die Anwendungsmöglichkeiten der Timer-Interrupts sind sehr vielfältig. Ein sehr einfaches Beispiel ist die zyklische Erzeugung von Interrupts zur Unterbrechung des Hauptprogramms, um regelmäßig anfallende Aufgaben abzuarbeiten. Darüber hinaus sind regelmäßige Timer-Interrupts eine wesentliche Grundlage vieler Betriebssysteme.

Eine wichtige Bedeutung kommt der Timer-Interrupt-Programmierung auch bei der Erzeugung von Ausgangssignalen zu. Häufig ist es erforderlich, die Parameter des

Tab. 14.22 Belegung der Register TIMSK und TIFR

TIMSK								
Bit	7	6	5	4	3	2	1	0
Name	OCIE2	TOIE2	TICIE1	OCIE1A	OCIE1B	TOIE1	OCIE0	TOIE0
TIFR								
Bit	7	6	5	4	3	2	1	0
Name	OCF2	TOV2	ICF1	OCF1A	OCF1B	TOV1	OCF0	TOV0

mithilfe des Timers erzeugten Signals dynamisch zu modifizieren. Der Timer muss also im laufenden Betrieb umkonfiguriert werden. Hierbei muss beachtet werden, dass der Zählerstand des Timers nicht unbedingt mit der Ausführung des Programms synchronisiert ist. Um diese Synchronisation zu unterstützen, sind die *OCR*-Register der Timer mit „Schattenregistern“ ausgestattet. Wird in einem solchen Schattenregister ein Wert abgelegt, wirkt sich der neue Wert nicht sofort in der Timer-Hardware aus. Vielmehr wird der im Schattenregister gespeicherte Wert erst bei Erreichen eines definierten Zählerstands (zum Beispiel Timerendwert) in das eigentliche Timer-Register übernommen. Für die dynamische Timer-Programmierung ist es sehr bequem den Interrupt freizugeben, der dem o.g. Zählerstand zugeordnet ist. Die Konfiguration des Timers erfolgt dann jeweils in der entsprechenden ISR. Auf diese Weise wird der Timer nur zu definierten Zeiten neu konfiguriert und das Verhalten des Ausgangssignals ist nicht von den Laufzeiten der einzelnen Programmteile der Software abhängig.

Als ein einfaches Beispiel für die Interrupt-Programmierung wird im Folgenden ein C-Programm vorgestellt, mit dem eine Uhr realisiert werden kann.

Die Uhrzeit wird in den globalen Variablen *Sekunden, Minuten* und *Stunden* abgelegt, die von einer Timer-ISR beschrieben werden. Im Hauptprogramm wird der Timer 1 so konfiguriert, dass alle 8000 Systemtaktzyklen ein Interrupt ausgelöst wird (Vorteiler = 8, OCR-Register = 999). Beträgt die Systemtaktfrequenz beispielsweise 16 MHz, werden also 2000 Interrupts pro Sekunde auftreten.

In der Timer-ISR werden die aufgetretenen Timerinterrupts gezählt. Ist eine Sekunde vergangen, wird die Variable *Sekunden* inkrementiert. Ist diese anschließend gleich 60, wird sie auf Null gesetzt und die Variable *Minuten* inkrementiert. Der Wert der Variablen *Minuten* wird anschließend überprüft und gegebenenfalls auf Null gesetzt und die Variable *Stunden* inkrementiert.

Da die Anzahl der pro Sekunde auftretenden Interrupts von der Systemtaktfrequenz abhängt, muss diese bekannt sein. Für AVR-Programme gilt die Vereinbarung, dass die Systemtaktfrequenz im Präprozessorsymbol *F_CPU* abgelegt wird. In der ISR wird überprüft, ob der Interruptzähler den Wert *F_CPU*/8000 erreicht hat, also eine Sekunde vergangen ist.

Im Folgenden ist das Programm für die Realisierung einer Uhr dargestellt.

```
#include <avr/io.h>
#include <avr/interrupt.h>     // Header-Datei für Interrupts
// Symbol für Systemtaktfrequenz setzen. Dies kann auch in der
// Entwicklungsumgebung erfolgen
#define F_CPU 16000000          // Beispiel: 16 MHz
volatile unsigned char Sekunden;
volatile unsigned char Minuten;
volatile unsigned char Stunden;
// Unterprogramm zur Ausgabe der Uhrzeit
void Zeitausgabe(unsigned char Stunden, unsigned char Minuten,
                 unsigned char Sekunden) {
   // zum Beispiel Ausgabe auf einem Sieben-Segment-Display
}
// Hauptprogramm
void main() {
   // Timer 1 initialisieren
   // nicht gesetzte Bits sind nach dem Reset des Controllers 0
   TCCR1B |= 1<<WGM12;  // CTC-Modus
   TCCR1B |= 1<<CS11;   // Vorteiler = 8
   OCR1A = 999;         // alle 1000 Taktzyklen ein Interr.
   TIMSK  |= 1<<OCIE1A; // Freigabe Vergleichsinterrupt
   sei();               // Globale Interruptfreigabe
   // Normaler Betrieb: hier erfolgt die Ausgabe der Uhrzeit
   // das Zählen der Sekunden erfolgt in der ISR
   while (1) {
        Zeitausgabe(Stunden, Minuten, Sekunden);
   }
}

// Timer 1 ISR
ISR (TIMER1_COMPA_vect) {
   static unsigned long IntCount = 0;
   IntCount++;
   if (IntCount == F_CPU/8000) { // 1 Sekunde vergangen ?
      IntCount = 0;      // Zähler zurücksetzen
      Sekunden++;        // Uhrzeit setzen…
      if (Sekunden==60) {
         Sekunden = 0;
         Minuten++;
         if (Minuten==60) {
            Minuten = 0;
            Stunden++;
            if (Stunden==24) Stunden = 0;
         }
      }
   }
}
```

14.7.2.9 Watchdog-Timer

Grundsätzlich kann selbst bei sorgfältigster Entwicklung von Softwarekomponenten nicht sichergestellt werden, dass ein Programm komplett fehlerfrei ist und in allen Betriebszuständen des Systems reibungslos funktioniert. Unentdeckte Softwarefehler können je nach Anwendung fatale Folgen für ein System oder für die Umgebung des Systems, incl. der Benutzer haben. Um einen Systemabsturz, der zum Beispiel aufgrund eines Softwarefehlers aufgetreten ist, abfangen zu können, besitzen Mikrocontroller einen sogenannten Watchdog-Timer.

Die Arbeitsweise des Watchdogs ähnelt dem Prinzip des sogenannten „Totmann-Knopfes“, wie er in Schienenfahrzeugen eingesetzt wird: Der Fahrzeugführer muss in regelmäßigen Abständen den Knopf bedienen. Unterlässt er dies, wird automatisch ein Nothalt des Systems ausgeführt.

Der Watchdog-Timer basiert auf einem Abwärtszähler. Erreicht der Zähler den Zählerstand 0, wird durch den Timer ein Zurücksetzen des Controllers ausgelöst. Um dieses Zurücksetzen zu vermeiden, muss der Zähler des Watchdog-Timers per Software regelmäßig auf einen von Null verschiedenen Wert gesetzt werden. Arbeitet das System einwandfrei, wird der Zähler des Watchdogs nie den Wert 0 erreichen und somit kein Zurücksetzen des Systems auslösen.

Das Taktsignal für die Watchdog-Timer ATmega-Serie wird mithilfe eines eingebetteten Oszillators realisiert, sodass für die Takterzeugung keine externen Komponenten benötigt werden. Durch den Einsatz eines Vorteilers können dem Zähler des Watchdogs verschiedene Taktfrequenzen zugeführt werden, wodurch die Zeit bis zum Erreichen des Zählerstandes 0 über das CPU-Programm festgelegt werden kann.

Das softwarebasierte Setzen des Watchdog-Zählers eines *ATmega32* erfolgt in Assembler mit dem Spezialbefehl *wdr* (Watchdog Reset) beziehungsweise in C durch den Aufruf der Funktion *wdt_reset()*. Zur Programmierung des Watchdogs steht das Watchdog Timer Control Register *(WDTCR)* zur Verfügung (Tab. 14.23).

Das Bit *WDE* dient zum Aktivieren ($WDE = 1$) oder Deaktivieren ($WDE = 0$) des Watchdog-Timers. Soll der Watchdog deaktiviert werden, müssen zunächst die Bits *WDTOE* und *WDE* gesetzt und anschließend das Bit *WDE* innerhalb von 4 Taktzyklen gelöscht werden. Auf diese Weise soll ein unbeabsichtigtes Deaktivieren des Watchdogs ausgeschlossen werden.

Die Bits WDP2 bis WDP0 werden zur Programmierung des Vorteilers verwendet. Die Zeit, die zwischen dem Ausführen des *wdr*-Befehls und dem Erreichen des Zählwertes 0 vergeht, ergibt sich gemäß Tab. 14.24.

Tab. 14.23 Belegung des Registers WDTCR

WDTCR								
Bit	7	6	5	4	3	2	1	0
Name	-	-	-	WDTOE	WDE	WDP2	WDP1	WDP0

Tab. 14.24 Bedeutung der Bits WDP2, WDP1 und WDP0

WDP2	WDP1	WDP0	Zeit bis zum Erreichen des Zählerstands 0 (circa)
0	0	0	17 ms
0	0	1	33 ms
0	1	0	65 ms
0	1	1	130 ms
1	0	0	260 ms
1	0	1	520 ms
1	1	0	1,0 s
1	1	1	2,1 s

Der Grund für das Zurücksetzen des Controllers kann mithilfe des „Microcontroller Unit Control and Status“ Registers *(MCUCSR)* von der CPU abgefragt werden (Tab. 14.25).

Je nach Grund des Resets wird eines der fünf niederwertigen Bits des *MCUCSR*-Registers gesetzt. Die Bedeutung dieser Bits ist in Tab. 14.26 zusammengefasst.

14.7.3 Schnittstellen für die serielle Datenübertragung

Fast alle Mikrocontroller stellen eingebettete Peripheriekomponenten zur Verfügung, die eine bitserielle Datenübertragung unterstützen. Der wesentliche Vorteil einer bitseriellen Übertragung im Gegensatz zu einer bitparallelen Übertragung ist die Reduktion des Verdrahtungsaufwands zwischen Sender und Empfänger. Die Reduktion dieses Aufwands

Tab. 14.25 Belegung des Registers MCUCSR

MCUCSR								
Bit	7	6	5	4	3	2	1	0
Name	JTD	ISC2	–	JTRF	WDRF	BORF	EXTRF	PORF

Tab. 14.26 Bedeutung der Bits im MCUCSR-Register

JTRF	Reset durch das JTAG-Programmier- und Debug-Interface
WDRF	Watchdog-Reset
BORF	Brownout-Detection-Reset (Versorgungsspannung unterschreitet programmierten Wert)
EXTRF	Externer Reset-Anschluss wurde aktiviert
PORF	Versorgungsspannung wurde eingeschaltet („Power-On-Reset“)

kann insbesondere bei einfachen, kostensensitiven Systemen einen wichtigen Aspekt darstellen.

In den folgenden Abschnitten wird die Diskussion auf die in Mikrocontrollern häufig anzutreffenden seriellen Schnittstellen beschränkt. Zunächst wird jeweils das verwendete Übertragungsprotokoll vorgestellt und im Anschluss daran wird die spezifische Implementierung der Schnittstellen am Beispiel des *ATmega32* näher erläutert.

14.7.3.1 U(S)ART

Die Abkürzung *UART* steht für *Universal Asynchronous Receiver/Transmitter,* also ein universell einsetzbarer Sender und Empfänger für asynchrone Datenübertragungen. Der Begriff „asynchron" bedeutet hier, dass bei dieser Datenübertragung kein Taktsignal zwischen Sender und Empfänger ausgetauscht wird. Der Empfänger muss allein aus der Kenntnis des Datensignals die übertragenen Datenbits extrahieren. Eine Erweiterung des UARTs stellt der USART dar. Der zusätzliche Buchstabe „S" soll andeuten, dass diese Komponente auch eine synchrone Datenübertragung unterstützen kann. In diesem Fall wird vom Sender ein Taktsignal erzeugt, das zusammen mit dem Datensignal übertragen wird.

Bereits um 1960 wurde ein geeignetes Protokoll zur asynchronen seriellen Datenübertragung zwischen Rechnern entwickelt und standardisiert. Die bekannteste Implementierung dieser Anwendung stellt die serielle Schnittstelle eines PCs dar, die häufig auch als RS232-Schnittstelle, V.24-Schnittstelle, COM-Port oder einfach als *serielle Schnittstelle* bezeichnet wird. Diese Schnittstelle diente viele Jahre als Kommunikationsschnittstelle zwischen Rechnern oder zwischen Rechnern und Modems, welche eine Datenfernübertragung über Telefonleitungen ermöglicht.

Die Bedeutung der RS232-Schnittstelle für PCs hat in den letzten 30 Jahren kontinuierlich abgenommen. Rechner werden heute meist über Ethernet-Leitungen oder WLAN vernetzt, die deutlich höhere Übertragungsraten ermöglichen. Im Bereich der Datenfernübertragung werden Technologien wie DSL eingesetzt, wobei die Verbindung zu einem DSL-Modem über USB oder Ethernet realisiert wird. Daher werden von heutigen PCs in der Regel keine RS232-Schnittstellen mehr zur Verfügung gestellt. Zur Nutzung dieser Schnittstelle müssen häufig entweder entsprechende Erweiterungskarten oder USB-Geräte angeschafft werden, die über einen USB-Anschluss des Rechners die gewünschte RS232-Schnittstelle zur Verfügung stellen.

Eine größere Bedeutung besitzt die RS232-Schnittstelle im Bereich der Mikrorechnersysteme. Hier steht häufig nicht die erzielbare Datenrate im Vordergrund, sondern zunächst die einfache Implementierbarkeit der Kommunikation zweier Komponenten. Eine häufige Anwendung ist die Verbindung eines Mikrorechnersystems mit einem PC, um Statusmeldungen an den PC zu senden oder auch um Programme und Daten an den Mikrorechner zu senden.

Entsprechend der ursprünglichen Anwendung im Bereich der Datenfernübertragung werden sogenannte Datenendeinrichtungen (Data Terminal Equipment, *DTE*) und Datenübertragungseinrichtungen (Data Communication Equipment, *DCE*) unterschieden. Ein

PC arbeitet in der Regel als DTE, während die angeschlossenen Geräte meist als DCE betrieben werden. Diese Unterscheidung ist insbesondere für die Steckerbelegung der Geräte von Bedeutung. Für die serielle Übertragung mithilfe des RS232-Standards werden heute fast ausschließlich 9-polige Sub-D-Steckverbindungen verwendet, deren Belegung in Tab. 14.27 zusammengefasst ist.

In vielen Anwendungsfällen wird nur ein Teil der dargestellten Signale verwendet. Im einfachsten Fall ist es möglich eine bidirektionale Verbindung zwischen zwei Stationen (zum Beispiel PC und Mikrocontroller) mithilfe der Anschlüsse RXD, TXD und GND zu realisieren.

Die RS232-Schnittstelle arbeitet mit negativer Logik. Eine logische Null wird durch einen Spannungspegel im Bereich von $+3$ bis $+15$ V, eine logische Eins durch einen Pegel zwischen -3 und -15 *V* dargestellt. Ein direkter Anschluss der Signale der seriellen Schnittstelle eines PCs an einen Mikrocontroller sollte niemals erfolgen, da der Controller hierbei zerstört werden würde. Es ist also ein Umsetzen der Pegel der seriellen Schnittstelle erforderlich. Hierfür stehen verschiedene integrierte Bausteine zur Verfügung, die auch eine Umwandlung zwischen negativer und positiver Logik durchführen. Ein Beispiel ist der von verschiedenen Herstellern angebotene Baustein MAX232.

14.7.3.2 Datenübertragung mit dem UART-Protokoll

Der Empfänger erhält nur das vom Sender generierte Datensignal. Um allein aus der Kenntnis des Datensignals die übertragenen Daten zu extrahieren können, muss der Empfänger den Beginn einer Datenübertragung erkennen können. Der Beginn einer Datenübertragung wird durch ein sogenanntes Startbit gekennzeichnet, welches den vordefinierten Wert 0 besitzt. Anschließend erfolgt die Übertragung einer zwischen Sender und Empfänger vereinbarten Anzahl von Datenbits. Hierbei gilt die Vereinbarung, dass zuerst das niederwertigste Bit *(Least Significant Bit, LSB)* übertragen wird. In der Regel wird eine Übertragung von 8 Datenbits ausgewählt.

Tab. 14.27 Belegung der 9-poligen Sub-D-Steckverbindungen

Nr.	Kürzel	Name	Bedeutung	Datenrichtung
1	DCD	Data Carrier Detect	DCE erhält einlaufende Daten	DCE → DTE
2	RXD	Receive Data	Empfangsdaten (des DTE, z. B. PC)	DCE → DTE
3	TXD	Transmit Data	Sendedaten (des DTE, z. B. PC)	DTE → DCE
4	DTR	Data Terminal Ready	Einsatzbereitschaft des DTE	DTE → DCE
5	GND	Ground	Signalmasse	
6	DSR	Data Set Ready	Einsatzbereitschaft des DCE	DCE → DTE
7	RTS	Ready To Send	DTE (z. B. PC) möchte Daten übertragen	DTE → DCE
8	CTS	Clear To Send	DCE kann Daten entgegennehmen	DCE → DTE
9	RI	Ring Indicator	Modem erkennt Anruf	DCE → DTE

Im Anschluss an die Datenübertragung erfolgt die Übertragung von ein bis zwei Stoppbits, welche den Wert 1 besitzen. Bis zum Beginn der nächsten Datenübertragung verbleibt das Sendesignal auf dem Wert 1.

Um die Datenübertragung gegenüber kurzzeitigen Störungen abzusichern, kann zwischen den Daten und dem Stoppbit ein Paritätsbit *(parity bit)* eingefügt werden. Verwendet der Sender die Übertragung eines Paritätsbits, muss dies dem Empfänger bekannt sein. Ebenso müssen Sender und Empfänger die gleiche Rechenvorschrift zur Berechnung des Paritätsbits verwenden.

Der Empfänger berechnet aus den empfangenen Daten das erwartete Paritätsbit und vergleicht dieses mit dem vom Sender empfangenen Paritätsbit. Sind die Werte beider Bits identisch, wird davon ausgegangen, dass eine fehlerfreie Übertragung stattgefunden hat.

Für die Berechnung des Paritätsbits werden zwei Vorschriften verwendet, die als „ungerade Parität“ (odd parity) beziehungsweise „gerade Parität“ (even parity) bezeichnet werden. In beiden Fällen erfolgt die Berechnung des Paritätsbits p derart, dass eine Exklusiv-Oder-Verknüpfung der Datenbits d_i und eines Modusbits m ($m = 0$ für even parity, $m = 1$ für odd parity) durchgeführt wird:

$$p = d_{n-1} \oplus d_{n-2} \oplus \cdots \oplus d_1 \oplus d_0 \oplus m$$

Aufgrund dieser Vorschrift zur Berechnung des Paritätsbits lassen sich vom Empfänger nur Übertragungsfehler erkennen, bei denen nur ein Fehler oder eine ungerade Anzahl fehlerhafter Bits auftritt. Ist die Anzahl der durch Übertragungsfehler modifizierten Bits dagegen gerade, würde der Empfänger die Daten als korrekt übertragen ansehen. Darüber hinaus ermöglicht dieser sehr einfache Fehlerschutz keine empfängerseitige Fehlerkorrektur, da der Empfänger nicht bestimmen kann, welches Datenbit fehlerhaft übertragen wurde.

In der Praxis wird die Übertragung eines Paritätsbits häufig nicht genutzt, wenn von einem relativ sicheren Übertragungskanal ausgegangen werden kann. Dies ist meist bei einer Verbindung zwischen einem PC und einem Mikrocontroller der Fall, wenn die Datenleitungen nicht länger als wenige Meter sind und die Umgebung keine starken elektromagnetischen Störquellen besitzt.

Abb. 14.13 zeigt exemplarisch den zeitlichen Verlauf der Übertragung eines Bytes mit den Einstellungen: 8 Datenbits, 1 Stoppbit, gerade Parität.

Neben der Anzahl der Daten- und Stoppbits sowie der verwendeten Paritätsberechnung (odd, even, keine), muss dem Empfänger die Dauer der Übertragung eines einzelnen Bits *(Bitdauer)* bekannt sein. Da die Bitdauer direkt die Übertragungsrate beeinflusst, wird von Bitrate oder von *Baudrate* gesprochen.

Theoretisch können beliebige Baudraten verwendet werden. In der Praxis werden jedoch meist standardisierte Baudraten verwendet. Typische Baudraten sind in Tab. 14.28 zusammengefasst.

14.7.3.3 Handshake zwischen Sender und Empfänger

In vielen Anwendungsfällen kann davon ausgegangen werden, dass der Empfänger die vom Sender empfangenen Daten stets verarbeiten kann. Dies ist zum Beispiel der Fall,

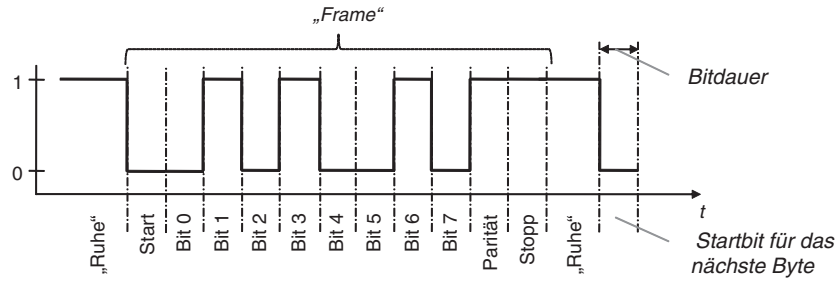


Abb. 14.13 Zeitdiagramm für die Übertragung eines Bytes

Tab. 14.28 In der Praxis häufig verwendete Baudraten

Baudrate (in bit/s)	Bitdauer (µs)
2400	416,67
9600	104,17
19.200	52,08
38.400	26,04
57.600	17,36
115.200	8,68

wenn ein Mikrocontroller Daten an einen PC sendet, um diese mithilfe eines Terminalprogramms auf dem Monitor anzuzeigen. Aufgrund der hohen Rechenleistung eines PCs und der vergleichsweise geringen Datenrate der seriellen Schnittstelle, wird der PC in der Regel alle vom Controller gesendeten Daten korrekt verarbeiten können.

