Technische Informatik

Prozessorarchitektur

Universität Ulm
Fakultät für Informatik

Abteilung Rechnerstrukturen

Wintersemester 1996/97

Prof. Dr. Jörg Kaiser

Script

Technische Informatik II

(vorläufige Version)

Vorwort

Das vorliegende Script umfaßt den Stoff der Prozessorarchitektur als Teil der Technischen Informatik II. Ziel ist, aufbauend auf den Grundlagen der Digitalen Logik und des Schaltwerkentwurfs, die Komponenten eines klassischen Rechners zu entwickeln und seine Programmierung darzustellen. Dabei soll ein Verständnis für die Probleme und die Zielkonflikte in der Rechnerarchitektur geweckt werden.

Leider sind einige Teile des Scripts noch in einem vorläufigen Stadium. Insbesondere die Kapitel 11, 12 und 13 liegen nur als Vorlesungsfolien vor. Der Stoff von Kapitel 13 lehnt sich allerdings sehr stark an das Buch von Patterson/Hennessy an, das im Literaturverzeichnis angegeben ist.

Für die Durchsicht des Scripts möchte ich Jörg Siedenburg und Axel Schmolitzky danken.

Für Anregungen und "Bug Reports" bin ich den Lesern dankbar.

Jörg Kaiser

Ulm, Januar 1997

Rechnerorganisation

Annotiertes Inhaltsverzeichnis

1. Einführung

2. Grundlagen elektronischer Schaltkreise und der Hochintegration

Inhalt: Die "stoffliche Grundlage" der Rechnerarchitektur und Organisation sind heute Halbleiter, welche die Logik der Rechner physikalisch umsetzen. Der vorliegende Teil der TI-II führt kurz die physikalischen Grundlagen der Halbleiter ein, erklärt die Basiselemente wie Diode und Transistor und diskutiert die Auswirkungen der Miniaturisierung. Ziel ist ein grundlegendes (informales) Verständnis der physikalischen Vorgänge elektronischer Logikbausteine.

3. Technische Grundkomponenten zur Realisierung von Rechnern

Inhalt: Die wichtigsten Eigenschaften der technischen Grundkomponenten von Rechnern werden eingeführt. Sie sollen erklären, wie die einzelnen Bausteine arbeiten und zu größeren Logikschaltungen kombiniert werden können. Ein wichtiger Aspekt vom Standpunkt der technischen Realisierung ist dabei, eine möglichst reguläre Struktur technisch vorzufertigen, auf die später die entsprechende anwendungsspezifische logische Funktion aufgeprägt werden kann. Dies führt zu den "programmierbaren Logikelementen". Diese Strukturen werden auch bei der Realisierung moderner Prozessoren angewendet.

4. Ein einfacher Modellrechner

Inhalt: Ausgehend von den Arbeitsprinzipien eines Rechners wird ein einfacher Befehlssatz eingeführt. Auf der Grundlage dieses Befehlessatzes wird eine Kontrolleinheit und eine ALU entwickelt, an der die Arbeitsweise der Steuerung eines Rechners im Detail gezeigt wird. Als Hilfmittel wird eine einfache, sogenannte Register-Transfer- Sprache eingeführt, die es ermöglicht die einzelnen Schritte bei der Bearbeitung eines Befehls im Detail zu beschreiben. Ziel ist es, das Zusammenspiel aller wesentlichen Komponenten eines Rechners auf der Logikebene zu zeigen.

5. Mikroprogrammierung

Inhalt: Am Beispiel des Modellrechners werden Entwurfsalternativen für die Kontrolleinheit diskutiert. Die Technik der Mikroprogrammierung wird eingeführt und es wird gezeigt, wie die Logik der festverdrahteten Kontrolleinheit des Modellrechners durch ein Mikroprogramm realisiert werden kann. Es werden Varianten der Mikro- und Nanoprogrammierung besprochen.

6. Assemblerprogrammierung

Inhalt: Die Grundlagen eines Assemblers werden eingeführt. Am Beispiel des 6809 Assemblers werden die Steuerung der Assemblierung durch Assemblerdirektiven gezeigt. Es werden verschiedene Arten von Assemblern kurz beschrieben.

7. Ein einfacher Mikroprozessor

Inhalt: Bedingt durch seine Einfachheit, hat der eingeführte Modellrechner eine Reihe von Unzulänglichkeiten, die seine Programmierung erschweren. Am Beispiel des Mikroprozessors MC6809 wird schrittweise das Speichermodell und der Befehlssatz erweitert und die Vorteile für den Komfort bei der Programmierung dargelegt. Neben einer Diskussion des MC 6809 Befehlssatzes werden Befehlsformate, Speichermodelle und Adressierungsarten behandelt.

8. Prozessornahe Programmiertechniken

Inhalt: Die Realisierung höherer Programmiersprachenkonstrukte in Assembler wird gezeigt. Darüberhinaus werden Unterprogrammtechniken und die Erstellung positionsunabhängigen Codes besprochen. Ziel ist es, ein Basisverständnis für die Programmierung in Assemblers und die Umsetzung einer höheren Programmiersprache auf die Maschinenebene zu vermitteln.

9. Ein- und Ausgabe

Inhalt: Die Anbindung peripherer Geräte an eine CPU wird in diesem Kapitel behandelt. Schwergewicht wird auf die Hardwarestruktur einer einfachen parallelen Schnittstelle gelegt. Die Steuerung des Datenstroms durch entsprechende Signale, Kontrollregister und Datenkanäle wird auf der Register-Transferebene gezeigt. Die Konfigurierung der parallelen Schnittstelle und die Realisierung des Datentransfers auf Assemblerebene wird besprochen. Darüberhinaus wird ein einfaches Protokoll zur Kontrolle des Datenflusses eingeführt.

10. Unterbrechungsbearbeitung

Inhalt: Die Hard-und Softwarestrukturen zur Behandlung asynchroner (systembezogener) und synchroner (programmbezogener) Unterbrechungen werden eingeführt. Verschiedene Techniken zur Bestimmung der Unterbrechungsquelle sowie der Einhaltung festgelegter Vorrangrelationen (Prioritäten) werden besprochen.

11. Ein 32-Bit Microprozessor

Inhalt: Am Beispiel des MC68020 werden Eigenschaften von 32-Bit Prozessoren besonders in Hinblick auf die Unterstützung von Übersetzern und Betriebssystemen dargelegt. Es wird ein kurzer Überblick über den Befehlssatz und die komplexen Adressierungsarten gegeben. Am Beispiel der 680x0 Familie wird ein asynchrones Bussystem besprochen. Dabei werden die Kontrolltechniken wie die des Handshakes und der Arbitrierung eines Busses gezeigt.

12. Erweiterung des Befehlssatzes durch Coprozessoren

Inhalt: Die Erweiterung des Befehlssatzes eines Prozessors durch Spezialfunktionen wird diskutiert. Verschiedene Alternativen werden vorgestellt. Die Coprozessortechnik wird eingeführt. Coprozessoralternativen werden nach verschiedenen Gesichtspunkten klassifiziert. Am Beispiel der Coprozessor-Schnittstelle des 68020 wird eine Realisierung besprochen.

13. RISC-Prozessoren

Inhalt: Die wesentlichen Gesichtspunkte der RISC-Prozessoren werden dargelegt. Es werden die grundlegenden Unterschiede zur CISC-Technik erarbeitet. Danach wird einer der Schlüssel zur RISC-Technik, das Pipelining, im Detail behandelt. Die grundlegenden Gesichtspunkte der Leistungsbewertung von Prozessoren werden ebenfalls dargestellt. Ziel ist es, die Problematik der Leistungsbewertung zu verdeutlichen.

Technische Informatik

Rechnerorganisation

Inhalt:

1 Einführung

Der Aufbau und die Arbeitsweise der Zentraleinheit eines Rechners, der CPU (Central Processing Unit) ist Gegenstand dieses Skripts. Die CPU umfaßt alle Komponenten, die gebraucht werden, um die Befehle eines Programms in geordneter Weise zu lesen, sie zu entschlüsseln und auszuführen. Die Begriffe CPU und Prozessor werden hier synonym verwandt. Die CPU ist der aktive Bestandteil eines Computersystems, da sie den kontrollierten Fortschritt jeder Berechnung steuert.

Die Fortschritte, die digitale Prozessoren seit den ersten funktionierenden Prototypen gemacht haben, sind enorm. Nach [PaH94] beträgt das Preis/Leistungsverhältnis zwischen dem ersten kommerziellen Rechner (UNIVAC 1) und einer modernen Workstation (HP9000/model 750) **1:16122356**, die Verbesserung liegt also bei etwa 7 Größenordnungen. Zum großen Teil trug die technische Entwicklung zu diesem Fortschritt bei. In Tab. 1 sind die Computergenerationen und die jeweilige Technologie aufgelistet, wobei der Sprung von einer zur nächsten Generation durch den Wechsel der Technologie begründet ist. Die volumenmäßige Verkleinerung der Schaltelement von der Elektronenröhre zum integrierten Transistor beträgt mehr als 12 Größenordnungen bei vorsichtiger Schätzung.

Gen.	Zeitraum	Technologie	Produkt, Anwendungsbereich
1	1950-1959	Elektronenröhren	Kommerzielle elektr. Comp. z.B. UNIVAC
2	1960-1968	Transistoren	Billigere und zuverlässigere Comp., z.B. IBM 360/50
3	1969-1977	Integrierte Schaltkreise	Minicomputer z.B. PDP-8
	1971	erster Mikroprozessor Intel 4004 (2300 Transistoren)	
4	1978-199?	LSI, VLSI[1]	Personal Comp. Workstations
5	199?-20??	VHSIC[2], ??	Integrated Personal Assistent, computergestützte Systeme

Tab. 1 Computergenerationen

[1] LSI: Large Scale Integration, VLSI: Very Large Scale Integration
[2] VHSIC: Very High Speed Integrated Circuits

Aber es ist nicht nur der technologische Fortschritt, sondern gerade die Wechselwirkung zwischen technologischen Möglichkeiten und der kreativen Nutzung dieser Möglichkeiten, die das Gebiet der Architektur und Organisation von Prozessoren vorangetrieben hat.

Rechnerarchitektur ist deshalb eine "Kunst". Allerdings eine, die im Gegensatz zur bildenden Kunst nach relativ objektiven Kriterien gemessen und bewertet werden kann. Deshalb bilden zwei Kapitel den Rahmen um die eigentliche Diskussion der Prozessorstrukturen: ein einführendes Kapitel über die physikalischen Grundlagen der Hochintegration, das mit einer kurzen Diskussion über die Auswirkungen der Miniaturisierung abschließt, und ein Kapitel über die Leistungsmessung bei Prozessoren, die eine Bewertung der jeweiligen Strukturen erlaubt.

Was ist Rechnerarchitektur und was ist Rechnerorganisation ?

Die gröbste Unterteilung für ein Computersystem ist wohl die in Hardware und Software. Die Software sind Programme, die in einer beliebigen höheren Programmiersprache geschrieben sein können. Sie werden durch den Compiler in elementare Befehle übersetzt, die von der Hardware ausgeführt werden können. Der Vorrat an elementaren Befehlen der Hardware, der sogenannten Maschinenbefehle, bildet die Schnittstelle zwischen der Software und der Hardware. Die Festlegung dieser Schnittstelle und die Realisierung der Maschinenbefehle durch digitale Komponenten ist der Gegenstand der Rechnerarchitektur und der Rechnerorganisation.

Was ist aber nun der Unterschied zwischen Rechnerarchitektur und Rechnerorganisation? Dies ist eine bereits lange andauernde und vielleicht etwas überflüssige akademische Diskussion, über die sich jedoch trefflich streiten läßt. Historisch kam die Diskussion mit der Einführung von Rechnerfamilien durch IBM auf, d.h. Rechnern, die bezüglich eines kompilierten Programms kompatibel sind, aber möglicherweise sehr unterschiedlichen Leistungsklassen angehören[3]. Bisher hatten die Rechner einen gewissen Befehlsvorrat, auf den ein kompiliertes Programm zurückgreifen konnte. Dieser Befehlsvorrat wurde von der Hardware in möglichst optimaler Weise realisiert. Es bestand keine Notwendigkeit, zwischen dem Entwurf der Hardware-/Softwareschnittstelle und ihrer Umsetzung in digitale Logik zu unterscheiden. Unter Rechnerarchitektur konnte man alles fassen, den Entwurf eines Befehlsvorrates und seine effiziente Umsetzung in eine Hardwarearchitektur. Die änderte sich mit der Einführung von Rechnerfamilien, in denen alle Familienmitglieder dieselbe Hardware-/Softwareschnittstelle

[3] Prominente Beispiele sind die Intel x86 Prozessorfamilie, Motorola 680x0 Familie und die verschiedenen Rechnerfamilien von DEC (z.B. PDP 11, VAX) und IBM (angefangen hat es mit der 360/xxx Architektur), wobei manche Familien, z.B. die erwähnten Intel und Motorola Familien sogenannte "Aufwärtskompatibilität" aufweisen, d.h. ein Objektprogramm der älteren Prozessorgeneration läuft auf einer jüngeren, aber nicht umgekehrt.

besitzen, aber die Umsetzung in digitale Logik völlig unterschiedlich ist. Es bildete sich die begriffliche Unterscheidung heraus zwischen *Rechnerarchitektur*, die den Entwurf des Vorrats von Maschinenbefehlen und damit der Festlegung der Hardware-/Softwareschnittstelle definiert, und *Rechnerorganisation*, die die Realisierung dieser Schnittstelle durch entsprechende Komponenten behandelt. Die Rechnerarchitektur beschreibt also eine gewisse Abstraktion eines Rechners, ohne sich um die genaue hardwaremäßige Realisierung zu kümmern. Die Aufgabe des Rechnerarchitekten ist damit die Festlegung eines Maschinenbefehlssatzes, der möglichst gut auf die Anforderungen zugeschnitten ist, die von höheren Programmiersprachen, Compilern, Betriebssystemen und Anwendersoftware gestellt werden. Die Aufgabe der Rechnerorganisation ist es, Strukturen zu finden, die bei vorgegebenen Randbedingungen hinsichtlich der Kosten, Geschwindigkeit, Technologie usw. die Hardware-/Softwareschnittstelle möglichst optimal realisieren.

Natürlich lassen sich die Gebiete der Rechnerarchitektur und der Rechnerorganisation nicht beliebig trennen (daher ist auch die Diskussion und Abgrenzung oft überflüssig). Es gibt eine Vielzahl von Wechselbeziehungen und gegenseitigen Beeinflussungen zwischen ihnen, insbesondere dann, wenn man, unabhängig von den Einschränkungen irgendwelcher Kompatibilitätsvorgaben, nach neuen, leistungsfähigen Rechnerstrukturen sucht. Daß die Begriffe fließend benutzt werden, zeigt z.B. die Tatsache, daß die meisten Beiträge der Internationalen Konferenz für Rechnerarchitektur nach obiger Interpretation Themen der Rechnerorganisation behandeln, oder auch, daß ein neues (1994) Buch über das Thema von David Patterson und John Hennesy mit: "Computer Organization and Design - The Hardware/Software Interface" [PaH94] betitelt ist.

Abstraktionsebenen in einem Computersystem

Um komplexe Systeme wie z.B. eine CPU zu verstehen, ist es sinnvoll, verschiedenen Abstraktionsebenen zu etablieren. Eine der Abstraktionsebenen, die wir bereits in dem Bereich "Digitale Logik" genutzt haben, war die Betrachtung der logischen Funktion eines Schaltkreises, ohne daß wir die physikalischen oder elektrotechnischen Funktionsweisen berücksichtigt haben. Wir sind einfach davon ausgegangen, daß die Ingenieure uns einen Schaltkreis liefern, der, in gewissen Grenzen, eine spezifizierte logische Funktion realisiert, so daß wir die Funktionsweise eines Addierers nicht auf der Ebene einzelner Transistoren, Widerstände und Kondensatoren oder gar auf der Ebene von atomaren Gitterstrukturen und Elektronenwanderung verstehen müssen, sondern auf der Ebene von logischen Gattern nachvollziehen können.

Daniel P. Siewiorek, C. Gordon Bell und Allen Newell haben in einem Standardwerk der Rechnerarchitektur: "Computer Structures: Principles and Examples" [SBW82] eine solche hierarchische Schichtung eines Computersystems vorgeschlagen. Ein leicht modifiziertes Schichtenmodell ist in Abb. 1 angegeben. Es zeigt die Ebenen bis zur Hardware-/Softwareschnittstelle. Jede Ebene im Schichtenmodell beschreibt eine Abstraktion, die eine Schnittstelle zur nächsthöheren Ebene bildet. Dabei werden unnötige Details für die höhere Ebene herausfiltert und unterdrückt.