Genauso sind auch Anwendungsfälle denkbar, in denen der Empfänger die gesendeten Daten nicht sofort verarbeiten kann. Würde der Sender diese Situation ignorieren und weiter Daten senden, wäre ein Verlust von Daten die Folge. Um diesen Datenverlust zu vermeiden, muss die Möglichkeit bestehen, dem Sender mitzuteilen, dass der Empfänger kurzzeitig nicht in der Lage ist, weitere Daten zu empfangen. Die hierfür notwendige Kommunikation zwischen Empfänger und Sender wird als *Handshake* bezeichnet.

Eine Möglichkeit zur Implementierung stellt das sogenannte *Software-Handshake* dar. In diesem Fall wird die Handshake-Information über die Datenleitungen *RXD* und *TXD* ausgetauscht. Ist ein Gerät nicht bereit Daten zu empfangen, sendet es an die Gegenstelle den Wert 19 (0x13). Der Sender wird daraufhin das Senden weiterer Daten einstellen. Sobald der Empfänger wieder bereit ist, sendet er den Wert 17 (0x11) und die Datenübertragung wird fortgesetzt. Da die beiden Zahlenwerte im ASCII-Code als *XOFF* beziehungsweise *XON* bezeichnet werden, wird diese Art des Handshakes oft auch als *XON/XOFF-Handshake* bezeichnet.

Ein Nachteil des Software-Handshakes ist es, dass zwei Zahlenwerten eine besondere Bedeutung zugeordnet wird, sodass diese Werte nicht mehr für die Datenübertragung zur Verfügung stehen.

Dieser Nachteil kann durch das sogenannte *Hardware-Handshake* vermieden werden. Hierfür können die beiden Signale *RTS* und *CTS* herangezogen werden. Ein DTE (zum Beispiel ein PC) teilt einem DCE mit, dass es Daten senden möchte indem es die Leitung *RTS* aktiviert *(RTS = 0)*. Das *DCE* setzt daraufhin das Signal *CTS* auf 0 sobald es für den Datenempfang bereit ist. Für das *DTE* wird bei Verwendung dieses Handshakes angenommen, dass es jederzeit alle vom *DCE* gesendeten Daten korrekt verarbeiten kann. Gegebenenfalls kann jedoch vom *DTE* das *DTR*-Signal deaktiviert werden um so ein Senden von Daten zu unterbrechen.

In vielen Anwendungen wird der *RTS*-Anschluss als RTR-Signal *(Ready To Receive)* verwendet. In diesem Fall zeigt eine logische 0 auf der RTS-Leitung an, dass das *DTE* Daten empfangen kann. Das *DCE* signalisiert die Empfangsbereitschaft dagegen durch das Aktivieren der *CTS*-Leitung *(CTS = 0)*.

Darüber hinaus können auch die Signale *DSR* und *DTR* zur Realisierung eines Hardware-Handshakes verwendet werden, welche in ähnlicher Weise wie die Signale *RTR* und *CTS* angesteuert werden können.

Obwohl in vielen Mikrocontrollern eingebettete Peripheriekomponenten zur Datenübertragung mithilfe des RS232-Protokolls zur Verfügung stehen, werden von diesen Komponenten häufig nur die Signale *RXD* und *TXD* bedient. Soll die Kommunikation mithilfe eines Hardware-Handshakes erfolgen, ist hierfür die softwarebasierte Ansteuerung von zusätzlichen Portanschlüssen erforderlich. Das Hardware-Handshake wird in diesem Fall also durch das Programm in Software implementiert.

14.7.3.4 Der USART im AVR

Viele Mikrocontroller der AVR-Serie besitzen eine eingebettete Schnittstelle zur Realisierung einer asynchronen seriellen Kommunikation. Im Fall des *ATmega32* wird diese Peripheriekomponente als *USART (Universal Synchronous Asynchronous Receiver/Transmitter)* bezeichnet. Diese Komponente unterstützt serielle Übertragungen mit 5 bis 9 Datenbits und 1 oder 2 Stoppbits. Neben der Analyse des Paritätsbits existieren weitere Möglichkeiten zur Erkennung von Übertragungsfehlern, die in diesem Abschnitt beschrieben werden. Wie die Bezeichnung USART andeutet, kann diese Komponente sowohl in einem asynchronen als auch in einem synchronen Modus betrieben werden. Im Folgenden wird nur auf den asynchronen Betriebsmodus näher eingegangen.

Der USART des *ATmega32* stellt die beiden Signale TXD (Datenausgang) und RXD (Dateneingang) zur Verfügung. Diese Signale werden an den Anschlüssen *PD0* und *PD1* als alternative Portfunktionen herausgeführt. Wird der USART durch das auf dem Controller laufende Programm aktiviert, stehen die Portanschlüsse *PD0* und *PD1* nicht mehr als frei programmierbare Portanschlüsse zur Verfügung.

Für die Konfiguration des USARTs werden drei USART-Control-and-Status-Register *(UCSRA, UCSRB, UCSRC)* sowie zwei Bitratenregister *(UBBRL, UBBRH)* bereitgestellt (s. Tab. 14.29).

Mit Setzen der Bits *TXEN* (Transmitter Enable) beziehungsweise *RXEN* (Receiver Enable) wird der Sender beziehungsweise Empfänger der seriellen Schnittstelle des *ATmega32* aktiviert.

Tab. 14.29 USART-Control- und Statusregister: UCSRA, UCSRB, UCSRC

UCSRA								
Bit	7	6	5	4	3	2	1	0
Name	RXC	TXC	UDRE	FE	DOR	PE	U2X	MPCM
UCSRB								
Bit	7	6	5	4	3	2	1	0
Name	RXCIE	TXCIE	UDRIE	RXEN	TXEN	UCSZ2	RXB8	TXB8
UCSRC								
Bit	7	6	5	4	3	2	1	0
Name	URSEL	UMSEL	UPM1	UPM0	USBS	UCSZ1	UCSZ0	UCPOL

Die Baudrate hängt von der Systemtaktfrequenz und dem Wert im *UBRR*-Register ab. Mithilfe der folgenden Formel kann ein geeigneter Wert für die Programmierung der Register *UBRRH* (höherwertiges Byte) und *UBRRL* (niederwertiges Byte) bestimmt werden.

$$UBRR = \frac{f_{\text{sys}} + 8 \cdot Baudrate}{16 \cdot Baudrate} - 1$$

Die Auswahl der Anzahl der Datenbits innerhalb eines Frames (zwischen Start- und Stoppbit) wird durch *UCSZ* festgelegt (s. Tab. 14.30).

Die zu sendenden (oder empfangenen) Daten werden im Register *UDR* (USART Data Register) abgelegt. Ein Schreibzugriff auf dieses Register übermittelt neue zu sendende Daten an die Schnittstelle, während die CPU mithilfe eines Lesezugriffs auf empfangene Daten zugreifen kann.

Bei der Verwendung von 9 Datenbits wird das höchstwertige Datenbit durch die Bits *TXB8* beziehungsweise *RXB8* repräsentiert. In allen anderen Fällen haben diese Bits keine Bedeutung. Die weiteren Bits der *UCSR*-Register sind in Tab. 14.31 zusammengefasst.

Für die häufig verwendete Konfiguration „8 Datenbits, keine Parität, 1 Stoppbit, asynchroner Modus“ ergeben sich für die Programmierung der UCSR-Register die Werte $UCSRB = 0x18$ und $UCSRC = 0x86$. Die Bits des Registers UCSRA können auf den

Tab. 14.30 Auswahl der Datenbits pro Frame mithilfe der UCSZ-Bits

UCSZ2	UCSZ1	UCSZ0	Datenbits
0	0	0	5
0	0	1	6
0	1	0	7
0	1	1	8
1	1	1	9

Tab. 14.31 Bedeutung der Bits der UCSR-Register

Bit	Name	Bedeutung
RXC	Receive Complete	1: Daten eines Frames empfangen
TXC	Transmit Complete	1: Daten eines Frames versendet
UDRE	Data Register Empty	1: Daten-Register (UDR) ist leer
FE	Frame Error	1: Ein empfangenes Stoppbit hatte den Wert 0
DOR	Data Overrun	1: Daten im UDR-Register wurden nicht rechtzeitig gelesen worden und wurden von neuen empfangenen Daten überschrieben
PE	Parity Error	1: Paritätsfehler erkannt
U2X	Double Speed	1: Verdopplung der Übertragungsgeschwindigkeit
MPCM	Multiprocessor Communication Mode	1: Multiprocessor Modus aktiviert
RXCIE	Receive Complete Interrupt Enable	Lokale Interruptfreigabe
TXCIE	Transmit Complete Interrupt Enable	Lokale Interruptfreigabe
UDRIE	Data Register Empty Interrupt Enable	Lokale Interruptfreigabe
URSEL	Register Select	0: Zugriff auf UBRRH; 1: Zugriff auf UCSRC
UMSEL	Mode Select	0: asynchroner Modus, 1: synchroner Modus
UPM1/0	Parity Mode	00: keine Parität; 10: Gerade Parität; 11: Ungerade Parität
USBS	Stop Bit Select	0: 1 Stoppbit; 1: 2 Stoppbits
UCPOL	Clock Polarity	Polarität des Taktsignals im synchronen Modus

Werten belassen werden, die sie nach dem Resetvorgang des Controllers erhalten haben ($UCSRA = 0$).

Funktionen zur Initialisierung des USARTs und zum Polling-basierten Empfang beziehungsweise Senden von Daten können in der Programmiersprache C wie folgt realisiert werden:

```
// Initialisierung des USARTs:
void USART_init (unsigned int baudrate)
{
   unsigned int bdr = ((F_CPU+8*baudrate)/baudrate/16)-1;
   UBRRH = (bdr>>8)&0x7F;
   UBRRL = bdr&0xFF;
   UCSRB = (1<<RXEN)|(1<<TXEN);   // Empfänger und Sender akt.
   UCSRC = (1<<URSEL)|(3<<UCSZ0); // 8 daten, 1 stopp
```

```
}

// Daten mit USART empfangen:
unsigned char UART_rx(void)
{
   while (!(UCSRA & (1<<RXC)));    // auf Daten warten
   return UDR;                     // empf. Zeichen zurückgeben
}

// Daten mit USART senden:
void UART_tx(unsigned char data)
{
   while(!(UCSRA & (1<<UDRE)));    // warten auf Ende des Sendens
   UDR=data;                      // neues Zeichen ausgeben
}
```

14.7.4 SPI

Die Abkürzung *SPI* steht für *Serial Peripheral Interface*. Es handelt sich um eine synchrone Schnittstelle, die zur Datenübertragung unidirektionale Signalleitungen verwendet und zur Verbindung integrierter Bausteine verwendet wird. Zusätzlich zu den Datenleitungen wird ein Taktsignal übertragen, welches zur Synchronisation eingesetzt wird.

14.7.4.1 Datenübertragung mit dem SPI-Protokoll

Das Protokoll arbeitet nach dem Master-Slave-Prinzip. Ein SPI-Master initiiert die Datenübertragung und ist insbesondere für die Erzeugung des Taktsignals verantwortlich. SPI-Slaves empfangen das Taktsignal die vom Master übermittelten Daten. Gleichzeitig werden Daten vom Slave an den Master übertragen.

Für die Bezeichnung der Anschlüsse eines SPI-Interfaces sind keine allgemeingültigen Namen spezifiziert. Die in der Praxis häufig verwendeten Anschlussbezeichnungen sind in Tab. 14.32 zusammengefasst.

Sowohl der Master als auch der Slave enthalten Schieberegister, in die die Daten bitseriell eingeschrieben werden. Die Übernahme eines Bits in diese Schieberegister erfolgt mit der aktiven Taktflanke des Taktsignals SCK.

Häufig können auf der Seite des SPI-Masters die wesentlichen Übertragungsparameter konfiguriert werden. Hierzu zählen die Auswahl der aktiven Taktflanke (fallende oder steigende Flanke), die Wortlänge der Übertragung und die Auswahl, ob zuerst das höchstwertigste Bit (MSB first) oder das niederwertigste Bit (LSB first) übertragen werden sollen.

Die Auswahl, welcher Slave an der Kommunikation teilnehmen soll, erfolgt durch den Slave-Select-Anschluss *(/SS)* des Slaves. Wird dieser auf 0 gelegt, nimmt der Slave

Tab. 14.32 Anschlussbezeichnungen eines SPI-Interfaces

Signalbezeichnungen	Bedeutung	Datenrichtung
MOSI, SDI, SIMO	Daten *(Master Out, Slave In)*	Master → Slave
MISO, SDO, SOMI	Daten *(Master In, Slave Out)*	Slave → Master
SCK, SCLK	Takt *(Serial Clock)*	Master → Slave
/SS, /SSEL, /CS, /STE	Slaveauswahl *(Slave Select)*	Master → Slave

mit der nächsten aktiven Flanke des *SCK*-Signals an der Kommunikation der Bausteine teil. Andernfalls ignoriert der Slave die SPI-Übertragung.

Die Grundstruktur der Verbindung zwischen einem Master und einem Slave zeigt Abb. 14.14.

Ein Zeitdiagramm für die Signale *SCK, MOSI* und *MISO* ist in Abb. 14.15 dargestellt. In diesem Beispiel gilt für die SPI-Übertragung: Ruhezustand des Taktes ist 0 und die Datenübernahme findet mit der ersten Taktflanke nach Verlassen des Ruhezustands statt.

Sollen mehrere Slaves mit einem SPI-Master verbunden werden, können zwei Grundstrukturen verwendet werden, die im Folgenden als SPI-Kaskadierung oder als SPI-Sternverbindung bezeichnet werden.

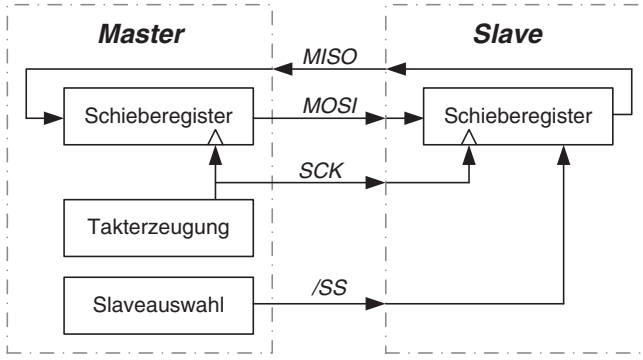
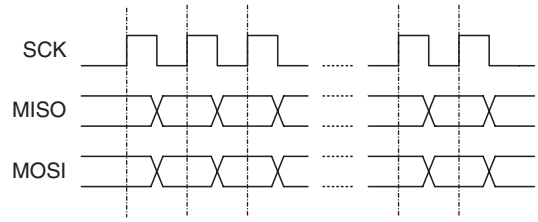


Abb. 14.14 Struktur der SPI-Verbindung zwischen einem Master und einem Slave

Abb. 14.15 SPI-Signalverlauf



Im Fall der SPI-Kaskadierung wird ausgenutzt, dass die Daten, die über den *MOSI*-Anschluss in einen Slave eingeschrieben werden, nach mehreren Taktzyklen unverändert am *MISO*-Ausgang des Slaves erscheinen. Wird dieser *MISO*-Ausgang mit dem *MOSI*-Eingang eines nachfolgenden Slaves verbunden, können somit „durch den ersten Slave hindurch“ Daten zu dem nachfolgenden Slave übertragen werden. Während der Übertragung von Daten mithilfe des SPI-Protokolls müssen alle */SS*-Eingänge auf dem Wert 0 gehalten werden. Hierfür kann eine gemeinsame */SS*-Leitung für alle kaskadierten SPI-Slaves verwendet werden. Die entsprechende Verbindungsstruktur ist in Abb. 14.16 exemplarisch für die Verbindung von einem Master und drei Slaves skizziert.

Die Alternative zur Kaskadierung stellt die SPI-Sternverbindung dar. Hierbei werden die *MISO*-Ausgänge der Slaves miteinander verbunden und an den *MISO*-Eingang des Masters angeschlossen. *MOSI*-Eingänge der Slaves werden mit dem *MOSI*-Ausgang des Masters verbunden. Um zu vermeiden, dass die Verbindung der *MISO*-Ausgänge der Slaves zu einem Kurzschluss führen kann, muss jeder der Slaves einzeln selektiert werden können. Wird vom Master nur einer der Slaves selektiert ($/SS = 0$), nimmt nur dieser an der Datenübertragung teil, während die Ausgänge der nicht selektierten Slaves hochohmig sind. Die SPI-Sternverbindung ist in Abb. 14.17 für einen Master und drei Slaves skizziert.

Der Vorteil der SPI-Kaskadierung ist der geringere Verdrahtungsaufwand. Bereits mit 4 Signalleitungen können beliebig viele Slaves an einen Master angeschlossen werden. Die Kaskadierung besitzt jedoch den Nachteil, dass die Daten durch alle Slaves hindurch gereicht werden müssen. Soll zum Beispiel der Slave 1 in Abb. 14.16 vom Master ausgelesen werden, so müssen die Daten des Slaves 1 zunächst durch die Slaves 2 und 3 geschoben werden, wodurch der Datentransfer mehr Zeit in Anspruch nimmt. Darüber hinaus ist zu beachten, dass die Slaves Daten unverändert durchreichen müssen. Diese Funktion wird von vielen Slaves nicht unterstützt und es muss die Sternverdrahtung gewählt werden. In diesem Fall ist jeder Slave direkt mit dem Master verbunden und die Zeit zur Übertragung zwischen dem Master und einem beliebigen Slave ist für alle

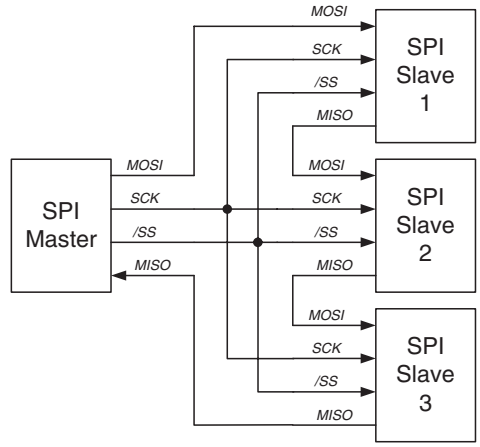


Abb. 14.16 SPI-Kaskadierung mit einem Master und drei Slaves

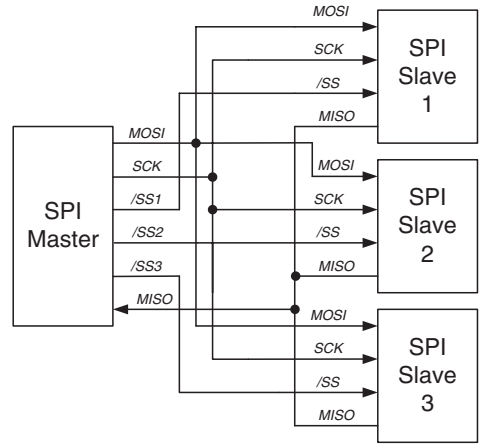


Abb. 14.17 SPI-Sternverbindung mit einem Master und drei Slaves

Slaves identisch. Diesem Vorteil steht der Nachteil gegenüber, dass für jeden Slave eine eigene */SS*-Leitung erforderlich ist.

14.7.4.2 SPI-Interface der AVR-Mikrocontroller

Die Mikrocontroller der AVR-Familie stellen ein SPI-Interface als eingebettete Peripheriekomponente zur Verfügung. Im Folgenden werden die Register der SPI-Schnittstelle eines *ATmega32* beschrieben. Der *ATmega32* stellt die SPI-spezifischen Anschlüsse als alternative Portfunktionen an den Anschlüssen *PB4 (/SS), PB5 (MOSI), PB6 (MISO)* und *PB7 (SCK)* zur Verfügung. Die Schnittstelle kann sowohl im Master- als auch im Slave-Modus betrieben werden.

Für die Programmierung der SPI-Schnittstelle stehen ein Steuerregister (SPI Control Register, *SPCR*), ein Statusregister (SPI Status Register, *SPSR*) und ein Datenregister (SPI Data Register, *SPDR*) zur Verfügung.

Die Belegung des Steuerregisters *SPCR* ist in Tab. 14.33 dargestellt. Das Register dient der Konfiguration der SPI-Schnittstelle. Die Bedeutung der einzelnen Bits dieses Registers ist in Tab. 14.34 zusammengefasst.

Die im Master-Modus erzeugte Frequenz des SPI-Taktsignals wird aus den Bits *SPR1*, *SPR0* und *SPI2X* (Bit 0 im Register *SPSR*) gemäß Tab. 14.35 aus dem Systemtakt abgeleitet.