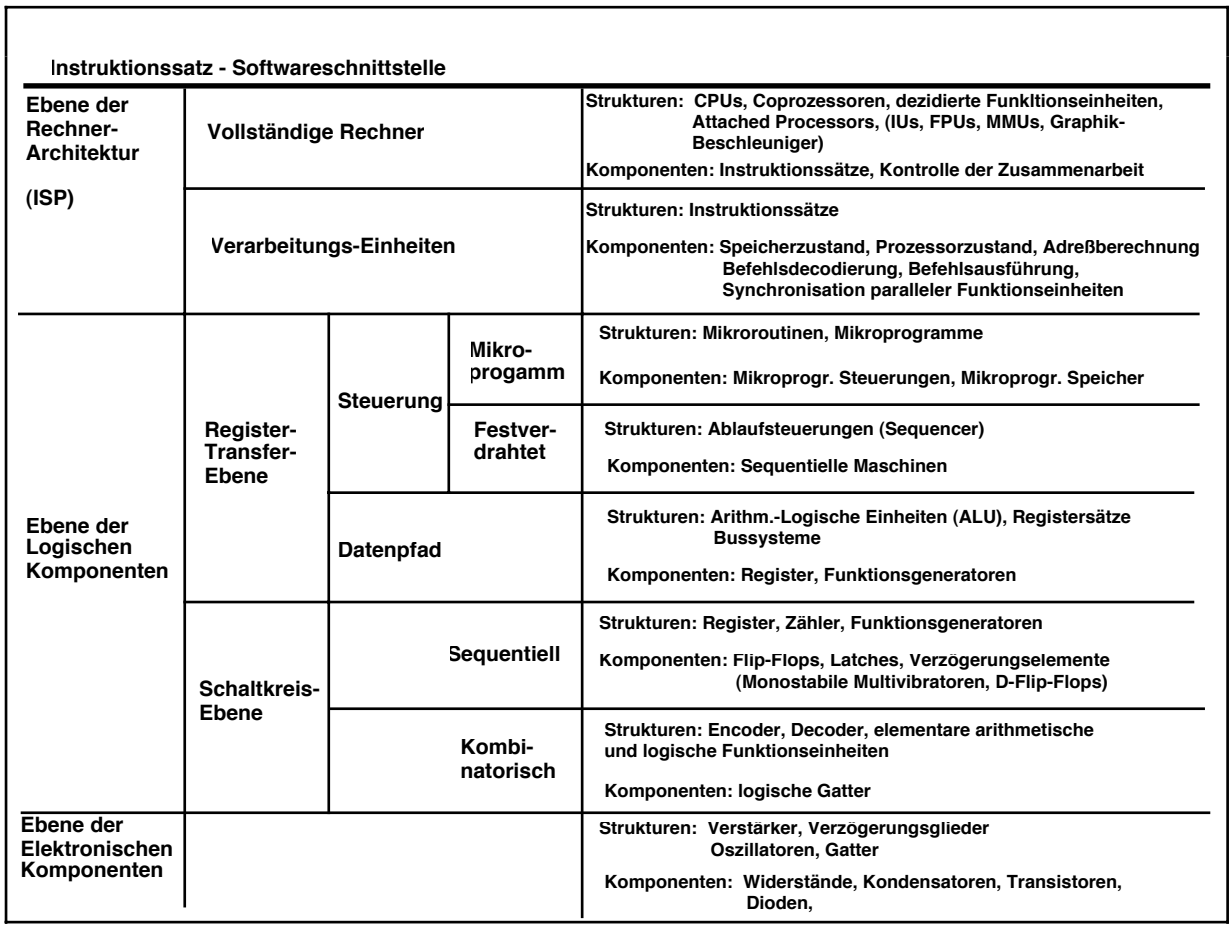


Abb. 1 Ebenen des Prozessorentwurfs

In jeder Ebene sind die Komponenten angegeben, die zur Realisierung der abstrakten Strukturen an der Schnittstelle nach oben notwendig sind. Diese Strukturen dienen in der höheren Ebene ihrerseits wieder als Komponenten, um komplexere Strukturen aufzubauen. Wenn wir also sicher sein können, daß sich ein logisches Gatter so verhält, wie in seiner Spezifikation angegeben ist, können wir damit arbeiten, ohne die darunterliegenden Ebenen unbedingt verstehen zu müssen.

Abb. 1 stellt auch einen Fahrplan für dieses Script dar, insofern wir uns von unteren Abstraktionsebenen nach oben arbeiten werden. Dabei können wir die Schaltkreisebene überspringen, da sie im Teil "Digitale Logik" bereits abgehandelt ist. Die dort eingeführten Strukturen wie z.B. Kodierer, Dekodierer, Addierer, Register und Zähler können wir in der nächsthöheren Ebene, der Register-Transferebene, benutzen, um Einheiten zur Verarbeitung von Daten und zur Kontrolle des Rechenfortschritts zu konstruieren und zu einer CPU zu verbinden.

Auf der Ebene der Rechnerarchitektur (ISP= Instruction Set Processor) werden wir Befehlssätze entwerfen, diskutieren und schrittweise Verbesserungen für eine komfortable Programmierung einführen.

Literatur:

[PaH94] David A. Patterson, John L. Hennessy
Computer Organization & Design - The Hardware/Software Interface
Morgan Kaufmann Publishers, San Mateo, CA, 1994

[Cle91] Alan Clements
The Principles of Computer Hardware
Second Edition, Oxford Science Publications, Oxford University Press,1991*

[Key87] R.W. Keyes
The Physics of VLSI Systems
Addison Wesley Company, NY, Reading (Mass.), 1987

[Lev81] Lance A. Leventhal
6809 Assembly Language Programming
OSBORNE/McGraw-Hill, Berkeley, CA, 1981

[Bar91] Thomas C. Bartee
Computer Architecture and Logic Design
McGraw-Hill, Inc., 1991

[SBN82] Daniel P. Siewiorek, C. Gordon Bell, Allen Newell
Computer Structures: Principles and Examples
McGraw-Hill, 1982

[Fos70] Caxton C. Foster
Computer Architecture
Computer Science Series, v. Nostrand Reinhold Company, 1970

[GrV94] K.E. Großpietsch, H. Th. Vierhaus
Entwurf hochintegrierter Schaltungen
BI, Reihe Informatik, Bd. 96, 1994

[Püt92] Jean Pütz (Hrsg.)
Einführung in die Elektronik
Fischer Taschenbuch, 1992

* Inzwischen ist eine neue Auflage (2000) erschienen.

2 Grundlagen elektronischer Schaltkreise und der Hochintegration

Zur Realisierung digitaler Logik werden steuerbare Schalter benötigt. Ein Schalter hat zwei unterscheidbare Zustände: eingeschaltet und ausgeschaltet. Wenn er eingeschaltet ist, hat er einen geringen ohmschen Widerstand, demzufolge kann ein elektrischer Strom fließen. Ist er ausgeschaltet, hat er einen hohen ohmschen Widerstand, so daß kein (oder nur ein sehr geringer) Strom fließt. Ein elektromechanisches Relais ist die anschaulichste Realisierung eines steuerbaren Schalters. Abb. 2.1 zeigt die zwei Zustände, die ein Relais einnehmen kann.

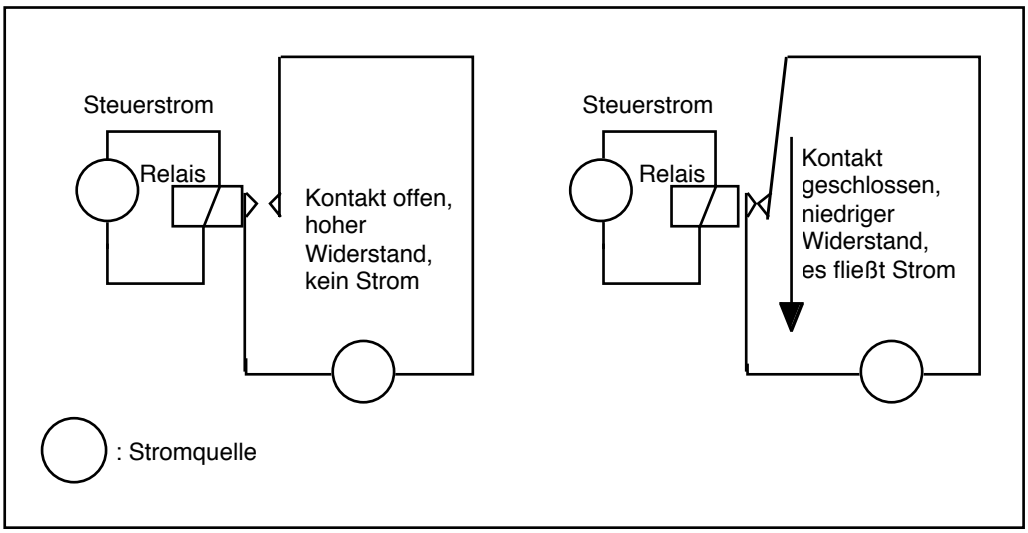


Abb. 2.1 Zustände eines Relais

Der hohe Widerstand bei offenem Kontakt wird erreicht durch eine gute Isolation der beiden Kontakte. Die Widerstandsänderung zwischen offenem und geschlossenem Kontakt beträgt viele Größenordnungen. In dieser Hinsicht stellen Relais sehr gute Schalter dar. Tatsächlich wurden Rechner mit Relais gebaut (z.B. die ersten Rechner, die von dem deutschen Computerpionier Konrad Zuse konstruiert wurden), allerdings verloren Relais durch Größe, Stromverbrauch und vor allem durch unzureichende Geschwindigkeit früh ihre Relevanz für Rechner. Die ersten elektronischen Schalter, die nicht mehr auf mechanischen Kontakten, sondern auf der Steuerung eines Elektronenstroms beruhten, wurden mit Elektronenröhren aufgebaut. Nicht so sehr die Geschwindigkeit, sondern Stromverbrauch und Größe setzten hier enge Grenzen für die Verwendbarkeit in einem Rechner[0]. Der Durchbruch in Hinblick auf

[0]Diese Tatsache wird durch die folgende technische “Vision" aus *Popular Mechanics, März 1949, S. 258* beleuchtet: Where the ENIAC is equipped with 18.000 vacuum tubes and weights 30 tons, computers in the future may have 1.000 vacuum tubes and weight just 1-1/2 tons".
Legt man den Stromverbrauch einer (vormals) gängigen Elektronenröhre (der E-Reihe) mit ca. 0,3 Ampere bei 6,3 Volt zugrunde, ergibt sich alleine durch die erforderliche Heizung der Röhren des Eniac ein Leistungsverbrauch von ca. 34 kW. Im Vergleich dazu verbraucht einer der schnellsten RISC-Prozessoren, der DEC Alpha mit einigen Millionen Schaltelementen ca. 25 Watt.

Stromverbrauch und Miniaturisierung kam mit der Entdeckung des Transistors durch John Bardeen und Walter H. Brattain in den Bell Labs. Bardeen und Brattain entdeckten durch Zufall, daß der Widerstand eines Halbleiters durch einen Steuerstrom (wie beim Relais) veränderbar ist. Wiliam Shockley war es, der diesen Vorgang physikalisch erklärte und mit Bardeen und Brattain dafür 1956 den Nobelpreis für Physik erhielt.

Abb. 2.2 zeigt den spezifischen Widerstand[1] verschiedener Materialien. Man sieht, daß sich der Widerstand von Halbleitermaterialien über mehrere Größenordnung verändern läßt. Diese Eigenschaft kann man zur Herstellung von Schaltern ausnutzen.

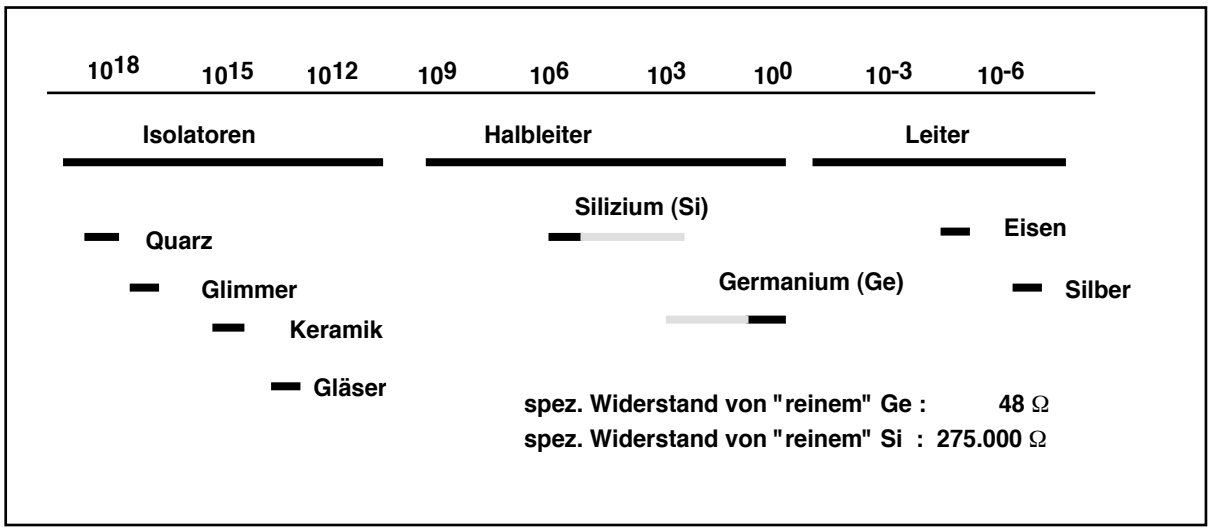


Abb. 2.2 Spezifische Widerstände verschiedener Materialien

Im folgenden sollen die physikalischen Ursachen, die diesem Phänomen zugrunde liegen, näher bertrachtet werden. Es muß vorausgeschickt werden, daß sich die Betrachtung auf sehr stark vereinfachte physikalische Modelle stützt und die Phänomene, die der elektrischen Leitung und den Halbleitern zugrunde liegen, nur recht grob beschreibt. Die gewählte Abstraktionsebene bietet allerdings einen anschaulichen Rahmen zum informalen Verständnis. Eine vertiefende Beschreibung findet man in [Key87]. Eine sehr allgemeinverständliche Darstellung wird in [Püt92] gegeben. In [GrV94] werden die Phänomene ebenfalls insbesondere in Hinblick auf Hochintegration beschrieben.

---

[1] Der ohmsche Widerstand in einem einheitlichen Körper ist:
- proportional zur Länge l des Körpes
- umgekehrt proportional zum Querschnitt q des Körpers
- abhängig vom Material
- abhängig von der Temperatur T

Der spezifische Widerstand wird definiert als der Widerstand eines bestimmten Materials wenn: l= 1cm, q= $1cm^2$ und T= 0^o C gegeben ist.

2.1 Das Phänomen der Leitfähigkeit in Halbleitern

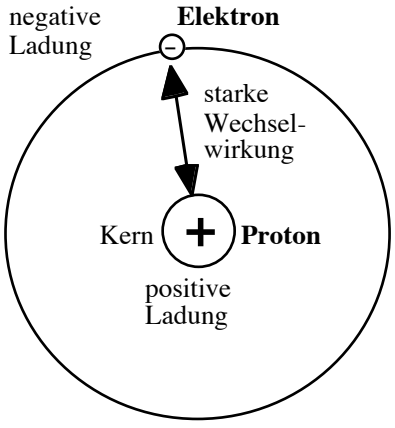
Prüfer: *Was ist Elektrizität ?*
Prüfling: *Oh, Herr Professor, ich bin sicher, ich habe es gelernt - ich bin sicher , ich habe es gewußt - aber ich hab's vergessen.*
Prüfer: *Wie sehr bedauerlich: Nur zwei Personen haben je gewußt, was Elektrizität ist, der Urheber der Natur und Sie. Jetzt hat es einer von Ihnen vergessen.*

Prüfungsprotokoll der Universität Oxford, um 1890

Es gibt verschiedene Prinzipien der elektrischen Leitfähigkeit, etwa die Ionenleitung in Flüssigkeiten, die Leitung in Metallen oder die Leitung in Halbleitermaterialien. Letztere wird in diesem Kapitel näher beschrieben, da sie die Grundlage elektronischer Schaltelemente ist.

Ohne nähere Erläuterung führen wir folgende Begriffe ein: Ein elektrisches Feld ist eine Kraft, die auf ein elektrisch geladenes Teilchen wirkt. Ein elektrisch geladenes Teilchen wird als Ladungsträger bezeichnet. Die elektrische Ladung kann positiv oder negativ sein. Zwischen zwei gleichnamigen Ladungen (++ oder --) wirkt eine abstoßende Kraft, zwischen ungleichnamigen Ladungen (+- oder -+) eine anziehende Kraft.

Ein Atom besteht aus einem Atomkern, der aus positiv geladenen Protonen und elektrisch neutralen Neutronen besteht, und aus um den Kern kreisenden, negativ geladenen Elektronen. Dabei ist die Anzahl der positiv geladenen Protonen und der negativ geladenen Elektronen gleich, so daß sich die Ladungen gegenseitig kompensieren. Ein einzelnes Atom ist nach außen elektrisch neutral. Abb. 2.3 zeigt das Modell eines Atoms mit einem Proton und einem Elektron.



- Die negative Ladung des Elektrons und die positive Ladung des Kerns kompensieren sich gegenseitig, d.h. nach außen ist das Atom elektrisch neutral
- je weiter das Elektron vom Kern entfernt ist, desto schwächer ist seine Bindung an den Kern
- Da sich die Bahnen eines oder mehrerer Elektronen räumlich auf einer Kugel um den Atomkern befinden, spricht man von Elektronenschalen. Elektronenschalen kennzeichnen bestimmte Energieniveaus, mit der Elektronen an den Kern gebunden sind.

Abb. 2.3 Atom mit einem Proton und einem Elektron

Die elektrische Leitfähigkeit beruht auf beweglichen Ladungsträger. Wir betrachten die Leitung durch freie Elektronen. Isolatoren haben keine frei beweglichen Ladungsträger - deshalb isolieren sie. Kupfer, als Repräsentant der gut leitenden Materialien, hat sehr viele freie Ladungsträger.

Die Frage ist daher: Wo kommen die freien Ladungsträger in Halbleitermaterialien her und woraus resultiert der große Variationsbereich bei der Leitfähigkeit? Dazu müssen wir die Struktur der Bindung mehrerer Atome eines Halbleitermaterials betrachten. Ein solcher Materialkörper besteht aus sehr vielen Atomen, die im wesentlichen durch elektrische Kräfte zusammengehalten werden.

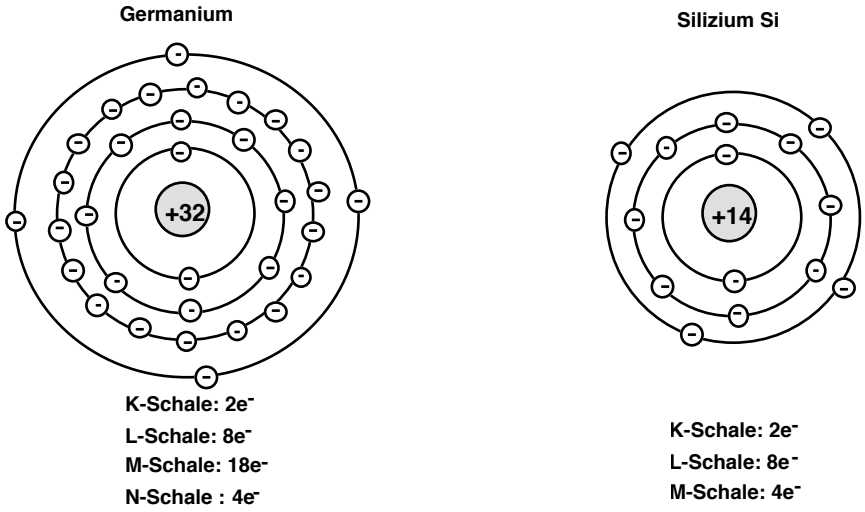


Abb. 2.4 Die Struktur eines Germanium und eines Siliziumatoms

Die Elektronen der äußeren Schale eines Atoms sind dabei für die Bindung an andere Atome entscheidend. Die Anzahl der Elektronen auf der äußeren Schale bestimmt die sogenannte **Valenz** (Wertigkeit) eines Atoms. Die äußere Schale ist mit 8 Elektronen vollständig besetzt. Als Beispiel werden in Abb. 2.4 das Germanium und das Silizium Atom gezeigt. Die Wertigkeit beider Atome ist 4, da sich 4 Elektronen auf der äußeren Schale befinden.