Tab. 14.33 Belegung des Registers SPCR

SPCR								
Bit	7	6	5	4	3	2	1	0
Name	SPIE	SPE	DORD	MSTR	CPOL	CPHA	SPR1	SPR0

Tab. 14.34 Bedeutung der SPCR-Steuerregisterbits

Bit	Name	Bedeutung
SPIE	SPI Interrupt Enable	1: Lokale Interruptfreigabe Ein Interrupt wird jeweils nach der Übertragung eines Bytes ausgelöst.
SPE	SPI Enable	0: normale Portfunktion (SPI deaktiviert)/1: SPI-Schnittstelle aktiviert
DORD	Data Ordering	0: MSB first/1: LSB first
MSTR	Master Mode	0: Betrieb als SPI-Slave/1: Betrieb als SPI-Master
CPOL	Clock Polarity	Ruhezustand des Taktes (= Polarität des Taktsignals, wenn keine Übertragung stattfindet)
CPHA	Clock Phase	Mit diesem Bit wird festgelegt, welche Taktflanke verwendet wird: 0: Die erste Flanke nach Verlassen des Ruhezustands des Taktes ist die aktive Taktflanke 1: Die zweite Flanke nach Verlassen des Ruhezustands des Taktes ist die aktive Taktflanke

Tab. 14.35 Festlegung der SPI-Taktfrequenz mit den Bits SPR1, SPR0 und SPI2X

SPR1	SPR0	SPI2X	SPI-Taktfrequenz
0	0	1	f_{sys} / 2
0	0	0	f_{sys} / 4
0	1	1	f_{sys} / 8
0	1	0	f_{sys} / 16
1	0	1	f_{sys} / 32
1	0	0	f_{sys} / 64
1	1	1	f_{sys} / 128
1	1	0	f_{sys} / 256

Das Statusregister *SPSR* enthält neben dem Bit *SPI2X,* welches die Takterzeugung beeinflusst, zwei Statusbits. Das Bit SPIF wird von der Schnittstellen-Hardware auf 1 gesetzt sobald ein Byte übertragen wurde (Tab. 14.36).

Wird die SPI-Schnittstelle im Interruptbetrieb eingesetzt (*SPIE* = 1), wird das *SPIF*-Bit durch die Hardware mit Aufruf der zugehörigen ISR gelöscht. Im Polling-Betrieb muss zum Löschen des Bits zunächst das Register *SPSR* und anschließend das Datenregister *SPDR* gelesen werden.

Zum Lesen empfangener Daten beziehungsweise Schreiben zu sendender Daten steht das Register *SPDR* zur Verfügung. Vor dem Beginn einer Datenübertragung wird das zu sendende Byte in diesem Register abgelegt. Durch einen Lesezugriff auf dieses Register kann die CPU nach Beendigung einer Übertragung die empfangenen Daten auslesen.

Tab. 14.36 Belegung des Registers SPSR

SPSR								
Bit	7	6	5	4	3	2	1	0
Name	SPIF	WCOL	–	–	–	–	–	SPI2X

Während eine Datenübertragung aktiv ist, darf das Datenregister nicht geschrieben werden. Ein versehentliches Überschreiben des Datenregisters signalisiert die SPI-Schnittstelle durch Setzen des *WCOL*-Bits im Statusregister.

Bei vielen AVR-Controllern wird die SPI-Schnittstelle nicht nur zur Kommunikation mit anderen Bausteinen eingesetzt. Sie dient darüber hinaus als In-System-Programming-Schnittstelle *(ISP)*. Mithilfe der ISP-Funktion kann ein AVR-Controller, welcher in einem System eingesetzt ist, programmiert werden, ohne den Controller aus der Umgebung entfernen zu müssen. Diese Möglichkeit ist insbesondere für die Entwicklungsphase eines Systems bequem und zeitsparend.

Im Folgenden sind exemplarisch zwei Funktionen zur Initialisierung des SPI-Interfaces und zum Polling-basierten Empfangen und Senden von Daten angeben:

```
// Initialisierung des SPI-Interfaces
void SPI_init (void)
{
   DDRB |= (1<<PB4)|(1<<PB5)|(1<<PB7);    // SS,MOSI,SCK -> Ausgang
   SPCR |= (1<<SPE) |(1<<MSTR)|(1<<SPR0); // Schnittstelle konfigurieren
}

// SPI-Datenübertragung
unsigned char SPI_io(unsigned char snd_data)
{
   SPDR = snd_data;                 // Daten senden
   while (!(SPSR & (1<<SPIF)));     // Übertragung abwarten
   return SPDR;                     // empf. Daten zurückgeben
}
```

14.7.5 TWI/I²C

In den frühen 1980er Jahren führte die Firma Philips den *Inter-Integrated-Circuit-Bus (I²C)* ein. Mit diesem Bus ist es möglich, mehrere integrierte Bausteine (Mikrocontroller, A/D-Umsetzer, D/A-Umsetzer, Speicher usw.) auf einer Leiterplatte mit nur zwei Signalleitungen zu verbinden. Aufgrund der Anzahl der Signalleitungen bezeichnen einige Hersteller diese Schnittstelle auch als *TWI* (Two-Wire-Interface). Die Abkürzungen I²C und TWI können als synonyme Bezeichnungen identischer Schnittstellen aufgefasst werden.

Die I²C-Schnittstelle dient der synchronen seriellen Übertragung von Daten. Mithilfe des Signals *SCL (Serial Clock)* wird ein Taktsignal an alle angeschlossenen Bausteine übertragen. Der Datenaustausch findet über die Leitung *SDA (Serial Data)* statt.

Die I²C-Anschlüsse eines integrierten Bausteins sind als Open-Collector- beziehungsweise Open-Drain-Ausgänge realisiert. Die *SDA*- und *SCL*-Anschlüsse der einzelnen Komponenten sind miteinander verbunden und werden über einen Pull-Up-Widerstand mit der Versorgungsspannung verbunden. Ein Baustein kann die I²C-Leitungen aktiv auf einen Low-Pegel (logische 0) ziehen, er ist jedoch nicht in der Lage einen High-Pegel (logische 1) aktiv auszugeben. Ein High-Pegel auf einer der Signalleitungen wird erzielt, wenn alle Bausteine ihre Anschlüsse hochohmig schalten. Durch den Pull-Up-Widerstand (einige Kiloohm) wird dann eine logische 1 auf der Signalleitung erscheinen.

Abb. 14.18 zeigt den prinzipiellen Aufbau eines Systems mit mehreren integrierten Bausteinen, welche über eine I²C-Schnittstelle miteinander kommunizieren können.

Im Ruhezustand befinden sich alle I²C-Anschlüsse der Bausteine in einem hochohmigen Zustand, sodass beide Busleitungen über die Pull-up-Widerstände einen High-Pegel führen. Soll ein Datenaustausch zwischen zwei Komponenten stattfinden, muss einer der Bausteine das benötigte Taktsignal erzeugen und die Datenübertragung initiieren. Dieser Baustein übernimmt damit die Funktion eines I²C-Masters. Alle anderen Bausteine arbeiten dagegen als I²C-Slave.

14.7.5.1 Das I²C-Protokoll

Die Übertragung von Daten mithilfe des I²C-Protokolls erfolgt in zeitlich aufeinanderfolgenden Schritten.

Im ersten Schritt übermittelt der Master eine sogenannte *Startbedingung*. Anschließend wird eine 7 bit breite Bausteinadresse vom Master an die Slaves übermittelt. Ist die Bausteinadresse eines Slaves mit der übermittelten Adresse identisch, wird dieser Slave an der Kommunikation mit dem Master teilnehmen. Alle anderen, nicht ausgewählte Slaves, belassen ihre I²C-Anschlüsse in einem hochohmigen Zustand. In der Regel wird die I²C-Adresse eines Bausteins durch den Hersteller festgelegt. Häufig ist es möglich, einzelne Bits dieser Adresse durch die äußere Beschaltung (oder im Fall eines Mikrocontrollers durch das Programm der CPU) festzulegen. Auf diese Weise kann erreicht werden, dass mehrere identische Komponenten im gleichen Bussystem kollisionsfrei betrieben werden können. Nach der Übertragung der Bausteinadresse folgt ein einzelnes Bit, welches angibt, ob der Master Daten vom Slave lesen möchte oder ob Daten vom Master an den Slave übertragen werden sollen (0: Schreibzugriff, 1: Lesezugriff).

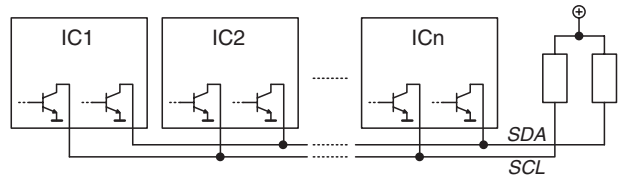


Abb. 14.18 Aufbau eines I²C-Systems mit mehreren integrierten Bausteinen

Nach der Übertragung der Adresse und der Schreib-/Leseinformation versetzt der Master seinen *SDA*-Anschluss in einen hochohmigen Zustand. Wurde ein Slave-Baustein durch die übertragene Adresse angesprochen, zieht dieser die *SDA*-Leitung für einen Taktzyklus auf Low. Auf diese Weise wird dem Master signalisiert, dass ein I²C-Slave mit der übertragenen Adresse im System existiert und dieser an der nachfolgenden Datenübertragung teilnimmt. Diese Bestätigung wird als *Acknowledge* bezeichnet.

Im nächsten Schritt erfolgt die eigentliche Datenübertragung. Für einen Schreibzugriff sendet der Master 8 Datenbits an den Slave, welcher den Empfang anschließend bestätigt. Bei einem Lesezugriff sendet dagegen der Slave Daten an den Master und der Master bestätigt den Empfang.

Nach der Übertragung eines Bytes können entweder weitere Bytes übertragen werden oder die Übertragung wird beendet. Zum Beenden einer Übertragung kann der Master entweder eine neue Startbedingung senden und so einen neuen Datentransfer einleiten oder der Master sendet eine sogenannte *Stoppbedingung,* welche das Ende der Übertragung signalisiert (s. Abb. 14.19).

Bei der Übertragung gemäß dem I²C-Protokoll gilt die Vereinbarung, dass sich der Wert der SDA-Leitung nur ändern darf, wenn die SCL-Leitung den Wert 0 besitzt. Diese Vereinbarung ist in Abb. 14.20 visualisiert.

Die oben genannte Vereinbarung gilt nur für die Adress- und Datenübertragung. Zur Signalisierung der Start- oder Stoppbedingung wird sie dagegen nicht eingehalten. Bei der Übertragung einer Startbedingung wird die SDA-Leitung vom Master auf Low gezogen während sich die SCL-Leitung noch im Ruhezustand (High) befindet. Entsprechend wird zur Übertragung einer Stoppbedingung zunächst die Taktleitung *SCL* von 0 auf 1 gesetzt. Mit einem anschließenden Wechsel der SDA-Leitung von 0 auf 1 wird wieder der Ruhezustand ($SDA = 1$, $SCL = 1$) erreicht. Der zeitliche Signalverlauf für Start- und Stoppbedingungen ist in Abb. 14.21 dargestellt.

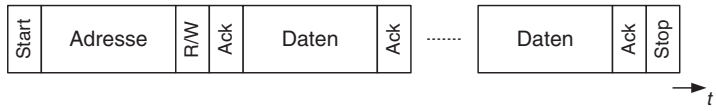
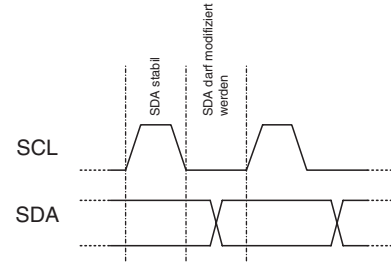


Abb. 14.19 Zeitlicher Verlauf einer I²C-Übertragung

Abb. 14.20 Synchronisierung beim I²C-Protokoll



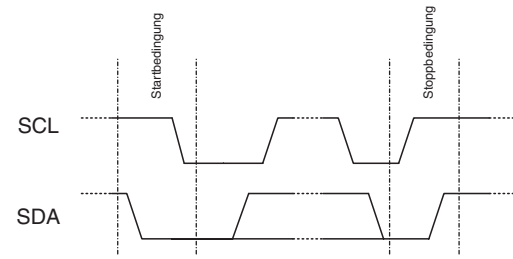


Abb. 14.21 Start- und Stoppbedingung des I²C-Protokolls

Die bitserielle Übertragung der Adressen oder Daten beginnt jeweils mit dem höchstwertigen Bit *(Most Significant Bit first, MSB first)*. Der zeitliche Verlauf einer Übertragung ist exemplarisch in Abb. 14.22 dargestellt. Der Master adressiert hierbei einen Baustein mit der Adresse 0x35 und empfängt vom Baustein den Wert 0xA5.

14.7.5.2 I²C-Interface der AVR-Mikrocontroller

Viele Mikrocontroller der AVR-Serie besitzen eine Hardware-Komponente, welche die Datenübertragung nach dem I²C-Protokoll unterstützt. Im Folgenden wird auf die

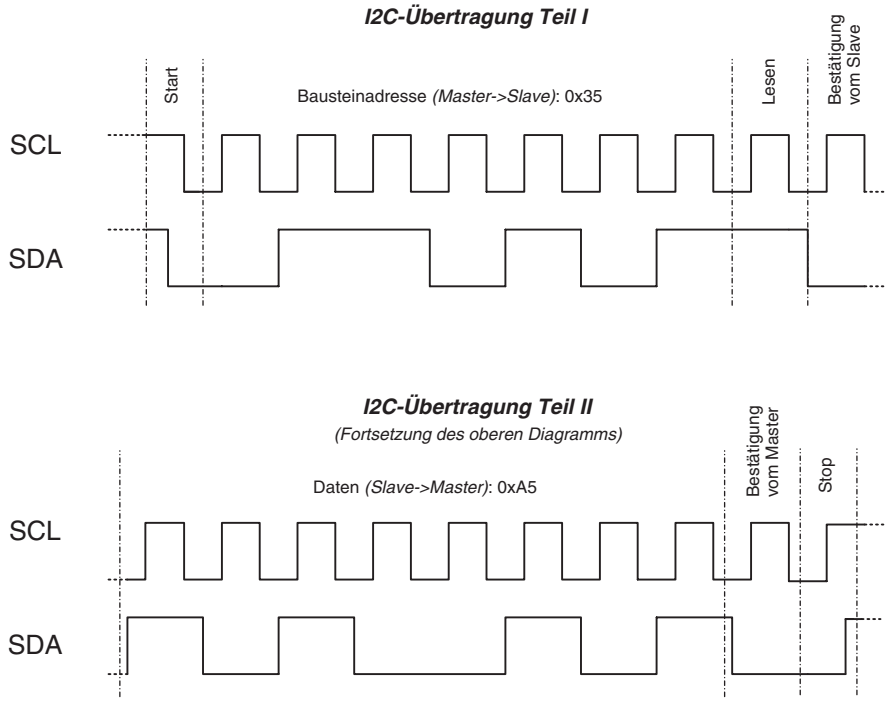


Abb. 14.22 Beispiel einer I²C-Übertragung

Schnittstelle eines *ATmega32*-Controllers eingegangen. Der Hersteller Atmel bezeichnet die I²C-Schnittstelle als *Two-Wire-Interface, TWI*.

Die I²C-Schnittstelle kann sowohl im Master- als auch im Slave-Betrieb arbeiten. Die Bausteinadresse für den Slavemodus kann durch eine entsprechende Programmierung durch den Controller frei festgelegt werden. Der AVR unterstützt *SCL*-Taktfrequenzen von bis zu 400 kHz, was dem sogenannten „Fast-Mode" entspricht. Da viele I²C-Bausteine nur den Standard-Mode mit einer Taktfrequenz von 100 kHz unterstützen, muss vor der Inbetriebnahme eines I²C-Systems überprüft werden, ob die gewählte Taktfrequenz von allen Bausteinen des Systems unterstützt wird.

Die Programmierung des I²C-Interfaces eines AVR ist sehr übersichtlich, da diese Hardwarekomponente lediglich 5 Register besitzt, die im Folgenden näher vorgestellt werden.

Das *TWI Control Register (TWCR)* dient zur Steuerung der I²C-Hardwarekomponente. Mithilfe dieses Registers kann die Komponente ein- oder ausgeschaltet oder die lokale Interruptfreigabe sowie einige Übertragungsparameter konfiguriert werden. Das *TWI Status Register (TWSR)* enthält 5 Bits, die als Statusinformation vom Programm ausgewertet werden können. Auf dieses Weise ist es möglich, Übertragungsfehler (zum Beispiel „Slave hat auf die Übertragung einer Adresse nicht mit einer Bestätigung geantwortet") im Programm zu erkennen. Das *TWSR*-Register enthält darüber hinaus zwei Bits *(TWPS1* und *TWPS0),* die zusammen mit dem Register *TWBR* (TWI Bitrate Register) die verwendete Taktfrequenz im Masterbetrieb festlegen. Hierbei wird I²C-Frequenz aus der Systemtaktfrequenz f_{sys} gemäß der nachfolgenden Formel abgeleitet:

$$f_{SCL} = \frac{f_{\text{sys}}}{16 + 2 \cdot TWBR \cdot 4^{TWPS}}$$

Mithilfe des TWI Slave Address Registers *(TWAR)* wird die vom Controller verwendete Bausteinadresse im Slave-Modus festgelegt. Die Übermittlung von Daten erfolgt mit dem TWI Data Register *(TWDR)*.

Die Belegung der Register *TWCR* und *TWSR* ist im Folgenden angegeben. Die anderen Register der I²C-Schnittstelle enthalten 8-Bit-Werte (Tab. 14.37, 14.38 und 14.39).

Mithilfe der *TWS*-Statusbits kann die CPU den aktuellen Zustand des I²C-Interfaces bestimmen. Hierbei wird der jeweilige Betriebsmodus (Master oder Slave) unterschieden. Darüber hinaus wird unterschieden, ob der AVR Daten empfängt (Receiver) beziehungsweise Daten sendet (Transmitter). Somit ergeben sich vier grundlegende

Tab. 14.37 Belegung des Registers TWCR

TWCR								
Bit	7	6	5	4	3	2	1	0
Name	TWINT	TWEA	TWSTA	TWSTO	TWWC	TWEN	-	TWIE

Tab. 14.38 Bedeutung der einzelnen Bits des TWCR-Registers

Bit	Name	Bedeutung
TWINT	TWI Interrupt Flag	1: Die I^2C-Komponente hat die zuvor programmierte Aufgabe abgearbeitet und kann von der CPU mit neuen Aufgaben belegt werden. Ein Löschen dieses Bits (durch Schreiben einer 1) startet die nachfolgende Aufgabe
TWEA	TWI Enable Acknowledge	1: Die Komponente generiert ein Bestätigungssignal, wenn Daten empfangen wurden oder falls (Slavemodus) die eigene Bausteinadresse empfangen wurde
TWSTA	TWI Start Condition	1: Startbedingung generieren
TWSTO	TWI Stop Condition	1: Stoppbedingung generieren (Mastermodus), Rücksetzen des Interfaces (zur Fehlerbehandlung im Slavemodus)
TWWC	TWI Write Collision	1: Das Datenregister (TWDR) wurde beschrieben bevor eine zuvor gestartete Übertragung abgeschlossen wurde
TWEN	TWI Enable	1: Die I^2C-Komponente ist aktiviert
TWIE	TWI Interrupt Enable	Lokale Interruptfreigabe

Tab. 14.39 Belegung des Registers TWSR

TWSR								
Bit	7	6	5	4	3	2	1	0
Name	TWS7	TWS6	TWS5	TWS4	TWS3	–	TWPS1	TWPS0

Betriebsmodi, für die eine Statusabfrage erfolgen kann: Master-Receiver-Modus, Master-Transmitter-Modus, Slave-Receiver-Modus und Slave-Transmitter-Modus.

In Tab. 14.40 sind die möglichen Statusinformationen für den Masterbetrieb zusammengefasst. Die in der Tabelle angegeben Konstanten können bei der Softwareentwicklung in der Programmiersprache C nach dem Inkludieren der Header-Datei *twi.h* verwendet werden.