Betrachtet man die Struktur der Bindung mehrerer Germaniumatome, erhält man ein sogenanntes Diamantgitter, das in Abb. 2.5 skizziert ist. Im Bestreben, seine äußere Schale mit 8 Elektronen aufzufüllen[2], bildet ein Germanium Atom einen Verbund mit jeweils vier Nachbaratomen. Dabei wird von jedem der vier Nachbaratome je ein Elektron zur Ergänzung der eigenen äußeren Schale übernommen. Gleichzeitig dient jedes der vier eigenen Valenzelektronen je einem Nachbarn als Ergänzung seiner äußeren Schale. Durch diese ‘Verzahnung‘ wird eine stabile Gitterstruktur gebildet, in der alle Elektronen fest eingebunden

[2] Jedes chemische Element ist bestrebt, einen besonders stabilen Zustand einzunehmen, der erreicht ist, wenn die äußere Schale mit 8 Elektronen aufgefüllt ist. Dies ist z.B. bei Edelgasen der Fall, die deshalb nur sehr schwer eine chemische Verbindung eingehen.

sind. In der räumlichen Struktur des Diamantgitters sind die vier Atome eines einzelnen Nachbarschaftsverbundes in der Form eines Tetraeders angeordnet (Abb. 2.6)

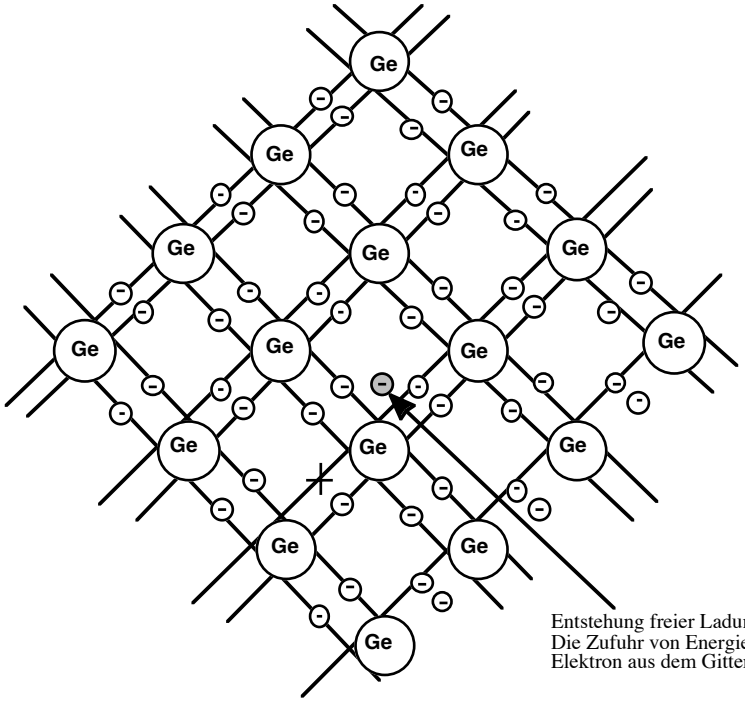


Abb. 2.5 Diamantgitterstruktur des Germaniums

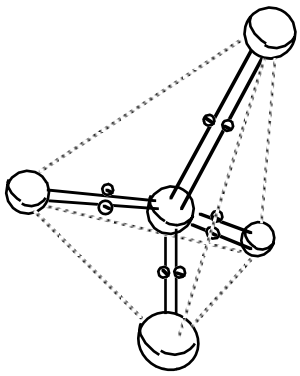


Abb. 2.6 Räumliche Anordnung der Atome

Wo kommen nun die freien Elektronen her, die zur Leitfähigkeit notwendig sind ?

Tatsächlich haben Halbleiter bei tiefen Temperaturen eine sehr schlechte Leitfähigkeit. Bei 0° K (0° Kelvin ist der absolute Nullpunkt) kann sich kein Elektron aus seinem Gitterverbund lösen. Der Widerstand ist unendlich groß. Führt man aber Energie in Form von Wärme zu, können einzelne Elektronen aus dem Gitter herausgelöst und zu freien Ladungsträgern werden. Abb. 2.5. zeigt ein solches freies Elektron. Die Erklärung, warum es zu einem freien Elektron kommen kann, wird durch das sogenannte Bänder-Modell geliefert, das die Beziehung zwischen dem Aufenthaltsort eines Elektrons auf einer bestimmten Schale und seiner Energie

herstellt. Die Energie eines Elektrons im Atom kann nur bestimmte diskrete Werte annehmen. Jede Schale des Atoms definiert eine bestimmte Energie der sich darauf befindenden Elektronen. Zwischenwerte sind nicht erlaubt, ebenso wie sich ein Elektron nicht im beliebigen Abstand vom Kern aufhalten kann. Abb. 2.7.a zeigt die Energieniveaus, die für die Elektronen eines Einzelatoms möglich sind, wobei die Energie eines Elektrons mit dem Abstand zum Kern zunimmt.

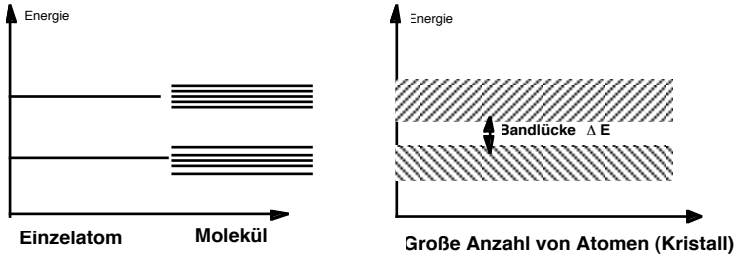


Abb. 2.7. Energieniveaus für Elektronen

Wegen eines "Ausschließungsprinzips" der Quantentheorie, des sogenannten Pauli-Prinzips, ist es nicht erlaubt, daß zwei Elektronen genau denselben Energiezustand einnehmen. Entsprechende Zustände von Einzelatomen spalten sich deshalb in eine Menge energetisch eng benachbarter Zustände auf, so daß die scharfen Energiezustände einzelner Elektronen zu sogenannten ***Energiebändern*** "verschmieren". Abb. 2.7.b zeigt die Energie von Elektronen für die begrenzte Zahl Atome eines Einzelmoleküls und Abb. 2.7.c zeigt die Energiebänder in einem Verband aus einer sehr großen Anzahl von Atomen. Zwischen den Energiebändern besteht eine ***Bandlücke*** ΔE, die die Energie angibt, die einem Elektron zugeführt werden muß, um in ein höheres Energieband gehoben zu werden. Man bezeichnet die Energiezustände auf der äußeren Schale des Atoms, die die Valenz definiert, auch als ***Valenzband***. Freie, nicht an ein Atom gebundene Elektronen befinden sich in einem energetisch höheren Zustand, im sogenannten ***Leitungsband***. Bei gut leitenden Materialien, wie z.B. Kupfer, befinden sich ständig eine große Zahl von Elektronen im Leitungsband und sind daher frei bewegliche Ladungsträger.

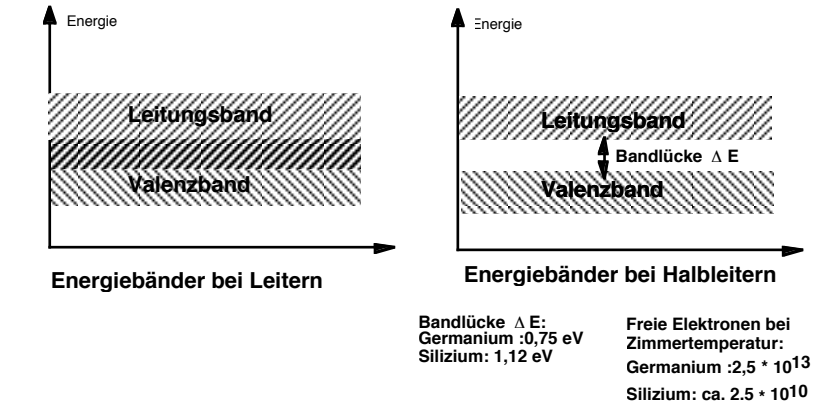


Abb. 2.8 Energiebänder und Bandlücke

Abb. 2.8 a.) zeigt, daß das Valenzband und das Leitungsband nicht durch eine Bandlücke getrennt, sondern zu einem breiten Band “verschmiert“ sind.

Bei Halbleitern existiert eine Energielücke. Befindet sich ein Elektron im Valenzband, kann es erst durch Energiezufuhr die Energielücke überwinden und in das Leitungsband gehoben werden. In Abb. 2.8 b.) sind die Energielücken für Germanium und Silizium angegeben. Diese Energie muß von außen zugeführt werden, um Elektronen aus der Gitterstruktur herauszulösen und in einen freien Zustand zu heben. Bereits bei Zimmertemperatur befinden sich eine großen Anzahl von Elektronen im Leitungsband (Abb. 2.8 b).Der Halbleiter besitzt dann nur noch einen endlichen ohmschen Widerstand - er leitet[3].