Im Folgenden werden exemplarisch zwei Beispielfunktionen angegeben, welche das Polling-basierte Senden und Empfangen eines Bytes ermöglichen. Zur Fehlerbehandlung wird die Funktion *TW_ERR()* verwendet, die im nachfolgenden Code nicht angegeben ist und für ein lauffähiges Programm erstellt werden müsste.

```
// Senden eines Bytes
#include <util/twi.h> // I2C-Header-Datei
void TWI_ERR ()
{
   // Hier Code zur Fehlerbehandlung
}
void TWI_sendbyte (unsigned char twi_addr, unsigned char twi_data)
```

Tab. 14.40 Statusinformationen für den Masterbetrieb

Statusbits im Register TWSR	Konstante in C-Bibliothek	Modus	Bedeutung
0x08	TW_START	Alle	„START“ übertragen
0x10	TW_REP_START	Alle	„Repeated START“ übertragen *(Startbed. ohne vorherige Stoppbed.)*
0x18	TW_MT_SLA_ACK	Master Transmit.	Adresse + „Write“ übertragen, Bestätigung (ACK) empfangen
0x20	TW_MT_SLA_NACK	Master Transmit.	Adresse + „Write“ übertragen, Keine Bestätigung (NACK) empfangen
0x28	TW_MT_DATA_ACK	Master Transmit.	Daten übertragen, Bestätigung (ACK) empfangen
0x30	TW_MT_DATA_NACK	Master Transmit.	Daten übertragen, keine Bestätigung (NACK) empfangen
0x38	TW_MR_ARB_LOST	Master Receiver	Ein anderer Master hat die Kontrolle der I^2C-Leitungen übernommen
0x40	TW_MR_SLA_ACK	Master Receiver	Adresse + „Read“ übertragen, Bestätigung (ACK) empfangen
0x48	TW_MR_SLA_NACK	Master Receiver	Adresse + „Read“ übertragen, keine Bestätigung (NACK) empfangen
0x50	TW_MR_DATA_ACK	Master Receiver	Daten empfangen, Bestätigung (ACK) gesendet
0x58	TW_MR_DATA_NACK	Master Receiver	Daten empfangen, keine Bestätigung NACK gesendet

```
{
   // --- Startbedingung ---
   TWCR = (1<<TWINT)|(1<<TWSTA)| (1<<TWEN);      // Sende START
   while (!(TWCR & (1<<TWINT)));                 // gesendet?
   if (TWSR != TW_START) TWI_ERR();              // Status prüfen

   // --- Adresse ---
   TWDR = (twi_data << 1) | TW_WRITE;            // Adresse nach TWDR
   TWCR = (1<<TWINT) | (1<<TWEN);                // Übertragung starten
   while (!(TWCR & (1<<TWINT)));                 // Adr. gesendet?
   if (TWSR != TW_MT_SLA_ACK) TWI_ERR();         // Status prüfen

   // --- Daten ---
   TWDR = twi_data;                              // Daten nach TWDR
```

```
   TWCR = (1<<TWINT) | (1<<TWEN);              // Übertragung starten
   while (!(TWCR & (1<<TWINT)));               // Daten gesendet?
   if (TWSR != TW_MT_DATA_ACK) TWI_ERR();      // Status prüfen

   // --- Stoppbedingung ---
   TWCR = (1<<TWINT)|(1<<TWEN)| (1<<TWSTO);    // STOP senden
}
// Empfangen eines Bytes
#include <util/twi.h>                          // I2C-Header-Datei
unsigned char TWI_recbyte (unsigned char twi_addr)
{

   unsigned char twi_data;

   // --- Startbedingung ---
   TWCR = (1<<TWINT)|(1<<TWSTA)| (1<<TWEN);    // Sende START
   while (!(TWCR & (1<<TWINT)));               // gesendet?
   if (TWSR != TW_START) TWI_ERR();            // Status prüfen

   // --- Adresse ---
   TWDR = (twi_addr << 1) | TW_READ;           // Adresse nach TWDR
   TWCR = (1<<TWINT) | (1<<TWEN);              // Übertragung starten
   while (!(TWCR & (1<<TWINT)));               // Adr. gesendet?
   if (TWSR != TW_MR_SLA_ACK) TWI_ERR();       // Status prüfen

   // --- Daten ---
   TWCR = (1<<TWINT) | (1<<TWEN) | (1<<TWEA);
   while (!(TWCR & (1<<TWINT)));               // Daten empf. ?
     twi_data = TWDR;                          // Daten sichern

   // --- Stoppbedingung ---
   TWCR = (1<<TWINT)|(1<<TWEN)| (1<<TWSTO);    // STOP senden
   return twi_data;
}
```

14.7.6 Analoge Peripheriekomponenten

Neben digitalen Ein-/Ausgabekomponenten stellen Mikrocontroller vielfach auch analoge Komponenten zur Verfügung. Der im Rahmen dieses Kapitels exemplarisch betrachtete Mikrocontroller *ATmega32* verfügt über einen A/D-Umsetzer und einen Analog-Komparator. Im Folgenden werden diese Komponenten näher vorgestellt.

14.7.6.1 Analog/Digital-Umsetzer

Der A/D-Umsetzer arbeitet nach dem Verfahren der sukzessiven Approximation und stellt eine Auflösung von 10 bit zur Verfügung. Die Umsetzung erfolgt nach dem Verfahren der sukzessiven Approximation und benötigt, je nach Betriebsmodus, eine Zeit von 13 bis 260 µs. Als analoge Eingänge können im Fall des *ATmega32* die Anschlüsse *PA0 (ADC0)* bis *PA7 (ADC7)* verwendet werden. Insgesamt stehen somit 8 analoge Anschlüsse zur Verfügung. Durch eine entsprechende Konfiguration des integrierten Analog-Multiplexers ist es möglich, jeweils einen dieser Anschlüsse mit dem Eingang des A/D-Umsetzers zu verbinden und eine Messung der anliegenden Eingangsspannung durchzuführen. Darüber hinaus wird eine differenzielle Messung unterstützt, die es ermöglicht, die Spannungsdifferenz zweier analoger Anschlüsse zu messen. Die für den A/D-Umsetzer benötigte Referenzspannung kann entweder intern erzeugt oder über den Anschluss *AREF* beziehungsweise *AVCC* zugeführt werden. Die Struktur des A/D-Umsetzers ist in Abb. 14.23 gezeigt.

Mithilfe eines Eingangsmultiplexers werden die Anschlüsse des Controllers ausgewählt, die dem A/D-Umsetzer zugeführt werden sollen. Neben den Anschlüssen *ADC0* bis *ADC7* kann auch eine interne Referenzspannung oder eine Masseverbindung

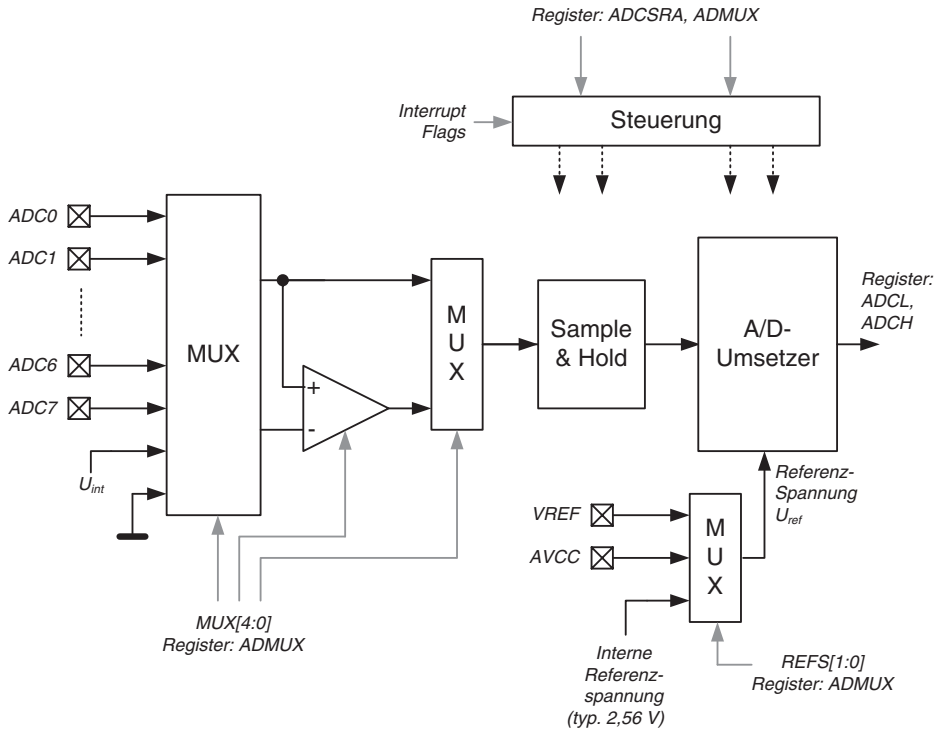


Abb. 14.23 Struktur des A/D-Umsetzers

ausgewählt werden. Bei Auswahl einer differenziellen Messung kann die Differenzspannung mithilfe eines Verstärkers mit den Werten 1, 10 und 200 multipliziert werden. Die ausgewählte Spannung wird zunächst über ein Sample-and-Hold-Glied geführt und anschließend dem A/D-Umsetzer zugeführt.

Nach der Durchführung der Wandlung kann der digitalisierte Wert aus den I/O-Registern *ADCL* und *ADCH* ausgelesen werden. Für diesen Wert gelten die folgenden Formeln:

Normale Messung (single-ended):

$$ADC = \frac{U_{\text{in}} \cdot 1024}{U_{\text{ref}}}$$

Differenzielle Messung:

$$ADC = \frac{(U_{\text{pos}} - U_{\text{neg}}) \cdot V \cdot 512}{U_{\text{ref}}}$$

mit: V – Verstärkungsfaktor

Für die Programmierung des A/D-Umsetzers werden drei Register verwendet: Das *ADC Multiplexer Selection Register (ADMUX),* das *ADC Control and Status Register A (ADCSRA)* sowie einige Bits des *Special Function IO Registers (SFIOR)*. Die Belegung der genannten Register ist im Folgenden angegeben (Tab. 14.41)

Mithilfe der Bits REFS1 und REFS0 wird die Referenzspannung für den A/D-Umsetzer ausgewählt *MUX4* bis *MUX0* steuern die Analogmultiplexer, und mithilfe des Bits *ADLAR* kann das Ausgabeformat in den Registern *ADCL* und *ADCH* ausgewählt werden. Tab. 14.42, 14.43 und 14.44 fassen die Bedeutung der Bits des *ADMUX*-Registers zusammen.

Tab. 14.41 Belegung des Registers ADMUX

ADMUX								
Bit	7	6	5	4	3	2	1	0
Name	REFS1	REFS0	ADLAR	MUX4	MUX3	MUX2	MUX1	MUX0

Tab. 14.42 Auswahl der Referenzspannung mit den Bits REFS1 und REFS0

REFS1	REFS0	Referenzspannung Uref
0	0	Anschluss AREF
0	1	Anschluss AVCC
1	0	reserviert
1	1	Interne 2.56 V Referenzspannung (Kapazität an AREF empfohlen)

Tab. 14.43 Bedeutung der Bits des ADMUX-Registers

ADCH (ADLAR = 0)								
Bit	7	6	5	4	3	2	1	0
Name	-	-	-	-	-	-	ADC9	ADC8
ADCL (ADLAR = 0)								
Bit	7	6	5	4	3	2	1	0
Name	ADC7	ADC6	ADC5	ADC4	ADC3	ADC2	ADC1	ADC0
ADCH (ADLAR = 1)								
Bit	7	6	5	4	3	2	1	0
Name	ADC9	ADC8	ADC7	ADC6	ADC5	ADC4	ADC3	ADC2
ADCL (ADLAR = 1)								
Bit	7	6	5	4	3	2	1	0
Name	ADC1	ADC0	-	-	-	-	-	-

Tab. 14.44 Auswahl der Eingangsmultiplexer in Abhängigkeit von MUX0 bis MUX4

MUX3	MUX2	MUX1	MUX0	Analogeingang (MUX4 = 0)	Analogeingang (MUX4 = 1)
0	0	0	0	ADC0	ADC0-ADC1
0	0	0	1	ADC1	ADC1-ADC1
0	0	1	0	ADC2	ADC2-ADC1
0	0	1	1	ADC3	ADC3-ADC1
0	1	0	0	ADC4	ADC4-ADC1
0	1	0	1	ADC5	ADC5-ADC1
0	1	1	0	ADC6	ADC6-ADC1
0	1	1	1	ADC7	ADC7-ADC1
1	0	0	0	(ADC0-ADC0)*10	ADC0-ADC2
1	0	0	1	(ADC1-ADC0)*10	ADC1-ADC2
1	0	1	0	(ADC0-ADC0)*200	ADC2-ADC2
1	0	1	1	(ADC1-ADC0)*200	ADC3-ADC2
1	1	0	0	(ADC2-ADC2)*10	ADC4-ADC2
1	1	0	1	(ADC3-ADC2)*10	ADC5-ADC2
1	1	1	0	(ADC2-ADC2)*200	Interne Spannung (1,22V)
1	1	1	1	(ADC3-ADC2)*200	Masse (0V)

Mit dem Register *ADSCRA* werden die Grundeinstellungen zum Betrieb des A/D-Umsetzers vorgenommen (Tab. 14.45 und 14.46).

Tab. 14.45 Belegung des Registers ADSCRA

ADSCRA								
Bit	7	6	5	4	3	2	1	0
Name	ADEN	ADSC	ADATE	ADIF	ADIE	ADPS2	ADPS1	ADPS0

Tab. 14.46 Bedeutung der Bits des Registers ADSCRA

Bit	Name	Bedeutung
ADEN	ADC Enable	Ein-/Ausschalten des A/D-Umsetzers (0: aus, 1: ein)
ADSC	ADC Start Conversion	1: Start einer A/D-Umsetzung Dieses Bit muss auch im „Free-Running-Mode“ (automatisch wiederholte Messungen) zum Start der ersten Wandlung gesetzt werden.
ADATE	ADC Auto Trigger Enable	0: Start der A/D-Umsetzungen durch SW 1: Kontinuierliche A/D-Umsetzung
ADIF	ADC Interrupt Flag	1: A/D-Umsetzung abgeschlossen (Löschen des Bits durch Schreiben einer 1)
ADIE	ADC Interrupt Enable	1: Lokale Interruptfreigabe. Auslösen einer Unterbrechung nach Abschließen der A/D-Umsetzung
ADPS	ADC Prescaler Selection	Auswahl des Taktes des A/D-Umsetzers

Tab. 14.47 Einstellung der ADC-Taktfrequenz durch Teilung der Systemfrequenz

ADPS2	ADPS1	ADPS0	ADC-Taktfrequenz
0	0	0	f_{sys} / 2
0	0	1	f_{sys} / 2
0	1	0	f_{sys} / 4
0	1	1	f_{sys} / 8
1	0	0	f_{sys} / 16
1	0	1	f_{sys} / 32
1	1	0	f_{sys} / 64
1	1	1	f_{sys} / 128

Tab. 14.48 Belegung des Registers TWSR

SFIOR								
Bit	7	6	5	4	3	2	1	0
Name	ADTS2	ADTS1	ADTS0	-	ACME	PUD	PSR2	PSR10

Tab. 14.49 Auswahl der Triggerquelle zum Start der A/D-Umsetzung

ADTS2	ADTS1	ADTS0	Trigger-Quelle zum Start einer A/D-Umsetzung
0	0	0	„Free Running Mode“: A/D-Umsetzung wird automatisch nach dem Beenden der vorangegangenen Umsetzung gestartet
0	0	1	Analog-Komparator
0	1	0	Externer Interrupt (Anschluss INT0)
0	1	1	Timer0: Zähler des Timers = Vergleichswert (Compare Match)
1	0	0	Timer0: Zählerüberlauf (Timer Overflow)
1	0	1	Timer1: Zähler des Timers = Vergleichswert B (Compare Match B)
1	1	0	Timer1: Zählerüberlauf (Timer Overflow)
1	1	1	Timer1: Ereignis der Input-Capture-Unit (Capture Event)

Die Taktfrequenz, mit welcher der A/D-Umsetzer betrieben wird, beeinflusst sowohl die Dauer der Umsetzung als auch die Genauigkeit des Ergebnisses. Für eine Genauigkeit von 10 Bit empfiehlt der Hersteller die Auswahl einer Frequenz zwischen 50 und 200 kHz. Ist eine geringere Genauigkeit ausreichend, kann der A/D-Umsetzer auch mit Frequenzen oberhalb von 200 kHz betrieben werden, um höhere Abtastraten zu erzielen. Eine Umsetzung dauert, je nach Betriebsmodus, zwischen 14,5 und 16,5 Taktzyklen. Die Auswahl der Taktfrequenz durch die CPU erfolgt durch Programmierung der *ADPS*-Bits im *ADSCRA*-Register. Die Taktfrequenz wird durch Teilung der Systemfrequenz f_{sys} entsprechend Tab. 14.47 erzeugt.

Neben dem softwarebasierten Start einer A/D-Umsetzung, kann eine Umsetzung auch durch controllerinterne Ereignisse ausgelöst werden. Zur Auswahl dieser Ereignisse müssen die *ADTS*-Bits *(ADTS: ADC Trigger Selection)* im Register *SFIOR* programmiert werden (Tab. 14.48).

Die Auswahl der möglichen Ereignisse zum Start einer Wandlung fasst Tab. 14.49 zusammen.

Eine einfache Beispielfunktion zur Verwendung des A/D-Umsetzers ist nachfolgend angegeben. Sie initialisiert den Umsetzer und startet eine Umsetzung, auf deren Ende Polling-basiert gewartet wird. Das Ergebnis wird als 16-Bit-Wert an das Hauptprogramm zurückgegeben.

```
#include <avr/io.h>
#include <util/delay.h>
unsigned int GET_ADC1()
{
   unsigned int adc;
   // Auswahl: Referenzspannung & Analogeingang ADC1
   ADMUX = (1<<REFS0) | (1<<MUX0);
   // A/D-Umsetzer einschalten und Vorteiler wählen
   ADCSRA = (1<<ADEN) | (1<<ADPS2) | (1<<ADPS1) | (1<<ADPS0);
```

```
    // Start der Umsetzung per Software
    ADCSRA |= (1<<ADSC);
    // Auf Ende der Umsetzung warten
    while (ADCSRA & (1<<ADSC)) _delay_us(1);
    // Ergebnis lesen
    adc = ADCL;
    adc |= (ADCH<<8);
    return adc;
}
```

14.7.6.2 Analog-Komparator

Mithilfe des Analog-Komparators können zwei analoge Spannungen miteinander verglichen werden. Das Ergebnis dieses Vergleichs wird vom Komparator als binärer Wert ausgegeben. Der Ausgangswert des Komparators kann durch die CPU über die Abfrage eines I/O-Registers eingelesen werden. Darüber hinaus ist es möglich, bei Änderungen des Ausgangswertes einen Interrupt auszulösen. Weiterhin kann der Ausgang des Analog-Komparators direkt in der Hardware des Mikrocontrollers (zum Beispiel im Timer 1 als Capture-Impuls für die Input-Capture-Unit) verwendet werden. Die Struktur des Komparators zeigt Abb. 14.24.

Die Funktion des Komparators wird im Wesentlichen durch das *ACSR*-Register festgelegt (Tab. 14.50).

Wie Tab. 14.51 zu entnehmen ist, erfolgt die Signalauswahl für den positiven Komparatoreingang durch das Bit *ACBG*. Der Multiplexer für den negativen Komparatoreingang wird über die Bits *ACME* (*SFIOR*-Register) und *ADEN* (*ADCSRA*-Register) gesteuert. Gilt $ACME = 1$ und $ADEN = 0$ (A/D-Umsetzer abgeschaltet), wird dem negativen Komparatoreingang das Ausgangssignal des Eingangsmultiplexers des A/D-Umsetzer zugeführt. In allen anderen Fällen wird das Signal am Anschluss *AIN1* ausgewählt.

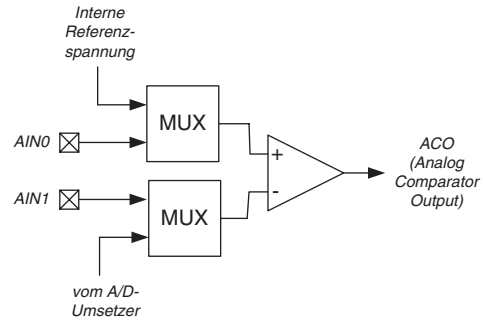


Abb. 14.24 Struktur des Analog-Komparators

Tab. 14.50 Belegung des Registers ACSR

ACSR								
Bit	7	6	5	4	3	2	1	0
Name	ACD	ACBG	ACO	ACI	ACIE	ACIC	ACIS1	ACIS0

Tab. 14.51 Signalauswahl am Komparatoreingang

Bit	Name	Bedeutung
ACD	Analog Comparator Disable	Ein-/Ausschalten des Komparators (0: ein, 1: aus)
ACBG	Analog Comparator Bandgap Select	Auswahl des Signals am positiven Komparator-Eingang 0: Anschluss AIN0 1: Intern erzeugte Referenzspannung (typ. 1,23 V)
ACO	Analog Comparator Output	Aktueller Status des Komparatorausgangs (zur Abfrage durch die CPU)
ACI	Analog Comparator Interrupt Flag	1: Ereignis (entspr. der ACIS-Bits) ist aufgetreten und es wird ein Interrupt ausgelöst, sofern der AC-Interrupt freigegeben ist
ACIE	Analog Comparator Interrupt Enable	1: Lokale Interruptfreigabe
ACIC	Analog Comparator Input Capture Enable	1: Ereignis des Analog-Komparators löst Capture-Event in Timer1 aus.
ACIS	Analog Comparator Interrupt Mode Select	00: Interrupt bei Wechsel des Ausgangs ACO 01: reserviert 10: Interrupt bei fallender Flanke des Ausgangs ACO 11: Interrupt bei steigender Flanke des Ausgangs ACO

14.7.7 Interrupt-basierte Kommunikation mit Peripheriekomponenten

Die Kommunikation zwischen CPU und eingebetteten Peripheriekomponenten kann Polling-basiert erfolgen. Beispiele hierzu wurden in den vorangegangenen Abschnitten für einzelne Peripheriekomponenten eines AVR-Mikrocontrollers dargestellt. Polling stellt die einfachste Möglichkeit dar, mit einer eingebetteten Komponente zu kommunizieren und besitzt den Vorteil, dass der Programmcode meist relativ gut nachvollziehbar ist, da er streng sequenziell ausgeführt wird. Mit Polling ist jedoch der Nachteil verbunden, dass ein signifikanter Anteil der verfügbaren Rechenleistung für Warteschleifen zur Abfrage von Peripheriekomponenten aufgebracht werden muss. Darüber hinaus muss bei Verwendung von Polling sichergestellt sein, dass die Komponenten ausreichend häufig abgefragt werden, da andernfalls Ereignisse (zum Beispiel der Empfang von Daten) verpasst werden könnten. Für sehr einfache Anwendungen kann Polling durchaus ein sinnvoller Ansatz zur Realisierung einer Anwendung auf einem Mikrocontroller sein. Für komplexere Anwendungen ist er meist nicht zu empfehlen, da entweder die rechtzeitige Abfrage aller Peripheriekomponenten nicht gewährleistet werden kann oder auch der Verbrauch der Rechenleistung für Warteschleifen zur Abfrage der Peripheriekomponenten nicht toleriert werden kann.