Wie wollen nun genauer betrachten, was im Diamantgitter eines Halbleiters passiert, wenn freie Elektronen vorhanden sind. Abb. 2.9 zeigt ein Atom mit seinen vier Nachbarn im Diamantgitter. Wird ein Elektron in das Leitungsband gehoben, entsteht an dieser Stelle ein Loch, auch Defektelektron genannt. Die lokale Struktur ist nun nach außen nicht mehr elektrisch neutral, sondern positiv. Da mit jedem herausgelösten Elektron ein Loch entsteht, spricht man von Elektron/Loch Paarbildung. Sind in einem Materialkörper die Elektronen und Löcher ungleich verteilt, wandern die Elektronen und es findet eine Rekombination statt, die wieder einen stabilen Zustand herstellt.

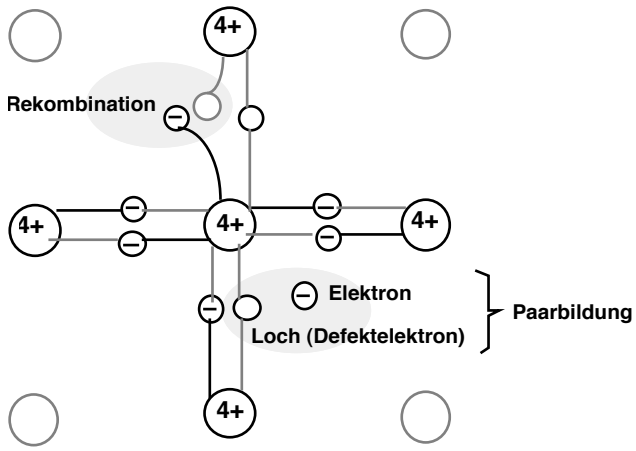


Abb. 2.9 Paarbildung und Rekombination

Betrachten wir nun die Vorgänge im Gitter, unter dem Einfluß einer elektrischen Spannung. Die Wanderung eines Einzelelektrons ist in Abb. 2.10 gezeigt.

---

[3] Die Temperaturabhängigkeit von Halbleitermaterialien wird zur Temperaturmessung ausgenutzt. Die negative Seite der Temperaturabhängigkeit ist, daß Halbleiter nur in einem begrenzten Temperaturbereich eingesetzt werden können. Für kommerzielle digitale Schaltkreise ist das der Bereich 0^{o} C bis 70^{o} C (militärisch -40^{o}C bis 155^{o}C). Außerhalb dieses Temperaturbereichs können die spezifizierten Eigenschaften nicht garantiert werden.

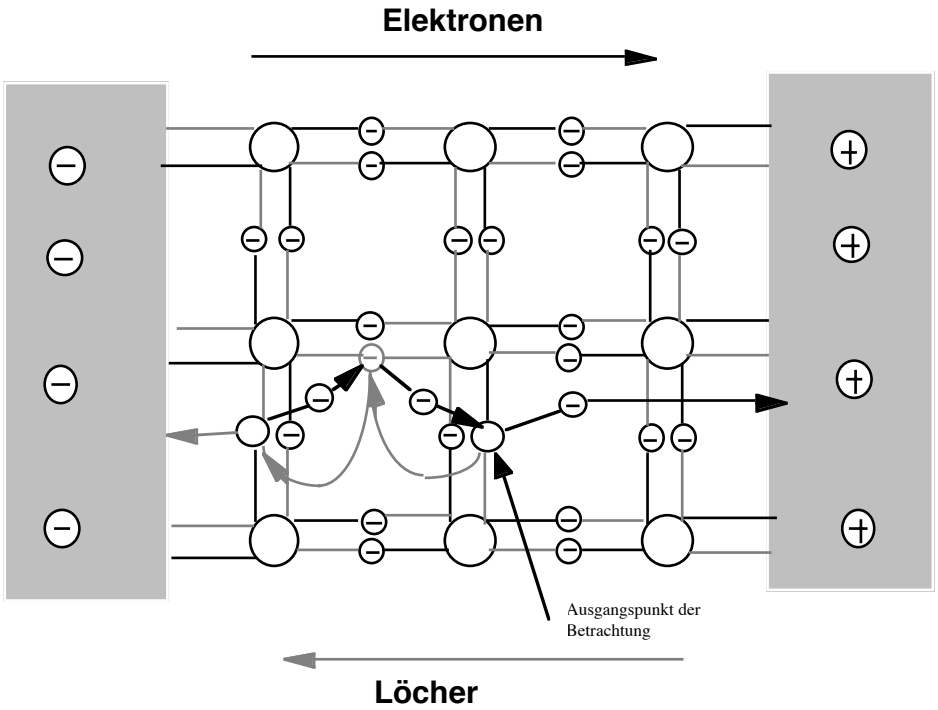


Abb. 2.10 Halbleiter unter Einfluß einer elektrischen Spannung

Auf der rechten Seite der Abb. 2.10 liege eine positive Spannung an, d.h. es herrscht ein Mangel an elektrisch negativ geladenen Ladungsträgern, den Elektronen. Auf der linken Seite sei ein Überschuß an Elektronen, so daß diese Seite negativ geladen ist. Nehmen wir an, an der mit dem Pfeil in Abb. 2.10 bezeichneten Stelle wird ein Elektron herausgelöst. Es wandert in die Richtung, in der ein Mangel an Elektronen herrscht. Durch die Paarbildung hinterläßt das Elektron ein Loch. Da an der linken Seite ein Elektronenüberschuß vorhanden ist und durch die positive Ladung, die das Defektelektron trägt, kommt es zur Wanderung von Elektronen von links nach rechts und zur Rekombination. An der Stelle des herausgelösten Elektrons entsteht wieder ein Loch, das durch weitere Rekombination seinerseits nun immer weiter nach links wandert. Wir können also eine Wanderung der Elektronen von links nach rechts und bedingt dadurch eine Wanderung von Löchern von rechts nach links beobachten. Wir sagen: es fließt ein Strom, bedingt durch die Wanderung der Ladungsträger. Da für die Leitung zwei Arten von Ladungsträgern, nämlich Elektronen und Löcher, eine Rolle spielen, spricht man von *bipolarem* Stromverhalten.

Um ein optimales Basismaterial für die Herstellung elektronischer Schaltkreise zu erhalten, müssen technisch zwei Hauptvoraussetzungen erfüllt werden:

1. Es müssen Halbleiterkristalle mit größtmöglicher Regularität und Reinheit der Gitterstruktur “gezüchtet“ werden.
2. Es müssen gezielt freie Ladungsträger geschaffen werden, da die in einem reinen Kristall vorhandenen Ladungsträger nicht ausreichen.

Der Stand der Technik ist, daß man "Einkristalle" in Form runder Stäbe mit ca. 200 cm Länge und einem Durchmesser von ca. 20 cm züchten kann (Mitsubishi 1995). Diese Stäbe werden dann in dünne Scheiben geschnitten, sogenannte Wafer, auf die man dann die Struktur der Schaltkreise aufbringt[4].

Die künstliche Schaffung freier Ladungsträger erzielt man mit einer kontrollierten Verunreinigung des Kristalls durch Fremdatome. Diesen Vorgang nennt man Dotieren. In Abb. 2.12 ist das Wesentliche der Dotierung zusammengefaßt. Will man die Anzahl der positiven Ladungsträger (Defektelektronen, Löcher) künstlich erhöhen, fügt man Atome, die für die Diamantgitterbindung ein Elektron zu wenig auf ihrer äußeren Schale besitzen, d.h. eine Valenz von 3 haben, hinzu. Ist ein Überschuß an negativen Ladungsträgern (Elektronen) das Ziel, wird das Diamantgitter mit Atomen verunreinigt, die ein Elektron mehr besitzen, als zur Kristallbindung notwendig ist.

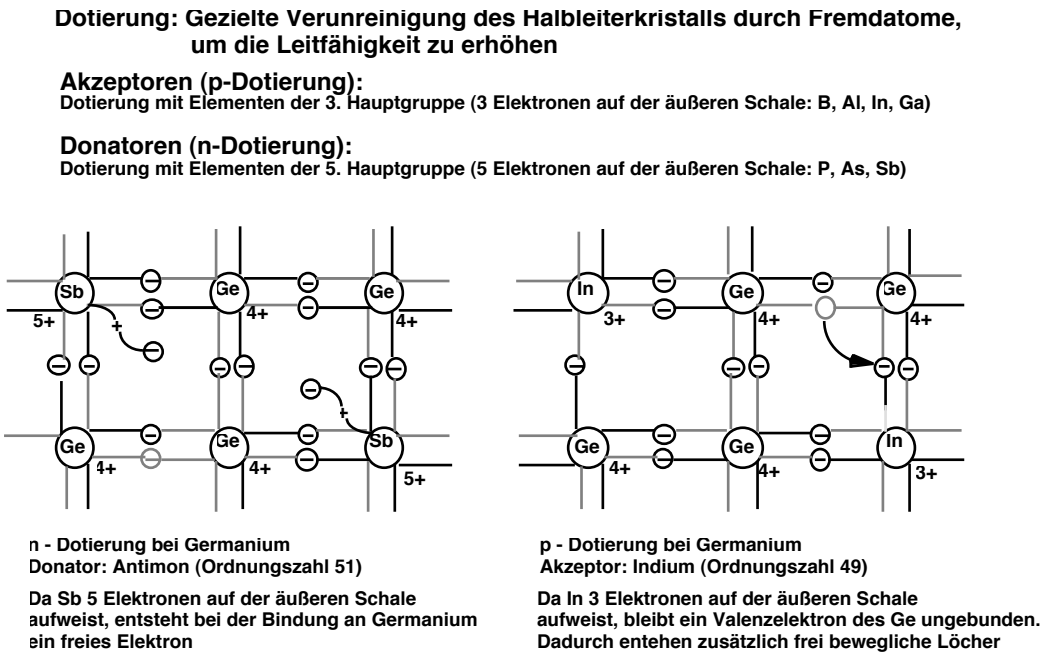


Abb. 2.11 Dotierung von Halbleitern

Die entsprechenden Elemente werden für die positive Dotierung (p-Dotierung) *Akzeptoren* genannt und entstammen der 3. Hauptgruppe des Periodensystems. Die Elemente für die n-Dotierung werden als *Donatoren* bezeichnet und sind in der 5. Hauptgruppe zu finden. Bei Germanium wird die n-Dotierung durch Antimon (Sb 5, Sb: Stibium) erzielt, die p-Dotierung

[4] Auf einem Wafer werden mehrere individuelle Schaltkreise aufgebracht. Nach der Fertigung wird der Wafer in rechteckige Plättchen zersägt, die diese Schaltkreise enthalten. Diese werden in einem entsprechenden Gehäuse eingekapselt.

durch Indium (In 3). Die durch die Dotierung eingepflanzten Ladungsträger nennt man ***Majoritätsträger***. Die Ladungsträger, welche die früher diskutierte natürliche Eigenleitung durch thermische Energie hervorrufen, werden ***Minoritätsträger*** genannt. In einem p-dotierten Kristall sind Löcher die Majoritätsträger, in einem negativ dotierten Kristall sind es die Elektronen.

Nun haben wir die physikalischen Vorgänge betrachtet, die der Leitfähigkeit zugrunde liegen. Wie aber können wir diese Leitfähigkeit verändern, um den Strom wie bei einem Relais zu steuern? Dazu müssen wir uns die Vorgänge klarmachen, die ablaufen, wenn wir einen p-dotierten und einen n-dotierten Kristall eng zusammenbringen.

Was passiert an der Grenzfläche ?

2.2 Der p-n Übergang - Die Diode

Abb. 2.13 (oberer Teil) skizziert die Verteilung der Elektronen und der Löcher in einem schmalen Bereich um die Grenzfläche der unterschiedlich dotierten Kristalle.

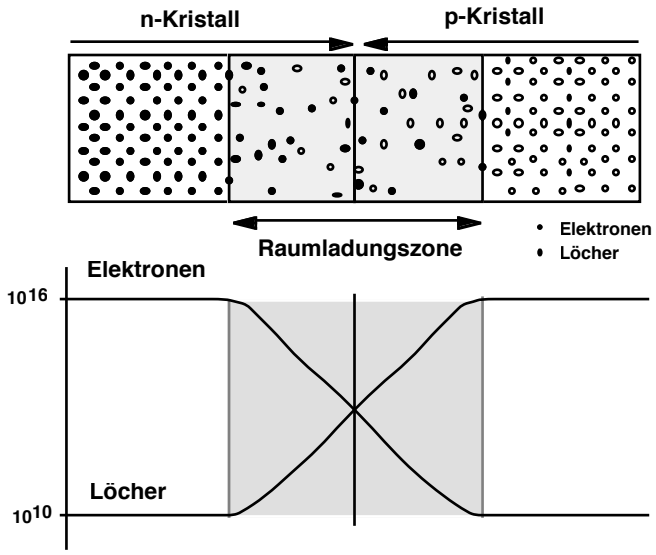


Abb. 2.13 Der p-n Übergang

Im diesem Bereich wandern durch die Wärmebewegung Elektronen in den p-Bereich und Löcher in den n-Bereich, wo sie mit den jeweils komplementären Ladungsträgern rekombinieren. Dadurch werden sie aus der Übergangszone abgezogen, in der dann keine

Majoritätsträger zurückbleiben. Die Grenzschicht wird zum Isolator[5]. Im unteren Teil der Abb. 2.13 ist die Verteilung der Konzentration von Elektronen und Löchern dargestellt. Man sieht, daß an der Grenzschicht insgesamt die wenigsten Ladungsträger existieren. Durch Abwanderung der Elektronen aus dem n-Kristall ensteht eine schmale Zone, die eine leicht positive Ladung aufweist. Analog bildet sich eine leicht negative Ladung durch Abwanderung der Löcher aus dem Grenzbereich des p-Kristalls. Diese lokalen Ladungen um die Grenzfläche werden als *Raumladungszone* bezeichnet.

Diese positive Raumladung verhindert, daß aus dem p-Kristall immer weitere Löcher in den n-Bereich diffundieren. Analog verhindert die negative Raumladung die Wanderung der Elektronen in den n-Kristall. Dadurch wird insgesamt verhindert, daß durch Diffusion schließlich alle Elektronen und Löcher im p-und n-Kristall rekombinieren. In der schmalen Raumladungszone stellt sich also, wenn keine äußeren Einflüsse wirken, ein stabiler Gleichgewichtszustand ein.

Was passiert nun, wenn wir von außen eine elektrische Spannung anlegen? Abb. 2.14 a zeigt die Verhältnisse an der Grenzschicht, wenn eine positive Spannung am n-Kristall und eine negative am p-Kristall anliegt.

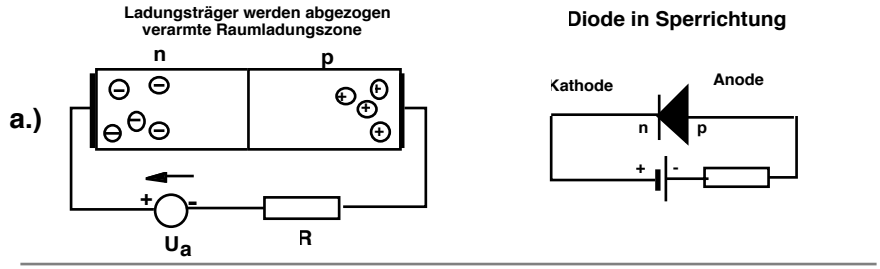


Abb.2.14 a.) Der p-n Übergang bei äußerer Beschaltung (Sperrichtung)

Die Elektronen werden zurm positiven Pol der Spannungsquelle gezogen, die Löcher zum negativen Pol. D.h. alle Ladungsträger werden von der Grenzschicht abgezogen, so daß eine an Ladungsträgern verarmte Zone entsteht. Da nun kein Strom mehr fließen kann, wirkt der p-n Übergang wie ein Isolator. Legen wir eine Spannung in umgekehrter Richtung an, wie in 2.14 b dargestellt, werden die Elektronen und Löcher nun durch die jeweiligen elektrischen Felder in die Übergangszone hineingepreßt, so daß jetzt ein Überschuß an Ladungsträgern vorhanden ist, die rekombinieren. Dadurch wird der Widerstand herabgesetzt und es kann ein entsprechender Strom fließen.

[5] Durch die relativ wenigen Minoritätsträger ist allerdings immer noch Paarbildung und Rekombination infolge thermischer Energie möglich, so daß der Widerstand hoch aber endlich ist.

Die Kombination eines p-Kristalls und eines n-Kristalls führt zu einem wichtigen elektronischen Bauelement, der ***Diode***. Das Schaltzeichen und die Beschaltung einer Diode sind in Abb 2.14 a.) und b.) analog zu den beschriebenen Fällen angegeben. Ist die Diode so geschaltet, daß kein Strom fließt, sagt man: die Diode ist in ***Sperrichtung*** gepolt. Umgekehrt spricht man von ***Durchlaßrichtung***. Die Grenzschicht nennt man wegen ihrer isolierenden Eigenschaft auch ***Sperrschicht***.

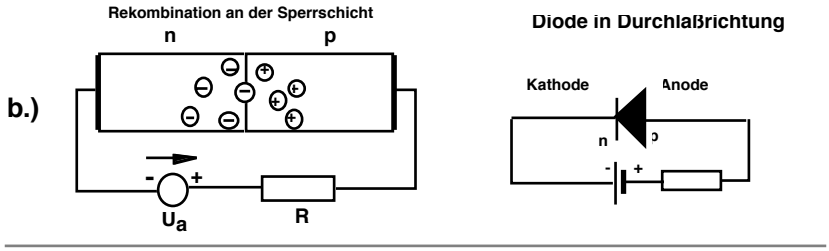


Abb.2.14 b.) Der p-n Übergang bei äußerer Beschaltung (Durchlaßrichtung)

Mißt man den Strom als Funktion der Spannung, der durch eine Diode fließt, erhält man die sogenannte *Diodenkennlinie*, die in Abb. 2.15 dargestellt ist.