Als Alternative zu Polling kann eine interruptbasierte Kommunikation mit Peripheriekomponenten eingesetzt werden. Das Hauptprogramm wird in diesem Fall zunächst die Initialisierung des Systems vornehmen, die benötigten Interrupts lokal freigeben und abschließend eine globale Interruptfreigabe durch Setzen des I-Flags im Statusregister der CPU vornehmen. Anschließend wird das Hauptprogramm in eine Endlosschleife springen, die in einfachen Anwendungsfällen leer ist. Ein Beispiel für die Verwendung von Interrupts wird anhand des folgenden Beispiels verdeutlicht:

Mithilfe eines Mikrocontrollers soll eine einfache Temperaturüberwachung realisiert werden. Ein hypothetischer Sensor liefert die Temperatur als 8-Bit-Wert an den Mikrocontroller. Der Sensor misst kontinuierlich die Temperatur. Über die steigende Flanke eines Synchronisationssignals wird vom Sensor angezeigt, dass ein neuer Messwert ausgegeben wurde. Übersteigt die gemessene Temperatur einen vorprogrammierten Wert, soll der Mikrocontroller ein Alarmsignal ($Alarm = 1$) ausgeben. Eine Hardwarerealisierung auf Basis eines *ATmega32* ist in Abb. 14.25 skizziert.

Ein entsprechendes Programm für den Mikrocontroller kann wie folgt aussehen:

```
// Einfache Temperaturüberwachung
#include <avr/io.h>
#include <avr/interrupt.h>
#define ALARM_SCHWELLE 100
void InitSystem() {
   // Portrichtungen einstellen
   DDRA = 0;
   DDRB |= (1<<PB0);
   DDRD &= ~(1<<PD2);
   // Interrupt konfigurieren und lokal freigeben
   MCUCR |= (1<<ISC01)|(1<<ISC00);
   GICR |= (1<<INT0);
   // Globale Interrupt-Freigabe
   sei();
}
```

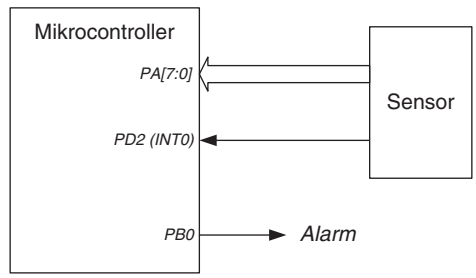


Abb. 14.25 Anwendungsbeispiel „Temperaturüberwachung“

```
// Interrupt-Service-Routine
ISR(INT0_vect) {
   // PB0 = 1 falls Alarmschwelle überschritten
   if (PINA > ALARM_SCHWELLE) PORTB |= (1<<PB0);
}
void main () {
   InitSystem();
   while (1) {
        // Sofern keine anderen regelmäßigen Aufgaben zu
        // erledigen sind, eine leere Endlosschleife…
   }
}
```

14.7.7.1 Interruptverarbeitung und atomare Operationen

Nun soll das System zur Temperaturüberwachung zunächst so erweitert werden, dass der Benutzer über eine entsprechende Schnittstelle den Schwellwert zur Auslösung eines Alarms einstellen kann. Zur Bedienung der Schnittstelle wird dem Programm die Funktion *UIF()* hinzugefügt. Diese Funktion könnte zum Beispiel mithilfe einer Tastatur und eines Displays mit dem Benutzer kommunizieren und den jeweils aktuell gewählten Schwellwert als Rückgabewert liefern. Die Implementierung dieser Funktion ist hier irrelevant und wird nicht näher betrachtet. Da die Benutzereingabe durch die Bedienung der Tastatur jedoch einige Zeit benötigt, muss berücksichtigt werden, dass die Ausführungszeit der Funktion nicht exakt bestimmbar ist und mehrere 100 ms oder auch mehrere Sekunden betragen kann. Ein erweitertes Programm, welches einen einstellbaren Alarmwert unterstützt, kann wie folgt realisiert werden.

```
// Temperaturüberwachung mit einstellbarem Alarmwert
#include <avr/io.h>
#include <avr/interrupt.h>

volatile unsigned char Schwelle;

void InitSystem() {
   // Programmcode wie oben angegeben
}

unsigned char UIF() {
   // User-Interface, die genaue Implementierung ist irrelevant
}

ISR(INT0_vect) {
   // PB0 = 1 falls Alarmschwelle überschritten
   if (PINA > Schwelle) PORTB |= (1<<PB0);
}
```

```
void main () {
   InitSystem();
   while (1) {
        Schwelle = UIF();
   }
}
```

Nun soll das Programm ein weiteres Mal erweitert werden. Es wird ein neuer Sensor verwendet, der einen 16 bit breiten Temperaturwert liefert. Der vom Sensor gelieferte Wert wird mithilfe der Ports PORTA und PORTC vom Mikrocontroller eingelesen. Der Code wird wie folgt modifiziert:

```
// Temperaturüberwachung mit einstellbarem 16-Bit-Alarmwert
// >>> Fehlerhafte Implementierung !!! <<<
// Modifikationen zum vorangegangenen Programm sind fett gedruckt
#include <avr/io.h>
#include <avr/interrupt.h>
volatile unsigned short Schwelle;
void InitSystem(void) { … }

unsigned short UIF(void) { … }

ISR(INT0_vect) {
   unsigned short Messwert;
   Messwert = PINA;
   Messwert = (Messwert<<8) | PINC;
   if (Messwert > Schwelle) PORTB |= (1<<PB0);
}

void main () {
   InitSystem();
   while (1) {
        Schwelle = UIF();
   }
}
```

Auf den ersten Blick mögen die Modifikationen des Programms plausibel und sinnvoll erscheinen: Ein lauffähiges und bewährtes Programm wurde durch die Modifikation der Wortbreite der verwendeten Variablen modifiziert. Allerdings würden bei Einsatz dieses Programms sporadische Fehlfunktionen auftreten. Um die Ursache dieser sporadischen Fehler zu verstehen, muss die Codezeile

```
Schwelle = UIF();
```

näher betrachtet werden.

Der Aufruf des Unterprogramms und die Zuweisung an die globale Variable *Schwelle* würde vom Compiler in den folgenden Assemblercode umgesetzt werden:

```
call   0x92; Aufruf von UIF, Rückgabewert in r24 und r25
sts    0x61, r25; Zuweisung des höherwertigen Bytes
sts    0x60, r24; Zuweisung des niederwertigen Bytes
```

Mit der Analyse des Assemblercodes wird das auftretende Problem deutlich: Der Compiler benötigt für die Zuweisung des Rückgabewertes an die 16-Bit-Variable *Schwelle* zwei Befehle. Sollte nun zufällig ein Sensor-Interrupt auftreten, während der erste Befehl der Zuweisung ausgeführt wird, würde die Interrupt-Service-Routine einen nicht vollständig erneuerten Wert in den Speicherstellen (hier: 0x60 und 0x61) der Variablen *Schwelle* vorfinden. Die erste Zuweisung würde dem Sprung in die ISR ausgeführt, während die zweite Zuweisung erst nach Verlassen der ISR aufgerufen wird.

In vielen Fällen wird sich dieser Programmfehler nicht bemerkbar machen, da mehrere Bedingungen zum Auftreten einer Fehlfunktion gelten müssen: Der Interrupt muss genau zum oben beschriebenen Zeitpunkt auftreten und das höherwertige Byte des Schwellwertes muss sich gegenüber dem vorangegangenen Wert geändert haben. Darüber hinaus müsste der vom Temperatursensor gelieferte Wert dazu führen, dass aufgrund des falsch übergebenen Schwellwertes ein Alarm fälschlich ausgelöst wird. Anhand dieser Überlegung ist zu erkennen, dass der Fehler vermutlich nur sehr selten auftreten wird. Genau hierin liegt jedoch die Schwierigkeit, den Fehler durch praktische Tests des Systems zu detektieren. Während der Entwicklungsphase tritt der Fehler aufgrund der geringen Auftrittswahrscheinlichkeit eventuell nicht zutage, hat jedoch im Betrieb des Systems möglicherweise fatale Folgen.

Anhand dieses einfachen Beispiels wird deutlich, dass man sich mit der CPU des verwendeten Systems auskennen sollte. In diesem Beispiel muss bei der Programmierung klar sein, dass eine 16-Bit-Zuweisung nicht durch einen einzelnen Befehl ausgeführt werden kann, da die CPU zwei aufeinanderfolgende 8-Bit-Zuweisungen verwenden muss.

Operationen, die nicht durch Interrupts (oder auch andere hier nicht näher betrachtete Mechanismen) unterbrochen werden können, werden auch als *atomare Operationen* bezeichnet. Der Begriff „atomar“ ist hierbei aus dem griechischen Wort átomo (= unteilbar) abgeleitet.

Die in dem Beispiel gezeigte Zuweisung eines 16-Bit-Wertes stellt somit keine atomare Operation dar, da sie durch einen Interrupt unterbrochen werden kann.

Zur Lösung dieser Problematik können die für die Zuweisung relevanten Interrupts kurzzeitig gesperrt werden. Eine mögliche Implementierung des Hauptprogramms könnte wie folgt aussehen.

```
// Ungünstige Implementierung des Hauptprogramms
void main () {
   InitSystem();
```

```
   while (1) {
        GICR &= ~(1<<INT0); // Interruptereignis INT0 sperren
        Schwelle = UIF();
        GICR |= (1<<INT0); // Interruptereignis INT0 freigeben
   }
}
```

Dieser Ansatz ist „interruptfest“. Es können also keine sporadischen Fehler aufgrund einer unvollständigen Zuweisung auftreten. Allerdings tritt hierbei eine weitere Problematik auf: Die Ausführung der Schleifenanweisung *(while(1))* und das Sperren beziehungsweise das Freigeben des INT0-Interrupts können von der CPU in wenigen Taktzyklen abgearbeitet werden. Für den Aufruf der Benutzerschnittstelle wird dagegen signifikant mehr Rechenzeit benötigt. Die Konsequenz ist, dass die Interrupts die überwiegende Zeit gesperrt sind. Daher ist die Wahrscheinlichkeit hoch, dass die ISR nicht aufgerufen wird und damit einige vom Temperatursensor gelieferten Werte nicht verarbeitet werden. Daher sollte bei der Programmentwicklung darauf geachtet werden, dass Interrupts nur so kurz wie möglich gesperrt werden.

Eine entsprechende Modifikation des Hauptprogramms könnte wie folgt aussehen.

```
// Sinnvollere Implementierung des Hauptprogramms
void main ()
{
   unsigned short Schwelle_lokal;
   InitSystem();
   while (1) {
        Schwelle_lokal = UIF();
        GICR &= ~(1<<INT0); // INT0 sperren
        Schwelle = Schwelle_lokal;
        GICR |= (1<<INT0); // INT0 freigeben
   }
}
```

Zusammenfassend lässt sich festhalten, dass die folgenden Überlegungen und Regeln bei der Verwendung von Interrupts beachtet werden sollten.

- Interrupts sollten, wenn überhaupt, so kurz wie möglich vom Hauptprogramm gesperrt werden.
- Da nach dem Aufruf einer ISR keine weiteren Interrupts zugelassen sind, sollte eine ISR eine möglichst kleine Rechenzeit benötigen.
- Für die Kommunikation zwischen dem Hauptprogramm und einer ISR sollte geprüft werden, ob die implementierte Kommunikation atomar ist. Gegebenenfalls sollte die Implementierung des Programms angepasst werden, um sporadische Fehler zu vermeiden.

14.7.7.2 FIFO-basierte Kommunikation mit Peripheriekomponenten

In vielen Fällen ist es wünschenswert, die Kommunikation mit einer Peripheriekomponente nicht byteweise auszuführen, sondern zunächst eine Zwischenspeicherung vorzunehmen. Da die Reihenfolge der Daten durch den Speicher nicht verändert werden soll, bietet sich die Implementierung eines First-In-First-Out-Speichers (FIFO) an. Eine mögliche FIFO-Realisierung ist im Folgenden dargestellt.

Die im Folgenden dargestellte Implementierung eines FIFOs verwendet zur Speicherung der Daten einen Bereich im SRAM des Controllers. Der Speicherbereich wird mithilfe der C-Bibliotheksfunktion *malloc()* reserviert. Zur Adressierung der Daten werden zwei Zeiger verwendet. Ein Schreibzeiger *(wp)* dient zur Adressierung der Daten, die in den FIFO-Speicher geschrieben werden. Ein Lesezeiger *(rp)* adressiert die Daten, die bei einem Lesezugriff auf das FIFO ausgeben werden. Die zur Verwaltung des FIFOs benötigten Daten (Schreibzeiger, Lesezeiger, Größe des FIFOs sowie ein Zeiger auf den im SRAM allokierten Speicherbereich) werden in einer Datenstruktur abgelegt.

Der allokierte Speicherbereich mit der Größe N Bytes wird als Ringspeicher genutzt. Für die Adressierung des Speichers bieten sich verschiedene Varianten an.

Bei der im Folgenden verwendeten Variante durchlaufen der Lese- und der Schreibzeiger einen Wertebereich von 0 bis $2N-1$. Das FIFO ist leer, wenn die Werte des Schreib- und des Lesezeigers identisch sind. Dagegen ist das FIFO voll, wenn die Differenz zwischen Schreib- und Lesezeiger genau N beträgt. Ein gesondertes Mitführen der „Voll/Leer"-Information oder des FIFO-Füllstandes ist bei dieser Variante nicht erforderlich. Werden die Zeiger als Bytevariablen ausgelegt, kann die atomare Ausführung des Codes sichergestellt werden, ohne dass Interrupts kurzzeitig gesperrt werden müssten. Der Nachteil dieses Ansatzes ist jedoch, dass die beiden Zeiger nur dann direkt zur Adressierung des Speichers verwendet werden, wenn ihre Werte kleiner als $N-1$ sind. Andernfalls muss vor der Adressierung vom Wert des Zeigers N subtrahiert werden.

Abb. 14.26 zeigt verschiedene Beispiele für mögliche Zustände der gewählten FIFO-Implementierung. Es ist jeweils der Füllstand (= Anzahl gültiger Werte im FIFO) sowie der Wert des Schreibzeigers und des Lesezeigers angegeben.

Die FIFO-Implementierung stellt verschiedene C-Funktionen zur Verfügung. Die Funktion *FIFO_Init()* allokiert Speicher für den Pufferspeicher zur Aufnahme der zu speichernden Daten und die Parameter des FIFOs (Schreibzeiger, Lesezeiger, FIFO-Größe und einen Zeiger auf den Pufferspeicher). Der Rückgabewert dieser Funktion ist ein Zeiger auf die angelegte Datenstruktur zur Verwaltung des FIFOs, die für die folgenden Funktionen als Parameter verwendet wird. Da die Allokation des Speichers dynamisch erfolgt, kann es vorkommen, dass der benötigte Speicherbereich zur Laufzeit des Programms nicht zur Verfügung steht. In diesem Fall ist der Rückgabewert der Funktion *NULL*.

Mithilfe der Funktionen *FIFO_Read()* beziehungsweise *FIFO_Write()* können Daten aus dem FIFO-Speicher gelesen beziehungsweise in das FIFO geschrieben werden. Diese Funktionen sind nicht blockierend („non-blocking"). Dies bedeutet, dass beispielsweise der Aufruf der Funktion *FIFO_Write()* auch bei einem vollen FIFO nicht wartet, bis ein Eintrag im FIFO frei wird. In diesem Fall wird die Funktion mit dem Rückgabewert 0

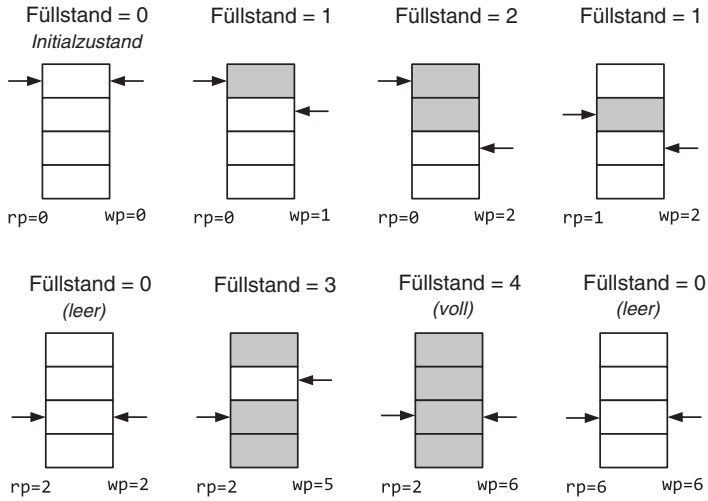


Abb. 14.26 Beispiele für Zustände der FIFO-Implementierung am Beispiel eines FIFOs mit 4 Einträgen

verlassen, um dem aufrufenden Programmteil anzuzeigen, dass der Schreibvorgang nicht erfolgreich ausgeführt wurde. Das aufrufende Programm kann mithilfe dieser Information entscheiden, ob die Fortsetzung des Programms sinnvoll ist oder gegebenenfalls in einer Warteschleife auf das Freiwerden eines Eintrages im FIFO warten und den Schreibvorgang erneut anstoßen. Entsprechendes gilt für die Funktion *FIFO_Read()*. Die Funktion *FIFO_Free()* gibt den mit *FIFO_Init()* belegten Speicherbereich wieder frei.

Der folgende Code zeigt eine mögliche FIFO-Implementierung in der Programmiersprache C.

```
// ************************************************************
// File:   fifo.h
// ************************************************************
#ifndef __FIFO_H__
#define __FIFO_H__
#include <stdlib.h>
// FIFO Struktur zur Aufnahme der FIFO-Parameter
typedef volatile struct {
   unsigned char size;    // FIFO Größe
   unsigned char rp;      // Lesezeiger
   unsigned char wp;      // Schreibzeiger
   unsigned char *buffer; // Zeiger auf Pufferspeicher
} TS_Fifo;
// FIFO Initialisierung (Speicher wird mittels malloc allokiert)
extern TS_Fifo* FIFO_Init(unsigned char log2size);
// FIFO Speicher freigeben
```

```
extern void FIFO_Free(TS_Fifo *fifo);
// Wert aus FIFO lesen
extern unsigned char FIFO_Read(TS_Fifo *fifo, unsigned char* value);
// Wert in FIFO schreiben
extern unsigned char FIFO_Write(TS_Fifo *fifo, unsigned char value);
#endif
// ***********************************************************
// File:   fifo.c
// ***********************************************************
#include "fifo.h"
TS_Fifo* FIFO_Init(unsigned char size)
{
   TS_Fifo *fifo;
   if (size>127) return NULL;
   fifo = malloc(sizeof(TS_Fifo));
   if (fifo==NULL) return NULL;
   fifo->buffer = malloc(size);
   if (fifo->buffer==NULL) {
        free((void*)fifo);
        return NULL;
   }

   fifo->size = size;
   fifo->rp = 0;
   fifo->wp = 0;
   return (fifo);
}

void FIFO_Free(TS_Fifo *fifo)
{
   free((void*)fifo->buffer);
   free((void*)fifo);
}
unsigned char FIFO_Read(TS_Fifo *fifo, unsigned char *value)
{
   unsigned char wp_tmp;
   unsigned char rp_tmp;
   unsigned char rp_adr;
   wp_tmp = fifo->wp;
   rp_tmp = fifo->rp;
   rp_adr = (rp_tmp>=fifo->size)? rp_tmp-fifo->size:rp_tmp;
   if (wp_tmp==rp_tmp) {       // FIFO leer ?
        return 0;
   } else {
        *value = fifo->buffer[rp_adr]; // Wert holen
        rp_tmp++;                // Lesezeiger erhöhen
```

```
        // Bei Überlauf rp auf 0 setzen
        if (rp_tmp==2*fifo->size) rp_tmp = 0;
        fifo->rp = rp_tmp;       // atomare Zuweisung
        return 1;
    }
}
unsigned char FIFO_Write(TS_Fifo *fifo, unsigned char value)
{
    unsigned char wp_tmp;
    unsigned char rp_tmp;
    unsigned char wp_adr;
    unsigned char rp_adr;
    wp_tmp = fifo->wp;
    rp_tmp = fifo->rp;
    wp_adr = (wp_tmp>=fifo->size)?wp_tmp-fifo->size:wp_tmp;
    rp_adr = (rp_tmp>=fifo->size)?rp_tmp-fifo->size:rp_tmp;
    if (wp_adr==rp_adr && wp_tmp!=rp_tmp) {
        // FIFO ist voll
        return 0;
    } else {
        // Wert in FIFO eintragen
        fifo->buffer[wp_adr] = value; // Wert schreiben
        wp_tmp++;                     // Schreibzeiger erhöhen
        // Bei Überlauf wp auf 0 setzen
        if (wp_tmp==2*fifo->size) wp_tmp = 0;
        fifo->wp = wp_tmp;            // atomare Zuweisung
        return 1;
    }
}
```

Die FIFO-Funktionen können für die Kommunikation mit Peripheriekomponenten verwendet werden. Der nachfolgende Code verwendet FIFO-Speicher für die Kommunikation über den USART. Es werden zwei FIFOs angelegt. Ein FIFO nimmt die empfangenen Daten in einer Interrupt-Service-Routine auf und legt diese in einem Empfangs-FIFO *(rx_fifo)* ab. Die empfangenen Daten werden mithilfe der Funktion *UART_GetFifo()* an das Hauptprogramm übergeben. Für das Senden von Daten wird ein weiteres FIFO *(tx_fifo)* verwendet. Das Hauptprogramm legt Daten durch Aufruf der Funktion *UART_PutFifo()* in diesem Sende-FIFO ab. Mithilfe einer ISR werden die Daten aus diesem FIFO ausgelesen und an die eingebettete serielle Schnittstelle des Mikrocontrollers übergeben.

```
// ***********************************************************
// File:   uart_fifo.h
// ***********************************************************
#ifndef __UART_H__
#define __UART_H__
```

```
#include <avr/io.h>
#include <avr/interrupt.h>
#include "fifo.h"
// Interruptbasierter Transfer mit FIFOs
extern unsigned char UART_InitFIFOTransfer (unsigned long baudrate,
     unsigned char rx_size, unsigned char tx_size);
// Zeichen aus RX FIFO abholen
// Rückgabewert ist 0 falls kein Zeichen verfügbar, sonst 1
extern unsigned char UART_GetFifo(unsigned char *data);
// Zeichen in TX FIFO schreiben
// Rückgabewert ist 0 falls Schreibpuffer voll, sonst 1
extern unsigned char UART_PutFifo(unsigned char data);
#endif

// ***********************************************************
// File:   uart_fifo.c
// ***********************************************************
#include "uart_fifo.h"
static volatile TS_Fifo *rx_fifo;
static volatile TS_Fifo *tx_fifo;
// Initialisierung des UARTs und der FIFOs
unsigned char UART_InitFIFOTransfer (unsigned long baudrate,
     unsigned char rx_size,     unsigned char tx_size)
{
   unsigned long bdrate;
   // Uebertragungsrate setzen
   bdrate = (F_CPU+baudrate*8)/(baudrate*16)-1;
   UBRRH = (bdrate>>8)&0xFF;
   UBRRL = bdrate&0xFF;
   //Uebertragungsformat: 8 data bits, no parity, 1 stop bit
   UCSRC = (1<<URSEL)|(1<<UCSZ1)|(1<<UCSZ0);
   // FIFOs initialisieren
   rx_fifo = FIFO_Init(rx_size);
   if (rx_fifo == NULL) return 0;
   tx_fifo = FIFO_Init(tx_size);
   if (tx_fifo == NULL) return 0;
   // UART einschalten
   UCSRB = (1<<RXEN)|(1<<TXEN);
   // Lokale Interruptfreigabe
   UCSRB |= (1<<RXCIE)|(1<<TXCIE);
   return 1;
}

// Wert in Sende-FIFO schreiben (Aufruf durch Hauptprogramm)
unsigned char UART_PutFifo(unsigned char data)
```

```
{
   unsigned char tmp_data;
   unsigned char num;
   // Wert in FIFO eintragen
   num = FIFO_Write(tx_fifo,data);
   // Falls Sendepuffer leer, Wert ausgeben
   if (UCSRA & (1<<UDRE)) {
        UCSRB &= ~(1<<TXCIE);
        FIFO_Read(tx_fifo,&tmp_data);
        UDR = tmp_data;
        UCSRB |= (1<<TXCIE);
   }
   return num;
}

// Wert aus Empfangs-FIFO lesen (Aufruf durch Hauptprogramm)
unsigned char UART_GetFifo(unsigned char *data)
{
   return (FIFO_Read(rx_fifo,data));
}

// Interrupt-Service-Routinen für Senden und Empfangen
ISR (USART_TXC_vect)
{
   unsigned char data;
   // Falls FIFO Daten enthält, diese übertragen
   if (FIFO_Read(tx_fifo,&data)) UDR = data;
}

ISR (USART_RXC_vect)
{
   // Empfangenen Wert in Empfangs-FIFO schreiben
   FIFO_Write(rx_fifo,UDR);
}
```

Ein einfaches Anwendungsbeispiel für die oben dargestellten Funktionen stellt das nachfolgende Hauptprogramm dar. Das Programm liest empfangene Daten ein und gibt diese über den USART unverändert wieder aus.

```
// ************************************************************
// File:   UartFifoDemo.c
// ************************************************************
#include "uart_fifo.h"
int main ()
{
   unsigned char data;
```

```
    if (UART_InitFIFOTransfer(9600,16,16)) {
         sei();
         while (1) {
             if (UART_GetFifo(&data)) UART_PutFifo(data);
         }
    }
}
```

14.8 Hinweise zum praktischen Selbststudium

In den vorangegangenen Abschnitten wurden die Grundlagen der Mikrorechnertechnik am Beispiel der AVR-Mikrocontroller-Familie behandelt. Die AVR-Controller zeichnen sich durch eine relativ einfache Struktur aus und sind für einen Einstieg in die Mikrorechnertechnik gut geeignet. Um das Verständnis der vorgestellten Themen zu vertiefen, ist es sehr empfehlenswert, eigene praktische Experimente mit Mikrocontrollern durchzuführen. Dieser Abschnitt soll einer ersten Orientierung dienen und so den Einstieg in das praktische Selbststudium erleichtern.

14.8.1 Hardwareauswahl

Für die AVR-Mikrocontroller werden von verschiedenen Herstellern zahlreiche Boards als Fertiggeräte oder als Bausatz angeboten. Neben dem Controller selbst stehen auf diesen Boards häufig auch weitere Bauteile wie LEDs, Taster, Lautsprecher oder Summer, LCD-Displays usw. zur Verfügung. In vielen Fällen steht auch eine Schnittstelle zur Verbindung mit einem PC zur Verfügung, mit welcher die entwickelten Programme in den Flashspeicher des Controllers übertragen werden können. Ein wesentliches Kriterium für die Auswahl eines Boards sollte neben dem Preis die Möglichkeiten zur Erweiterung durch eigene Schaltungsteile sein.

Eine Alternative zu bereits vorgefertigten Boards stellt die Anschaffung eines Steckbrettes dar. Viele Controller der AVR-Familie sind auch in Dual-Inline-Gehäusen (DIL) verfügbar. Mithilfe dieser Controller ist die Realisierung einfacher Systeme auf einem Steckbrett möglich.

14.8.2 Entwicklungsumgebungen

Für die AVR-Mikrocontroller steht die Entwicklungsumgebung *Atmel Studio* zur Verfügung, die kostenlos von der Homepage der Firma Atmel (www.atmel.com) heruntergeladen werden kann. Atmel Studio ist eine unter Windows-PCs lauffähige Entwicklungsumgebung, die neben der Erstellung von Programmen auch die Programmierung und das Debuggen der Controller unterstützt. Darüber hinaus besteht über einen integrierten Simulator die Möglichkeit, Programme auch ohne Anschaffung von Hardware zu testen.

14.8.3 Programmierung und Debugging von AVR-Mikrocontrollern

Die Übertragung von Programmen in den Flashspeicher eines AVR-Mikrocontrollers kann über die SPI-Schnittstelle des Controllers erfolgen. Ebenso können über diesen Weg auch Daten im eingebetteten EEPROM abgelegt werden. Für die Programmierung muss der Controller nicht aus der Zielapplikation entfernt werden. Daher wird dieser Vorgang als *In-System-Programming (ISP)* bezeichnet. Da das Protokoll zur Programmierung des Controllers offengelegt ist, sind verschiedene Programmiergeräte erhältlich, die eine Programmierung von AVR-Controllern unterstützen. Ein wesentlicher Nachteil des ISP-Verfahrens ist es, dass es nur zur Programmierung, nicht jedoch zum Debugging des Controllers verwendet werden kann.

Im Gegensatz zu ISP existieren für die AVR-Controller verschiedene Ansätze um ein Programm auch innerhalb des Systems zu debuggen. Hierbei können zum Beispiel Breakpoints gesetzt oder Variablenwerte ausgelesen werden, auch wenn sich der Mikrocontroller im Zielsystem befindet. Dieser als *On-Chip-Debugging (OCD)* oder *In-Circuit-Emulation (ICE)* bezeichnete Ansatz vereinfacht die Fehlersuche erheblich. Daher ist es auch für Einsteiger sinnvoll ein Programmiergerät anzuschaffen, welches das Debuggen im Zielsystem unterstützt. Hierbei ist jedoch darauf zu achten, dass nicht alle AVR-Controller den gleichen Ansatz verfolgen. Viele Controller der ATtiny-Serie unterstützen ein Verfahren, das von der Firma Atmel als *Debug-Wire* bezeichnet wird. Bei diesem Verfahren muss (abgesehen von der Versorgungsspannung des Mikrocontrollers) lediglich die Resetleitung des Controllers mit dem Programmiergerät verbunden werden, was insbesondere für Controller mit einer geringen Anzahl von Anschlüssen von Vorteil ist. Viele Controller der ATmega-Serie unterstützen dagegen ein Debuggen mittels eines JTAG-Interfaces. In diesem Fall müssen neben dem Reset-Anschluss auch die Anschlüsse TDO, TDI, TMS und TCK mit dem Programmieradapter verbunden werden. Diese Anschlüsse stehen dann nicht mehr uneingeschränkt als Portanschlüsse für die Zielapplikation zur Verfügung. Darüber hinaus kommt insbesondere bei den Mikrocontrollern der Xmega-Serie eine als „Program and Debug Interface“ (PDI) Schnittstelle zum Einsatz.

14.8.3.1 Programmiergeräte

Im Internet wird eine Vielzahl unterschiedlicher Programmiergeräte von diversen Herstellern angeboten. Sowohl die Preise wie auch die Funktionalität dieser Geräte differieren stark. Die günstigsten Geräte werden bereits ab ca. 15 EUR angeboten.

In der Regel ist es empfehlenswert, auf Originalgeräte der Firma Atmel zurückzugreifen. Auf diese Weise kann ausgeschlossen werden, dass Inkompatibilitäten des Programmieradapters zu Fehlern führen. Ein interessantes Gerät stellt der AVR-Dragon dar. Es unterstützt verschiedene Programmierprotokolle, unter anderem ISP und JTAG. Die Kosten für dieses Gerät liegen bei etwa 70 EUR.

14.8.3.2 Fuse-Bits

Bei der Durchführung eigener Experimente wird man recht schnell auf die sogenannten „Fuse-Bits“ stoßen. Fuse-Bits sind einzelne Bits, in der die Konfiguration des

Mikrocontrollers abgespeichert wird. Bei der Programmierung der Fuse-Bits ist besondere Vorsicht geboten. Wird durch eine falsche Programmierung des Fuse-Bits sowohl das ISP- als auch das JTAG-Interface gesperrt, ist eine weitere Programmierung des Controllers nicht mehr über diese Schnittstellen möglich. Um eine falsche Programmierung der Fuse-Bits zu korrigieren, muss der Controller in einem speziellen Programmiermodus betrieben werden, der nur von wenigen Geräten (zum Beispiel AVR-Dragon) unterstützt wird.

Im Folgenden werden die Fuse-Bits am Beispiel des *ATmega32* kurz erläutert:

OCDEN
Ist dieses Bit aktiviert, wird die oben beschriebene Möglichkeit des Debuggens im Zielsystem unterstützt.

JTAGEN
Mithilfe dieses Bits wird das JTAG-Interface zum Debuggen (*OCDEN* aktiviert) und/oder Programmieren des Controllers aktiviert.

SPIEN
Ist *SPIEN* aktiviert, kann die Programmierung des Controllers über die SPI-Schnittstelle mittels ISP erfolgen.

CKOPT
Dieses Bit findet Verwendung, wenn der Takt mithilfe eines Keramikresonators erzeugt wird und eine hohe Taktfrequenz benötigt wird. Im Normalfall sollte dieses Bit nicht aktiviert werden.

EESAVE
Bei Aktivierung eines sogenannten Chip-Erase-Cycles (Löschen des gesamten Chips) wird das EEPROM nicht gelöscht, wenn das Bit *EESAVE* aktiviert ist.

BOOTSZ, BOOTRST
Diese Bits ermöglichen es, den Einsprungpunkt nach einem Reset von der Programmspeicheradresse 0 an eine hohe Speicheradresse zu setzen. Mithilfe des so eingesprungenen Programms kann dann untere Bereich des Programmspeichers mit dem eigentlichen Applikationscode programmiert. Programme, die nach dem Reset den eigentlichen Programmcode laden, werden als Bootloader bezeichnet.

BODLEVEL, BODEN
Die AVR-Controller erlauben es, die Betriebsspannung kontinuierlich zu überwachen. Unterschreitet die Betriebsspannung einen vorprogrammierten Wert (Auswahlmöglichkeiten im Fall des *ATmega32*: 2,7 V oder 4,0 V), wird ein Reset ausgelöst. Diese auch als Brown-Out-Detection bezeichnete Möglichkeit kann unter anderem dazu genutzt werden, ein System bei einem Ausfall der Spannungsversorgung geordnet herunterzufahren.

SUT_CKSEL
Diese unter dem Namen SUT_CKSEL zusammengefassten Fuse-Bits dienen zur Auswahl der Takterzeugung für den Controller. Die gebräuchlichsten Fälle sind entweder die Verwendung des intern erzeugten Taktes oder die Aktivierung des eingebetteten Quarzoszillators, welcher an den Anschlüssen XTAL1 und XTAL2 einen externen Quarz benötigt. Darüber hinaus kann der Takt mithilfe eines externen RC-Gliedes, einem Keramikresonator oder von einer externen Quelle zugeführt werden.

Im Auslieferungszustand sind die Fuse-Bits der AVR-Controller mit sinnvollen Werten vorbelegt, sodass eine Neuprogrammierung in der Regel entfallen kann. Eine Ausnahme stellt die Programmierung der Taktauswahl dar. Im Auslieferungszustand ist für die Takterzeugung der eingebettete RC-Oszillator als Taktquelle ausgewählt. Viele Schaltungen verwenden jedoch einen externen Quarz zur Erzeugung des Taktsignals, sodass die *SUT_CKSEL*-Bits zunächst entsprechend programmiert werden müssen.

14.9 Übungsaufgaben

Die folgenden Übungsaufgaben greifen einige Themen dieses Kapitels auf. Die Lösungen finden Sie am Ende des Buches.

Sofern nicht anders vermerkt, ist nur eine Antwort richtig.

Aufgabe 14.1
Welche der folgenden Aussagen ist richtig? *(Mehrere Antworten sind richtig)*

a) Typische Mikrocontroller besitzen immer eine CPU.
b) Typische Mikrocontroller besitzen immer interne Speicherkomponenten.
c) Typische Mikrocontroller besitzen immer Ports.
d) Typische Mikrocontroller besitzen immer A/D-Umsetzer.

Aufgabe 14.2
Welche Aussagen sind richtig? *(Mehrere Antworten sind richtig)*

a) Die SPI-Schnittstelle wird zur asynchronen bitseriellen Datenübertragung verwendet.
b) Bei Verwendung einer I^2C-Schnittstelle erfolgt nach der Übertragung einer Startbedingung immer die Übertragung einer Bausteinadresse.
c) Das SPI-Protokoll verwendet getrennte Leitungen zur Übertragung von Daten vom Slave zum Master beziehungsweise vom Master zum Slave.
d) Das I^2C-Protokoll verwendet getrennte Leitungen zur Übertragung von Daten vom Slave zum Master beziehungsweise vom Master zum Slave.

Aufgabe 14.3
Welche Aussage über Unterprogramme ist richtig?

a) Beim Ausführen eines Unterprogramms wird dessen Code auf dem Stack abgelegt und anschließend ausgeführt.
b) Im aufrufenden Programmteil muss die Rücksprungadresse über geeignete Assemblerbefehle ermittelt und vor einem Unterprogrammaufruf auf dem Stack abgelegt werden.
c) Der aufrufende Programmteil kann einem Unterprogramm die Parameter über den Stack übergeben.
d) Der Code eines Unterprogramms muss im Programmspeicher immer vor dem Code des aufrufenden Programmteils abgelegt sein.

Aufgabe 14.4
Welche Aussage ist richtig?

a) Wird ein Wert auf dem Stack des AVR abgelegt, wird der Stackpointer dekrementiert.
b) Der Stackpointer des AVR kann nicht durch die Befehle eines Programms modifiziert werden.
c) Mithilfe des Befehls *pop* werden Daten auf dem Stack abgelegt.
d) Der Stackpointer der AVR-CPU zeigt immer auf den Wert, welcher als letztes auf dem Stack abgelegt wurde.

Aufgabe 14.5
Welche Aussage ist richtig?

a) Der AVR enthält nur Speicher, welche die gespeicherten Werte auch ohne Anliegen einer Versorgungsspannung halten können.
b) Der Programmspeicher des AVR kann nicht zur Speicherung von Daten verwendet werden, da kein Befehl existiert, mit dem der Programmspeicher gelesen werden kann.
c) Variablen eines C-Programms werden nicht im SRAM des AVR abgelegt.
d) Die Befehle eines AVR-Programms können nicht im EEPROM-Speicher abgelegt werden.

Aufgabe 14.6
Welche Aussagen sind richtig? *(Mehrere Antworten sind richtig)*

a) Ein typischer Timer kann so programmiert werden, dass beim Überlauf des timerinternen Zählers ein Interrupt ausgelöst wird.
b) Soll eine möglichst exakte Interruptrate (Interrupts pro Zeiteinheit) erzielt werden, sollte ein Timer bevorzugt im „CTC-Modus“ und nicht „Normal-Mode“ betrieben werden.
c) Eine Vorteiler-Einheit *(Prescaler)* ermöglicht es die Zählfrequenz eines Timers zu erhöhen.
d) Timer enthalten immer eine Input-Capture-Unit.

Aufgabe 14.7
Der Anschluss *PA2* eines *ATmega32* ist über einen Taster mit Masse (GND) verbunden. Am Anschluss *PC6* ist eine LED angeschlossen. Die LED leuchtet, wenn an *PC6* ein High-Pegel ausgegeben wird.

a) Erstellen Sie ein Programm in der Programmiersprache C, das die LED leuchten lässt, wenn der Taster geschlossen ist. Ist der Taster geöffnet, soll die LED nicht leuchten.
b) Realisieren Sie das Programm in Assembler.

Aufgabe 14.8
Mithilfe eines UARTs sollen Daten an einen PC übertragen werden. Für die Verbindung gilt: 8 Nutzdatenbits, keine Parität, 1 Stoppbit. Als Baudrate wird der Wert 9600 bps gewählt.

a) Skizzieren Sie den zeitlichen Verlauf des Signals am TXD-Anschluss des Controllers. Verwenden Sie für die Nutzdaten den Wert 0x35 (binär: 0011 0101).
b) Wie hoch ist die maximal erzielbare Netto-Datenrate (Daten-Bytes pro Sekunde)?
c) Nun wird auch ein Pariätsbit übertragen. Bei der Übertragung des Wertes 0x35 (binär: 0011 0101) sendet der Controller ein Paritätsbit mit dem Wert „1“. Welche Parität wurde gewählt?

Aufgabe 14.9
Der nachfolgende Code zeigt Ausschnitte eines AVR-Programms.

```
char  v8, *p8   // 8-Bit-Variable  bzw. Zeiger auf einen 8-Bit-Wert
short v16,*p16  // 16-Bit-Variable bzw. Zeiger auf einen 16-Bit-Wert
long  v32,*p32  // 32-Bit-Variable bzw. Zeiger auf einen 32-Bit-Wert
// hier weiterer Programmcode
v8  = 12;       // Zuweisung 1
p8  = &v8;      // Zuweisung 2
v16 = 1234;     // Zuweisung 3
p16 = &v16;     // Zuweisung 4
v32 = 12345678; // Zuweisung 5
p32 = &v32;     // Zuweisung 6
// hier weiterer Programmcode
```

a) Welche der Zuweisungen können mit einem AVR atomar ausgeführt werden?
b) Statt eines AVR wird ein 32-Bit-Mikrocontroller eingesetzt. Welche Zuweisungen sind nun atomar ausführbar?

Lösungen der Übungsaufgaben 15

In diesem Abschnitt finden Sie die Lösungen zu den Übungsaufgaben der vorangegangenen Kapitel.

Kapitel 1
Aufgabe 1.1 c
Aufgabe 1.2 b
Aufgabe 1.3 c
Aufgabe 1.4 b
Aufgabe 1.5 c
Aufgabe 1.6 b
Aufgabe 1.7 b
Aufgabe 1.8 c
Aufgabe 1.9 c
Aufgabe 1.10 b

Kapitel 2
Aufgabe 2.1
a) 111001_2
b) 71_8
c) 39_{16}

Aufgabe 2.2
a) 151
b) -105
c) 97

W. Gehrke et al., *Digitaltechnik,* Springer-Lehrbuch,
DOI 10.1007/978-3-662-49731-9_15

Aufgabe 2.3
a) 6 bit
b) 7 bit
c) 7 bit

Aufgabe 2.4
a) [0,255]
b) [−127,127]
c) [−128,127]

Aufgabe 2.5
a) 111101, kein Überlauf
b) 001011, Überlauf
c) 000100, Überlauf
d) Die Ergebnisse wären identisch
e) kein Überlauf bei a und c, Überlauf bei b

Aufgabe 2.6
a) 5A, Vorzeichenlos: kein Überlauf, 2er-Komplement: kein Überlauf
b) 23, Vorzeichenlos: Überlauf, 2er-Komplement: Überlauf
c) AB, Vorzeichenlos: Überlauf, 2er-Komplement: kein Überlauf

Aufgabe 2.7
a) 67, Vorzeichenlos: kein Überlauf, 2er-Komplement: Überlauf
b) 4C, Vorzeichenlos: kein Überlauf, 2er-Komplement: Überlauf
c) 9D, Vorzeichenlos: Überlauf, 2er-Komplement: Überlauf

Aufgabe 2.8
Wird ein Gray-codierter Wert inkrementiert, ändert sich das Codewort in genau einer Stelle.

Aufgabe 2.9
b. und c. sind Pseudotetraden

Aufgabe 2.10
a) Es werden 8 bit benötigt.
b) Es können 8 unterschiedliche Werte dargestellt werden.

Aufgabe 2.11
Überträgt man die Zweierkomplement-Darstellung auf das Dezimalsystem, entspräche die Codierung 999 dem Zahlenwert −1, da dies der Wert wäre, den man bei Durchlaufen des Zahlenkreises in negativer Richtung erhalten würde. Aus dieser Überlegung ergibt sich:

a) 000
b) 999

c) 998
d) 990

Kapitel 3
Aufgabe 3.1 a
Aufgabe 3.2 a, b, d
Aufgabe 3.3 a, b
Aufgabe 3.4 a, c
Aufgabe 3.5

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity my_module is

   port (a : in  std_logic_vector (7 downto 0);
         b : in  integer;
         c : in  std_logic;
         q : out std_logic_vector (7 downto 0) );
end;

architecture behave of my_module is
   signal tmp : unsigned (7 downto 0);
begin
   process
        variable vi : unsigned (7 downto 0);
   begin
        tmp <= unsigned(A);
        vi  := to_unsigned(B,8);
        if (c = '1') then
            q <= std_logic_vector(vi - tmp);
        else
            q <= std_logic_vector(vi + tmp);
        end if;
   end process;
end;
```

Aufgabe 3.6

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
```

```
entity my_module is
port (a : in  std_logic_vector (7 downto 0);
      b : in  std_logic_vector (7 downto 0);
      c : in  std_logic_vector (1 downto 0);
      q : out std_logic_vector (7 downto 0) );
end;

architecture behave of my_module is
begin
   process (a,b,c)
   begin
        if    c="00" then q <= a;
        elsif c="01" then q <= a and b;
        elsif c="10" then q <= a or b;
        elsif c="11" then q <= a xor b;
        -- std_logic! => c kann mehr als 4 Werte annehmen
        -- dies wird über das nachfolgende else abgefangen
        else  q <= (others=>'X');
        end if;
   end process;
end;
```

Aufgabe 3.7

```
   process (a,b,c)
   begin
        case c is
            when "00" =>  q <= a;
            when "01" =>  q <= a and b;
            when "10" =>  q <= a or b;
            when "11" =>  q <= a xor b;
            -- std_logic! => c kann mehr als 4 Werte annehmen
            -- also benötigen wir auch den "others"-Fall
            when others => q <= (others=>'X');
        end case;
   end process;
```

Aufgabe 3.8

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity my_module_16 is
port (a : in  std_logic_vector (15 downto 0);
      b : in  std_logic_vector (15 downto 0);
      c : in  std_logic_vector (1 downto 0);
```

```
      q : out std_logic_vector (15 downto 0) );
end;

architecture behave of my_module_16 is
begin
   my_module_inst1 : entity work.my_module
   port map (
         a => a(7 downto 0),
         b => b(7 downto 0),
         c => c,
         q => q(7 downto 0) );
     my_module_inst2 : entity work.my_module
     port map (
         a => a(15 downto 8),
         b => b(15 downto 8),
         c => c,
         q => q(15 downto 8) );
end;
```

Kapitel 4

Aufgabe 4.1 a

Aufgabe 4.2 a

Aufgabe 4.3

Die Funktionstabelle hat bei drei Eingangsvariablen acht mögliche Kombinationen. Schrittweise muss jeweils eine weitere LED eingeschaltet werden.

D2	D1	D0	L7	L6	L5	L4	L3	L2	L1
0	0	0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0	0	1
0	1	0	0	0	0	0	0	1	1
0	1	1	0	0	0	0	1	1	1
1	0	0	0	0	0	1	1	1	1
1	0	1	0	0	1	1	1	1	1
1	1	0	0	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1

Funktionstabelle „Lautstärke-LEDs“

Aufgabe 4.4

Die Funktionstabelle hat einen Eintrag ohne Tonausgabe (Mittelstellung), vier Einträge mit Ausgabe Ton T1 (Auslenkung in vier Richtungen) und vier Einträge mit Ausgabe Ton T2 (schräge Auslenkung in vier Ecken). Dies sind neun mögliche Kombinationen. Insgesamt sind für vier Eingänge 16 Kombinationen möglich, sodass für die übrigen sieben Kombinationen ein Don’t-Care eingetragen wird.

O (oben)	U (unten)	L (links)	R (rechts)	T1 (Ton 1)	T2 (Ton 2)
0	0	0	0	0	0
0	0	0	1	1	0
0	0	1	0	1	0
0	0	1	1	–	–
0	1	0	0	1	0
0	1	0	1	0	1
0	1	1	0	0	1
0	1	1	1	–	–
1	0	0	0	1	0
1	0	0	1	0	1
1	0	1	0	0	1
1	0	1	1	–	–
1	1	0	0	–	–
1	1	0	1	–	–
1	1	1	0	–	–
1	1	1	1	–	–

Funktionstabelle „Spielautomat“

Aufgabe 4.5
Produktterme 1 und 3 aus Abb. 15.1 sind erforderlich. Die Funktion für die Ausgangsvariable lautet:

$$Y = \overline{A(3)} \,\&\, A(2) \,\&\, A(0) \vee A(3) \,\&\, A(1)$$

Aufgabe 4.6
Alle Produktterme aus Abb. 15.2 sind erforderlich. Die Funktion für die Ausgangsvariable lautet:

$$Y = \overline{A(2)} \vee \overline{A(3)} \,\&\, \overline{A(1)} \,\&\, \overline{A(0)} \vee A(1) \,\&\, A(0)$$

Aufgabe 4.7
Produktterme 1 und 3 aus Abb. 15.3 sind erforderlich. Die Funktion für die Ausgangsvariable lautet:

$$Y = \overline{A(2)} \,\&\, A(0) \vee A(3)$$

Aufgabe 4.8
Alle Produktterme aus Abb. 15.4 sind erforderlich. Die Funktion für die Ausgangsvariable lautet:

$$Y = \overline{A(3)} \,\&\, \overline{A(1)} \,\&\, \overline{A(0)} \vee \overline{A(3)} \,\&\, A(2) \,\&\, \overline{A(1)} \vee A(3) \,\&\, A(1)$$

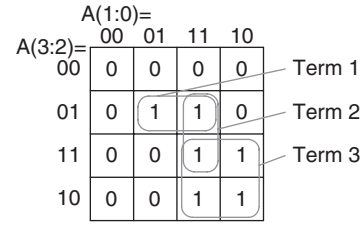


Abb. 15.1 Karnaugh-Diagramm zu Aufgabe 4.5

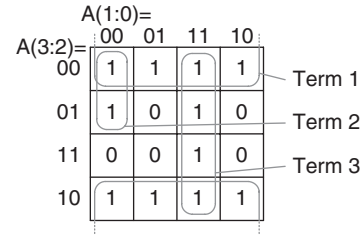


Abb. 15.2 Karnaugh-Diagramm zu Aufgabe 4.6

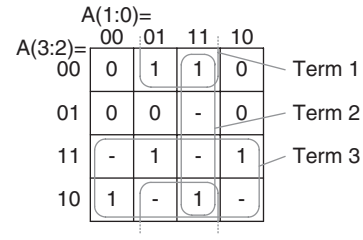


Abb. 15.3 Karnaugh-Diagramm zu Aufgabe 4.7

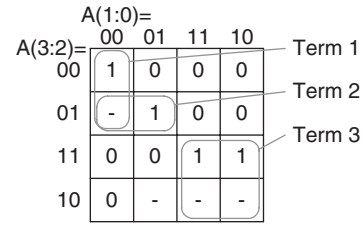


Abb. 15.4 Karnaugh-Diagramm zu Aufgabe 4.8

Kapitel 5
Aufgabe 5.1 a
Aufgabe 5.2 a
Aufgabe 5.3 d
Aufgabe 5.4 c
Aufgabe 5.5 c
Aufgabe 5.6 e
Aufgabe 5.7

a) Periodendauer $T = 100$ ns, Taktfrequenz $f = 10$ MHz, Duty-Cycle $D = 80$ %
b) Periodendauer $T = 1$ ms, Taktfrequenz $f = 1$ kHz, Duty-Cycle $D = 70$ %
c) Periodendauer $T = 0{,}5$ ms $= 500$ µs, Taktfrequenz $f = 2$ kHz, Duty-Cycle $D = 40$ %

Aufgabe 5.8
Das Codewort muss 4 Stellen für 11 Zustände besitzen. Die Berechnung kann über den Zweierlogarithmus von 11 erfolgen, der aufgerundet 4 ergibt.

$$ld11 = \log 11 / \log 2 = 1{,}041/0{,}301 = 3{,}46$$

Als alternativer Rechenweg können die Zweierpotenzen betrachtet werden. Mit 3 Stellen sind 2^3, also 8 Kombinationen möglich. Dies reicht nicht aus. 4 Stellen sind ausreichend, denn Sie ergeben 2^4, also 16 Kombinationen.

Aufgabe 5.9
Das Codewort muss 9 Stellen besitzen, denn die One-Hot-Codierung benötigt für jeden der 9 Zustände eine Stelle.

Aufgabe 5.10
Mit 5 Stellen sind 2^5, also 32 unterschiedliche Codierungen möglich.

Aufgabe 5.11
Es können 8 Zustände codiert werden, also genau so viele wie Stellen in der One-Hot-Codierung vorhanden sind.

Aufgabe 5.12
Der Automat benötigt vier Zustände mit den folgenden Bedeutungen:

- S0: Motor steht. Beim nächsten Tastendruck fährt die Jalousie herunter (Startzustand).
- S1: Taste ist gedrückt, der Motor fährt herunter.
- S2: Motor steht. Beim nächsten Tastendruck fährt die Jalousie herauf.
- S3: Taste ist gedrückt, der Motor fährt herauf.

Zustandsfolgediagramm und Zustandsfolgetabelle sind in Abb. 15.5 und 15.6 dargestellt.

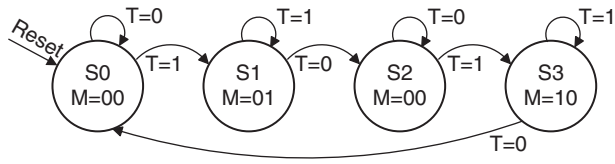


Abb. 15.5 Zustandsfolgediagramm des Automaten „Jalousie“

s^n	s^{n+1}		M
	T=0	T=1	
S0 *	S0	S1	00
S1	S2	S1	01
S2	S2	S3	00
S3	S0	S3	10

* = Reset

Abb. 15.6 Zustandsfolgetabelle des Automaten „Jalousie“

Aufgabe 5.13

Der Automat speichert in den Zuständen den bisher eingeworfenen Geldbetrag. Der Zustand mit der Bedeutung „50 Cent“ gibt an, dass die benötigte Summe erreicht ist und der Automat mit dem Ausgang P = 1 die Parkmünze ausgibt. Danach muss wieder neues Geld eingeworfen werden, das heißt, der Automat geht nach Ausgabe der Parkmünze wieder zu „0 Cent“.

- C_0: 0 Cent eingeworfen (Startzustand)
- C_10: 10 Cent eingeworfen
- C_20: 20 Cent eingeworfen
- C_30: 30 Cent eingeworfen
- C_40: 40 Cent eingeworfen
- C_50: 50 Cent oder mehr eingeworfen, Parkmünze wird ausgegeben

Der Startzustand war nicht ausdrücklich in der Aufgabenstellung angegeben, sondern ergibt sich durch Überlegung.

Zustandsfolgediagramm und Zustandsfolgetabelle sind in Abb. 15.7 und 15.8 dargestellt. Die beiden Eingänge werden in der kompakten Form „M(1:0)“ angegeben. Da der Eingang zwei Signale mit vier Kombinationsmöglichkeiten hat, sind für jeden Zustand vier Folgezustände möglich. In manchen Fällen sind einige dieser Folgezustände gleich.

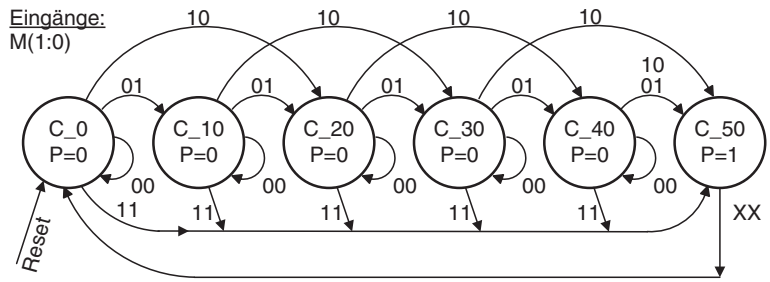


Abb. 15.7 Zustandsfolgediagramm des Automaten „Parkmünze“

s^n	s^{n+1}				P
	M= 00	01	10	11	
C_0*	C_0	C_10	C_20	C_50	0
C_10	C_10	C_20	C_30	C_50	0
C_20	C_20	C_30	C_40	C_50	0
C_30	C_30	C_40	C_50	C_50	0
C_40	C_40	C_50	C_50	C_50	0
C_50	C_0	C_0	C_0	C_0	1

* = Reset

Abb. 15.8 Zustandsfolgetabelle des Automaten „Parkmünze“

Übrigens werden im Zustand C_50 die Eingänge nicht ausgewertet. Der Automat geht nach einem Takt mit P = 1 wieder in den Startzustand. Dies ist möglich, da in der Aufgabenstellung spezifiziert ist, dass zwischen zwei Münzeinwürfen mehrere Takte vergehen.

Aufgabe 5.14

Der Automat muss sich weiterhin merken, ob die nächste 1 unterdrückt oder ausgegeben wird. Außerdem ist ein Zustand erforderlich, der nach der jeweils zweiten 1 die Ausgabe für einen Takt auf 1 setzt. Nach dieser Ausgabe wird die nächste 1 unterdrückt.

- S0: Nächste 1 unterdrücken, Ausgabe 0. (Startzustand)
- S1: Nächste 1 weitergeben, Ausgabe 0.
- S2: Gerade wurde die zweite 1 erkannt, 1 ausgeben, nächste 1 unterdrücken.

Zustandsfolgediagramm und Zustandsfolgetabelle sind in Abb. 15.9 und 15.10 dargestellt.

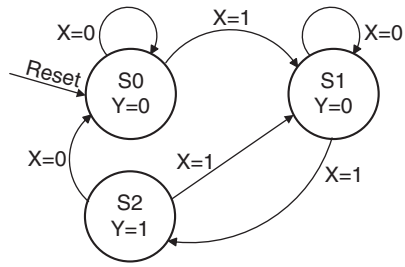


Abb. 15.9 Zustandsfolgediagramm des Automaten „Halbieren“

s^n	s^{n+1}		Y
	X=0	X=1	
S0 *	S0	S1	0
S1	S1	S2	0
S2	S0	S1	1

* = Reset

Abb. 15.10 Zustandsfolgetabelle des Automaten „Halbieren“

Aufgabe 5.15

Im Startzustand ist noch keine Stelle des Datenworts empfangen.

Wenn die erste Stelle empfangen wird, sind zwei Zustände erforderlich, die sich merken, erste Stelle empfangen und Wert 0 oder 1.

Wenn die zweite Stelle empfangen wird, können vier Fälle auftreten, und zwar: 00, 01, 10 und 11. Jetzt ist wichtig zu erkennen, dass der Automat nicht unterscheiden muss, ob 01 oder 10 empfangen wurde. Beide Fälle können den gleichen Zustand nutzen, denn der Automat muss sich nur merken, dass eine 1-Stelle auftrat. Wenn man weiterüberlegt, kann man erkennen, dass auch eine Unterscheidung von 00 und 11 nicht nötig ist. Darum sind für die vier Fälle nur zwei Zustände erforderlich, und zwar: „2 Stellen empfangen, ungerade“ und „2 Stellen empfangen, gerade“.

Das gleiche gilt nach drei Stellen, wo wieder zwei Zustände benötigt werden.

Beim Empfang der vierten Stelle wird eventuell das Fehlersignal $E = 1$ ausgegeben und der Automat geht direkt wieder in den Startzustand. Es ist also kein Zustand „4 Stellen empfangen“ nötig.

Insgesamt benötigt der Automat somit 7 Zustände:

- ST: Start, keine Stelle des Datenworts empfangen
- 1_G: Eine Stelle empfangen, Parität gerade. (Dies entspricht einer empfangenen 0. Die Bezeichnung wurde gewählt, da dies zu den folgenden Zuständen passt.)
- 1_U: Eine Stelle empfangen, Parität ungerade.
- 2_G: Eine Stelle empfangen, Parität gerade.
- 2_U: Eine Stelle empfangen, Parität ungerade.
- 3_G: Eine Stelle empfangen, Parität gerade.
- 3_U: Eine Stelle empfangen, Parität ungerade.

Zustandsfolgediagramm und Zustandsfolgetabelle sind in Abb. 15.11 und 15.12 dargestellt. Nach jeweils vier Takten ist die Bearbeitung eines Datenworts abgeschlossen und der Automat ist im Startzustand für das nächste Datenwort.

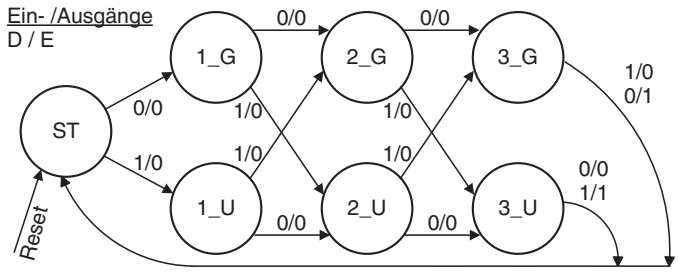


Abb. 15.11 Zustandsfolgediagramm des Automaten „Parity“

s^n	s^{n+1},E	
D=	0	1
ST *	1_G,0	1_U,0
1_G	2_G,0	2_U,0
1_U	2_U,0	2_G,0
2_G	3_G,0	3_U,0
2_U	3_U,0	3_G,0
3_G	ST,1	ST,0
3_U	ST,0	ST,1

* = Reset

Abb. 15.12 Zustandsfolgetabelle des Automaten „Parity“

Kapitel 6
Aufgabe 6.1 a
Aufgabe 6.2 e
Aufgabe 6.3 d
Aufgabe 6.4 c
Aufgabe 6.5 e
Aufgabe 6.6 c (4 Dateneingänge, 1 Datenausgang, 2 Steuerleitungen)
Aufgabe 6.7 d (1 Dateneingang, 8 Datenausgänge, 3 Steuerleitungen)

Aufgabe 6.8
Ein Modulo-2^10 Zähler durchläuft $2^{10} = 1024$ Werte, gerundet 1000 Werte. Bei 50 Mio. Werten pro Sekunde schafft der Zähler etwa 50.000 Zyklen pro Sekunde (Antwort b).

Aufgabe 6.9
Ein Modulo-2^8 Zähler durchläuft $2^8 = 256$ Werte, gerundet 250 Werte. Bei 500.000 Werten pro Sekunde schafft der Zähler etwa 2000 Zyklen pro Sekunde (Antwort b).

Aufgabe 6.10
Die Pipeline-Stufe sollte in der Mitte des kritischen Pfads eingefügt werden. Diese Position liegt in der Verbindungsleitung für den Übertrag nach vier Volladdierern. Die folgenden vier Volladdierer berechnen die zweite Hälfte der Addition im nächsten Taktzyklus. Damit die Informationen der Datenworte weiterhin zueinander passen, werden das Ergebnis der ersten vier Volladdierer sowie die Eingangswerte der nächsten vier Volladdierer jeweils um einen Takt verzögert. Die Addiererschaltung mit Pipelining zeigt Abb. 15.13.

Der kritische Pfad durchläuft 4 Addierer und besteht insgesamt aus:

- Flip-Flop Takt nach Ausgang: 0,2 ns
- 4 Volladdierer: $4 \cdot 0{,}3$ ns $= 1{,}2$ ns
- 5 Verbindungsleitungen: $5 \cdot 0{,}1$ ns $= 0{,}5$ ns
- Flip-Flop Setup-Zeit: 0,2 ns

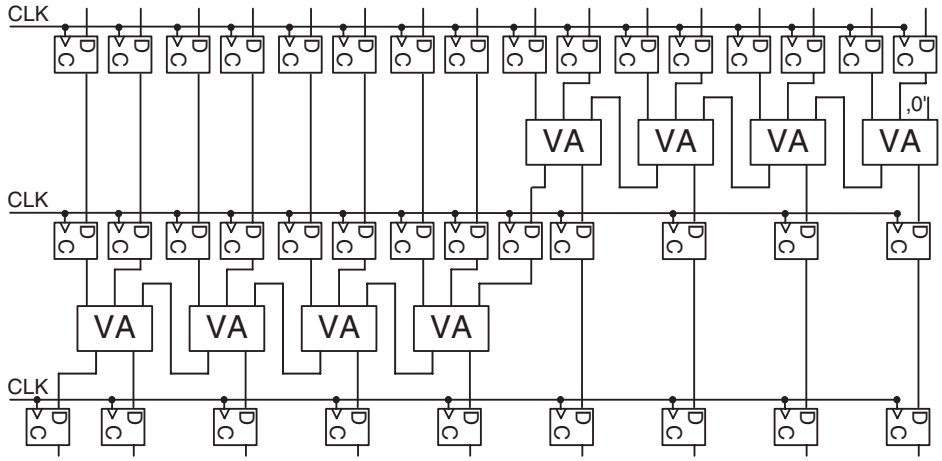


Abb. 15.13 Ripple-Carry-Adder mit Pipeline-Stufe

Dies ergibt in Summe 2,1 ns. Die mögliche Taktfrequenz beträgt damit rund 475 MHz.

Eventuell erscheint der Aufwand für das Pipelining in Abb. 15.13 recht hoch. Die ursprüngliche Schaltung hatte 8 Volladdierer und 25 Flip-Flop und erlaubt eine Taktfrequenz von 270 MHz. Für das Pipelining werden 13 zusätzliche Flip-Flops benötigt. Volladdierer und Flip-Flop sind ungefähr gleich groß, so dass der Mehraufwand 13 von 33 Elementen, also rund 40 % beträgt.

Im Gegenzug kann die Taktfrequenz, und damit die Rechenleistung, um 75 % gesteigert werden. Die theoretische Verdopplung der Taktfrequenz wird nicht erreicht, da das Pipeline-Flip-Flop eine Setup-Zeit sowie Verzögerungszeiten von Takt nach Ausgang und der Verbindungsleitung benötigt.

Kapitel 7
Aufgabe 7.1 b
Aufgabe 7.2 c
Aufgabe 7.3 d
Aufgabe 7.4 a, c
Aufgabe 7.5 b, c, d
Aufgabe 7.6 c
Aufgabe 7.7 a
Aufgabe 7.8 a, d

Kapitel 8
Aufgabe 8.1 b
Aufgabe 8.2 a, d
Aufgabe 8.3 b

Aufgabe 8.4 b
Aufgabe 8.5 b
Aufgabe 8.6 c

Kapitel 9
Aufgabe 9.1 a, b, d
Aufgabe 9.2 c
Aufgabe 9.3 d
Aufgabe 9.4 c
Aufgabe 9.5 c
Aufgabe 9.6 a, b, d

Kapitel 10
Aufgabe 10.1 a
Aufgabe 10.2 b
Aufgabe 10.3 a
Aufgabe 10.4 c
Aufgabe 10.5 a
Aufgabe 10.6 b
Aufgabe 10.7 d
Aufgabe 10.8 d
Aufgabe 10.9 e
Aufgabe 10.10

Nur wenn A und B beide 0 sind, ist die Reihenschaltung der beiden p-Kanal-Transistoren (oben) leitend und verbindet den Ausgang Y mit VDD. Wenn ein oder beide Eingänge 1 sind, verbindet die Parallelschaltung der n-Kanal-Transistoren (unten) den Ausgang Y mit GND.
Verhalten der Transistorschaltung

A	B	Y
0	0	1
0	1	0
1	0	0
1	1	0

Verhalten der Transistorschaltung

Dieses Verhalten entspricht der NOR-Funktion.

Kapitel 11
Aufgabe 11.1 d
Aufgabe 11.2 e

Aufgabe 11.3 d
Aufgabe 11.4 d
Aufgabe 11.5 c
Aufgabe 11.6 d
Aufgabe 11.7

a) Die Berechnung erfolgt am einfachsten über Zweierpotenzen. Mit 10 Adressleitungen lassen sich $2^{10} = 1024$ also 1 K Adressen ansprechen. Für den zusätzlichen Faktor 16 sind 4 Adressleitungen erforderlich, denn $2^4 = 16$. In der Summe werden $10 + 4 = 14$ Adressleitungen benötigt.
b) Zunächst werden wieder 10 Adressleitungen für 1 K Adressen benötigt. Für den zusätzlichen Faktor 256 sind 8 Adressleitungen erforderlich, denn $2^8 = 256$. In der Summe werden $10 + 8 = 18$ Adressleitungen benötigt.

Aufgabe 11.8
a) Mit 16 Adressleitungen lassen sich $2^{16} = 65.536$ Datenworten ansprechen. Jedes Datenwort hat 8 bit, somit beträgt die Speicherkapazität $65.536 \cdot 8 = 524.288$ bit.
In der Praxis wird oft der Faktor 1024 zu 1 K gerechnet. 16 Adressleitungen teilen sich dann auf in 6 Adressleitungen für den Faktor $2^6 = 64$ und $2^{10} = 1$ K, also 64 K Datenworte. Mit 8 bit je Datenwort ergibt sich 512 kbit Speicherkapazität.
b) 20 Adressleitungen entsprechen zweimal 10 Adressleitungen für 1 K Adressen, miteinander multipliziert 1 M Adressen. Mit 16 bit je Datenwort beträgt die Speicherkapazität 16 Mbit.
Der exakte Wert beträgt $2^{20} \cdot 16 = 16.777.216$ bit.

Aufgabe 11.9
Bei einer Dualzahl am Eingang des Speichermoduls entspricht die Reihenfolge der Speicherzellen zeilenweise ansteigenden Zahlen. Die erste Zeile entspricht also den Zahlen 0 bis 7, die zweite Zeile den Zahlen 8 bis 15, bis zur letzten Zeile mit den Zahlen 56 bis 63.

Primzahlen im möglichen Wertebereich 0 bis 63 sind die Zahlen: 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61.

Für die Primzahlen wird in die Speicherzelle eine 1 gespeichert, ansonsten eine 0. Das Ergebnis zeigt Abb. 15.14.

Kapitel 12
Aufgabe 12.1 A-2 B-1 C-3 D-4
Aufgabe 12.2 A-4 B-3 C-1 D-2
Aufgabe 12.3

a) Quantisierungsintervallbreite

$$Q = U_{\max}/2^n = 3\,\text{V}/1024 = 2{,}93\,\text{mV}$$

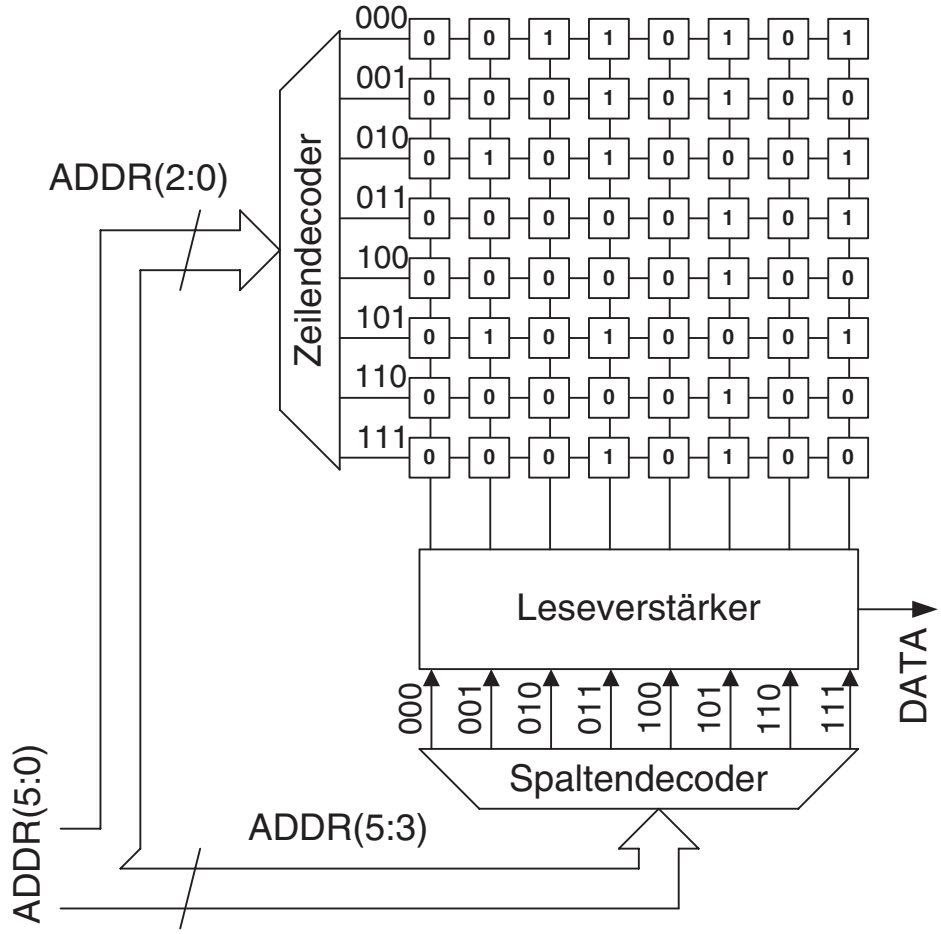


Abb. 15.14 Speichermodul als Primzahl-Detektor

b) Höchster codierbarer Spannungswert

$$U^*_{\max} = \left(2^n - 1\right) \cdot Q = 3\,\mathrm{V} \cdot (1023/1024) = 2{,}997\,\mathrm{V}$$

c) Die Eingangsspannung 1,2 V dividiert durch die Quantisierungsintervallbreite ergibt

$$\frac{1{,}2\,\mathrm{V}}{3\,\mathrm{V}/1024} = 409{,}6$$

Der gerundete Wert 410 entspricht der Codierung „01 1001 1010“.

d) Die Codierung „00 0100 1011“ entspricht dem Wert 75 und ergibt den Repräsentationswert

$$75 \cdot Q = 75 \cdot 3\,\text{V}/1024 = 0{,}2197\,\text{V}$$

Die Eingangsspannung liegt im Bereich der Quantisierungsintervallbreite um den Repräsentationswert

$$74{,}5 \cdot Q = 0{,}2183\,\text{V} \leq U_x \leq 0{,}2212\,\text{V} = 75{,}5 \cdot Q$$

Aufgabe 12.4

a) Quantisierungsintervallbreite

$$Q = U_{max}/2^n = 2\,\text{V}/256 = 7{,}8125\,\text{mV}$$

b) Schrittweiser Vergleich mit jeweils halber Spannung, beginnend bei $2^{n-1} \cdot Q = 1\text{V}$

- 0,7 V $\geq$ 1 V? Nicht erfüllt, also $b_7 = 0$
- 0,7 V $\geq$ 0,5 V? Erfüllt, also $b_6 = 1$ und Reduktion der Spannung um 0,5 V auf 0,2 V
- 0,2 V $\geq$ 0,25 V? Nicht erfüllt, also $b_5 = 0$
- 0,2 V $\geq$ 0,125 V? Erfüllt, also $b_4 = 1$ und Reduktion der Spannung um 0,125 V auf 0,075 V
- 0,075 V $\geq$ 0,0625 V? Erfüllt, also $b_3 = 1$ und Reduktion der Spannung um 0,0625 V auf 0,0125 V
- 0,0125 V $\geq$ 0,03125 V? Nicht erfüllt, also $b_2 = 0$
- 0,0125 V $\geq$ 0,015625 V? Nicht erfüllt, also $b_1 = 0$
- 0,0125 V $\geq$ 0,0078125 V? Erfüllt, also $b_0 = 1$ (letzter Schritt)

Als Digitalwert ergibt sich somit 0101 1001, also der Dezimalwert 89. Dies entspricht dem Repräsentationswert

$$89 \cdot Q = 89 \cdot 2\,\text{V}/256 = 0{,}6953\,\text{V}$$

Die Differenz zur Eingangsspannung von 0,7 V beträgt 4,7 mV und ist kleiner als die Quantisierungsintervallbreite.

Anmerkung: Der Quantisierungsfehler ist größer als $Q/2$. Dies liegt daran, dass das hier verwendete Berechnungsverfahren, wie im Text beschrieben, keine Rundung enthält, sondern Nachkommastellen abschneidet. Der rechnerische Ausgangswert wäre 0,7 V/(2 V/256) = 89,6, Wenn Sie *Q/*2 zum Eingangswert 0,7 V addieren, erhalten Sie mit dem Verfahren den korrekt gerundeten Digitalwert. Rechnen Sie erneut!

Aufgabe 12.5

Der Zeitablauf ist in der Tabelle dargestellt.

Zeit-schritt	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
U_x [in V]	−0,2	−0,2	−0,2	−0,2	−0,2	−0,2	−0,2	−0,2	−0,2	−0,2	−0,2	−0,2	−0,2	−0,2	−0,2
U_{dig} [in V]	0	−1	1	−1	1	−1	−1	1	−1	1	−1	−1	1	−1	1
U_{diff} [in V]	−0,2	0,8	−1,2	0,8	−1,2	0,8	0,8	−1,2	0,8	−1,2	0,8	0,8	−1,2	0,8	−1,2
U_{int} [in V]	−0,2	0,6	−0,6	0,2	−1	−0,2	0,6	−0,6	0,2	−1	−0,2	0,6	−0,6	0,2	−1
Plus [binär]	0	1	0	1	0	0	1	0	1	0	0	1	0	1	0

Sigma-Delta-Umsetzer mit Messbereich von ±1 V und Analogeingang $U_x = -0{,}2$ V.

Die Pulsfolge enthält zu 40 % den Wert 1. Dieser Anteil bezieht sich auf den Messbereich von ±1 V und entspricht

$$U_x = -1\,V + 0{,}4 \cdot 2\,\text{V} = \text{- } 1\,\text{V} + 0{,}8\,\text{V} = \text{- } 0{,}2\,\text{V}$$

Aufgabe 12.6
Im Zeitverlauf ist die Dauer des High-Pegels 8 ms bei einer Periodendauer von 10 ms. Dies entspricht einem Tastverhältnis von 80 %. Der High-Pegel ist 3 V und der Low-Pegel 0 V, so dass sich für die Ausgangsspannung ergibt

$$U_{DA} = 0\text{V} + \frac{8\,\text{ms}}{10\,\text{ms}} 3\,\text{V} = 2{,}4\,\text{V}$$

Kapitel 13
Aufgabe 13.1 c, d
Aufgabe 13.2 b
Aufgabe 13.3 c, d
Aufgabe 13.4 a, b, c, d
Aufgabe 13.5 c

Kapitel 14
Aufgabe 14.1 a, b, c (A/D-Umsetzer sind weit verbreitet, aber nicht immer vorhanden)
Aufgabe 14.2 b, c
Aufgabe 14.3 c
Aufgabe 14.4 a
Aufgabe 14.5 d
Aufgabe 14.6 a, b
Aufgabe 14.7

a)

```
#include <avr/io.h>

int main(void)
{
   // internen Pull-Up Widerstand für Tasteranschluss aktivieren
   PORTA |= 1<<2;
   // LED-Anschluss als Ausgang konfigurieren
   DDRC |= 1<<6;

   while (1){
            // Nachfolgende Bedingung liefert "true"
            // falls Taster nicht gedrückt
        if (PINA & (1<<2))
            DDRC &= ~(1<<6); // LED aus
        else
            DDRC |= 1<<6;    // LED an
   }
}
```

b)

```
.include "m32def.inc"

   ; Interner Pull-Up für Taster aktivieren
   in   r16, PORTA  ; PORTA nach r16
   ori  r16, (1<<2) ; Bit 2 setzen
   out  PORTA, r16  ; r16 wieder nach PORTA schreiben

   ; LED-Anschluss auf Ausgabe
   in   r16, DDRC   ; DDRC nach r16
   ori  r16, (1<<6) ; Bit 6 setzen
   out  DDRC, r16   ; und wieder in das Datenrichtungsregister schreiben

   ; Hier ist die Endlosschleife, in der
   ; der Taster abgefragt und die LED ein- oder ausgeschaltet wird
my_loop:
   in   r16,PINA    ; Wert von PINA holen
   andi r16,(1<<6)  ; nur Bit 6 ist von Interesse
   breq led_on      ; falls 0, springen
   in   r16, PORTC  ; sonst Bit 6
   andi r16,~(1<<6) ; in PORTC löschen
   out  PORTC, r16  ; und so LED ausschalten
   jmp  my_loop     ; fertig. Taster wieder abfragen
```

```
led_on:
   in   r16, PORTC  ; PORTC holen
   ori  r16, (1<<6); Bit 6 setzen
   out  PORTC, r16  ; und wieder nach PORTC schreiben => LED an
   jmp  my_loop      ; fertig. Taster wieder abfragen
```

Anmerkung: Der Code lässt sich auch kürzer schreiben, wenn die Befehle sbi und cbi verwendet werden.

Aufgabe 14.8

a) (Abb. 15.15).

b) 1 Frame besteht aus 10 Bit. Mit jedem Frame wird 1 Byte übertragen. Die Brutto-Datenrate beträgt 9600 bps. Also können 960 Bytes/s übertragen werden, wenn die Frames ohne Pause zwischen den Frames übertragen werden.

c) Die Anzahl der übertragenen Einsen (inklusive Paritätsbit) ist ungerade. Es wurde also *ungerade* Parität gewählt.

Aufgabe 14.9

Der AVR besitzt eine 8-Bit-CPU. Die Befehle verarbeiten also maximal 8 Bit. Werden Operanden mit einer größeren Wortbreite verarbeitet, sind hierfür mehrere Befehle notwendig. Zwischen der Ausführung zweier Befehle kann ein Interrupt ausgelöst werden. Also sind nur die Zuweisungen atomar, die mit einem einzelnen Befehl durchgeführt werden können.

Im Fall des AVR ist dies die Zuweisung an eine 8-Bit-Variable. Die Zuweisung an einen Zeiger ist nicht atomar, da der Zeiger eine größere Wortbreite als 8 bit besitzt und daher für die Zuweisung mehrere Befehle erforderlich sind. Im Fall einer 32-Bit-CPU sind alle Zuweisungen atomar (wenn vorausgesetzt wird, dass Zeiger eine maximale Wortbreite von 32 bit besitzen).

Für die Aufgabe ergibt sich also:

a) Nur Zuweisung 1 ist atomar.

b) Alle Zuweisungen sind atomar.

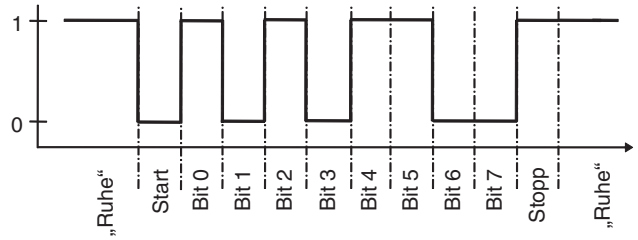


Abb. 15.15 Zeitverlauf des TXD-Signals bei Übertragung des Wertes 0×35

Literaturhinweise

Im Folgenden finden Sie Hinweise auf ergänzende und weiterführende Informationen, die wir nach den Themen des Lehrbuchs gegliedert haben.

Digitale Informationsverarbeitung und Grundlagen digitaler Schaltungen (Kapitel 1, 2, 4, 5, 6)

- C. Maxfield, „Bebop to the Boolean Boogie, An Unconventional Guide to Electronics“, Newnes, 2008.
- M. Alioto, E. Consoli, G. Palumbo, „Analysis and Comparison in the Energy-Delay-Area Domain of Nanometer CMOS Flip-Flops“, IEEE Trans. VLSI Systems, 2011.
- M. Aguirre-Hernandez, M. Linares-Aranda, „CMOS Full-Adders for Energy-Efficient Arithmetic Applications“, IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 2011.

Systementwurf mit VHDL (Kapitel 3, 8)

- P. Ashendon, „The Student's Guide to VHDL“, Morgan Kaufmann Publishers, 2008.
- B. Mealy, F. Tappero, „Free Range VHDL“, www.freerangefactory.org
- J. Bergeron, „Writing Testbenches: Functional Verification of HDL Models“, Springer 2003.
- J. Reichardt, B. Schwarz, „VHDL-Synthese“, De Gruyter Oldenbourg, 2015.
- A. Mäder, „VHDL kompakt“, Universität Hamburg, Fakultät für Mathematik, Informatik und Naturwissenschaften, tams-www.informatik.uni-hamburg.de/vhdl/doc/ajm-Material/vhdl.pdf

W. Gehrke et al., *Digitaltechnik,* Springer-Lehrbuch,
DOI 10.1007/978-3-662-49731-9

Schaltungsrealisierung (Kapitel 7, 10)

- K.-H. Cordes, A. Waag, N. Heuck, „Integrierte Schaltungen“, Pearson, 2010.
- H. Göbel, „Einführung in die Halbleiter-Schaltungstechnik“, Springer-Vieweg, 2014.
- L. Chen et.al., „Low Power Design Methodologies for Digital Signal Processors“, in N.N. Tan et.al. „Ultra-Low Power Integrated Circuit Design“, Springer 2014.
- I. Kuon, J. Rose, „Measuring the Gap Between FPGAs and ASICs“, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2007.

FPGAs und Komponenten digitaler Systeme (Kapitel 9, 11, 12)

- M. Qazi, M. E. Sinangil, A. P. Chandrakasan, „Challenges and Directions for Low-Voltage SRAM“, IEEE Design and Test of Computers, Jan/Feb 2011.
- J.M. de la Rosa, „Sigma-Delta Modulators: Tutorial Overview, Design Guide, and State-of-the-Art Survey“, IEEE Transactions on Circuits and Systems I, 2011.

Mikrocontroller (Kapitel 13, 14)

- J. Hennessy, D. Patterson, „Rechnerorganisation und Rechnerentwurf: Die Hardware/Software-Schnittstelle“, Oldenbourg, 2011.
- G. Schmitt, „Mikrocomputertechnik mit Controllern der Atmel AVR-RISC-Familie“, Oldenbourg, 2010.
- J. Wiegelmann, „Softwareentwicklung in C für Mikroprozessoren und Mikrocontroller“, VDE Verlag, 2011.

Weblinks

Für Informationen zu einzelnen Komponenten empfehlen wir die Herstellerseiten. In der nachfolgenden Übersicht sind einige Webseiten exemplarisch aufgeführt.

Standard-Logik:

- Texas Instruments: www.ti.com/lsds/ti/logic/home_overview.page
- NXP: www.nxp.com/products/logic

Programmierbare Logikbausteine (CPLDs, FPGAs):

- Xilinx: www.xilinx.com
- Altera: www.altera.com

- Lattice: www.latticesemi.com
- MicroSemi: www.microsemi.com

FPGA-Experimentierboards:

- Digilent: www.digilentinc.com
- Terasic: www.terasic.com

Speicher:

- Samsung: www.samsung.com/semiconductor/
- Hynix: www.skhynix.com
- Micron Technology: www.micron.com

AD/DA-Umsetzer:

- *Microchip:* www.microchip.com
- *Analog Devices:* www.analog.com

AVR-Mikrocontroller:

- allgemein: www.atmel.com/products/microcontrollers/avr/
- ATmega32: www.atmel.com/devices/ATMEGA32.aspx

Eine Übersicht über verschiedene Hersteller, sowie Information zu Preisen und Verfügbarkeit von Bauelementen bieten Distributoren bzw. Elektronikversandhändler, zum Beispiel

- Digikey: www.digikey.de
- Mouser: www.mouser.de
- Reichelt-Elektronik: www.reichelt.de
- Watterott: www.watterott.com

Viele Informationen zu digitalen Systemen und ein sehr gutes deutschsprachiges Forum finden Sie auf der Seite

- www.mikrocontroller.net

Stichwortverzeichnis

W. Gehrke et al., Digitaltechnik, Springer-Lehrbuch,
DOI 10.1007/978-3-662-49731-9