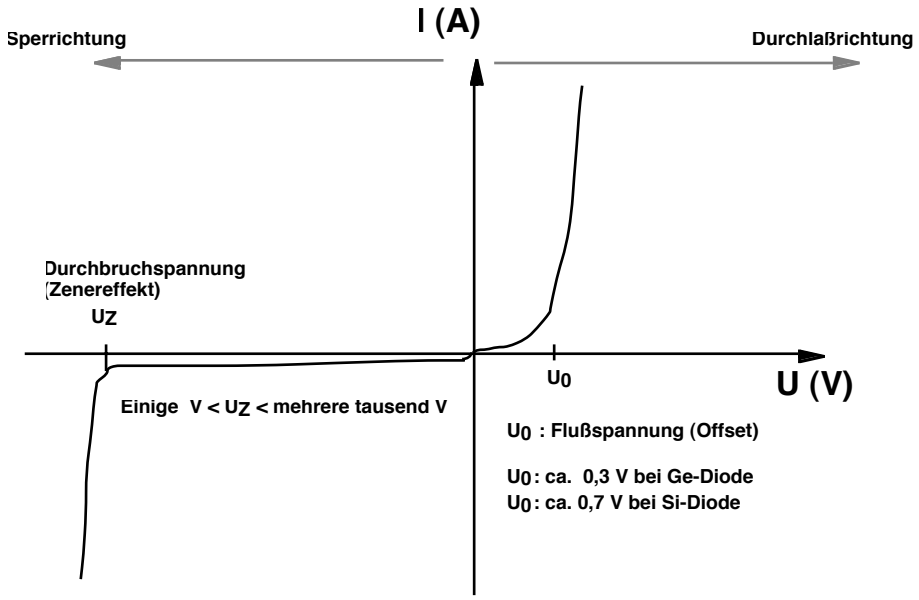


Abb. 2.15 Die Diodenkennlinie

Auf der Abszisse ist die angelegte Spannung, auf der Ordinate der Strom aufgetragen. Man sieht, daß im negativen Abschnitt der Abszisse, in dem die Diode in Sperrichtung arbeitet, über einen weiten Spannungsbereich nur ein geringer Strom fließt. Dies stimmt mit unseren bisherigen Überlegungen überein. Übersteigt die negative Spannung allerdings einen bestimmten Wert, werden durch die hohe Feldstärke lawinenartig Elektronen aus der

Gitterstruktur "herausgeschlagen" und es kommt zum sogenannten Durchbruch. Die entsprechende *Durchbruchspannung* ist abhängig von der technischen Realisierung der Diode und kann von einigen wenigen Volt bis zu mehreren tausend Volt betragen[6]. Im positiven Abschnitt der Abszisse fließt bei niedrigen Spannungswerten zunächst auch nur ein sehr geringer Leckstrom, d.h. wesentlich geringer als man im Durchlaßbetrieb der Diode erwarten würde. Erst ab einer bestimmten Spannung steigt der Strom, wie erwartet sehr stark an, da der Widerstand im Durchlaßbereich auf einen sehr kleinen Wert sinkt[7]. Die positive Spannung, die erst überschritten werden muß, damit die Diode in den Durchlaßbereich kommt (ca. 0,3 V bei einer Germanium-Diode und ca. 0,7 Volt bei einer Silizium-Diode), wird als Offset oder Schleusenspannung bezeichnet. Sie hängt mit der für den jeweilige p-n Übergang typischen Raumladung zusammen, die erst durch die von außen angelegte Spannung kompensiert werden muß.

Mit der Diode haben wir ein Bauteil, das, wie ein Rückschlagventil, den Strom nur in eine Richtung durchläßt[8]. Aber den Widerstand und damit den Strom steuern, um einen Schalter zu realisieren, können wir noch nicht. Wie am Anfang bereits erwähnt, entdeckten Bardeen und Brattain den Transistor, der gerade das leistet, durch Zufall bei Untersuchungen an p-n Übergängen bei Dioden. Als sie eine Prüfspitze auf die Sperrschicht aufsetzten, stellten sie dadurch unbeabsichtigt einen weiteren Übergang her und schufen eine p-n-p Zonenfolge, die überaschende Eigenschaften aufwies.

2.3 Der Transistor

Wir wollen nun die Vorgänge in einer Folge von positiv und negative dotierten Halbleiterkristallen betrachten. Dazu wählen wir die n-p-n Zonenfolge. Die Zonen bezeichnen wir als Kollektor, Basis und Emitter. Durch diese Konfiguration erhalten wir zwei Diodenübergänge, eine Emitter-Basis Diode und eine Kollektor-Basis Diode. Legen wir entsprechende Spannungen an, wie in Abb. 2.16 gezeigt wird, ist die Emitter-Basis Diode in Durchlaßrichtung geschaltet, die Kollektor-Basis Diode in Sperrichtung. Im Emitter-Basis Kreis fließt also ein Strom. Der Emitter-Kollektor Kreis ist dagegen gesperrt. Wird die Basiszone nun sehr dünn gemacht, wie in Abb. 2.17 gezeigt, passiert folgendes:

---

6 Dieser Effekt wird auch Zener-Effekt genannt und wird zur Spannungsstabilisation ausgenutzt. Entsprechende, speziell für diesen Zweck gebauten Dioden, heißen "Zener-Dioden".

7 Es sei noch einmal das ohmsche Gesetz in Erinnerung gerufen: Spannung (U) = Widerstand (R) * Strom (I). Für den Strom gilt dann: I = U/R, d.h. bei R gegen 0 wird der Strom unendlich groß. In Abb. 2.14 dient daher der eingezeichnete Widerstand der Strombegrenzung.

8 Eine der Hauptanwendungen für Dioden ist deshalb die Gleichrichtung von Wechselströmen.

Nur ein Teil der Elektronen, die durch die Emitter-Basis Spannung eine hohe kinetische Energie erhalten, rekombiniert in der dünnen Basiszone mit den dort vorhandenen Löchern.

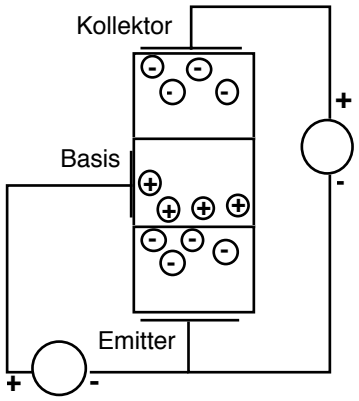


Abb. 2.16 Eine n-p-n Zonenfolge mit äußerer Beschaltung

Der weitaus größte Teil gelangt in die Kollektorzone, in der dadurch ein Überschuß an negativen Ladungsträgern entsteht. Durch die Emitter-Kollektor-Spannung werden diese negativen Ladungsträger weiter zur positiven Kollektorseite hingezogen. Dadurch fließt im Emitter-Kollektor-Kreis ebenfalls ein Strom. Technisch kann man die Basiszone so dünn machen, daß nur ca. 1% der Elektronen hier rekombinieren, während 99% einen großen Strom im Emitter-Kollektor Kreis hervorrufen. Das bedeutet, daß ein relativ kleiner Strom im Emitter-Basis-Kreis einen großen Strom im Emitter-Kollektor-Kreis hervorruft. Unterbricht man den Emitter-Basis-Kreis, so daß hier kein Strom fließen kann, wird auch der Stromfluß im Emitter-Kollektor-Kreis unterbrochen, da die Basis-Kollektordiode jetzt sperrt.

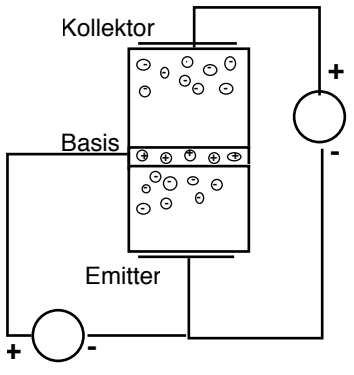


Abb. 2.17 n-p-n Zonenfolge mit sehr dünner Basiszone

Wir können also mit einem relativ kleinen Strom im Emitter-Basis Kreis einen großen Strom im Emitter-Kollektor-Kreis steuern und haben damit die Basis für einen elektronischen Schalter gefunden.

Abb. 2.18 zeigt die übliche (elektrotechnische) Darstellung, ein sogenanntes Schaltbild, eines Transistors in Emitter-Grundschaltung. Die Emitter-Grundschaltung soll genutzt werden, um die wichtigsten Eigenschaften eines Transistors einzuführen.

Die Betriebsspannung V_{CC} ist bei dem gezeigten npn-Transistor (d.h. die Zonenfolge ist n-p-n) positiv gegenüber dem Nullpotential, das mit Ground (GND) bezeichnet wird. Die Widerstände, die in unseren prinzipiellen Überlegungen nicht berücksichtigt wurden, dienen dazu, die Ströme in den jeweiligen Kreisen zu begrenzen, damit der Transistor nicht beschädigt wird. Der Lastwiderstand R_L stellt den ohmschen Widerstand eines angeschlossenen Verbrauchers dar[9]. Liegt keine Signalspannung an, ist die Emitter-Kollektor Strecke des Transistors gesperrt. Die Spannung am Kollektorausgang ist dann positiv gegenüber GND und wird durch das Verhältnis der Widerstände R_C und R_L bestimmt. Legt man zwischen den Basiswiderstand und GND eine Steuerspannung, beginnt der Transistor zu leiten, d.h. der Widerstand der Emitter-Kollektor Strecke wird gering und es fließt ein Strom von GND über den Transistor und den Kollektorwiderstand nach V_{CC}. Die Spannung am Kollektor sinkt auf einen geringen Wert (0,3V - 0,7V), der, wie bei der Diode, durch die Raumladungszonen an den Grenzschichten bedingt ist. Das Verhältnis der Steuerspannung zur Ausgangsspannung und des Steuerstroms im Basiskreis (I_B) zum Kollektorstrom (I_C) sind die wesentlichen Charakteristika eines Transistors und werden durch Kennlinien dargestellt.

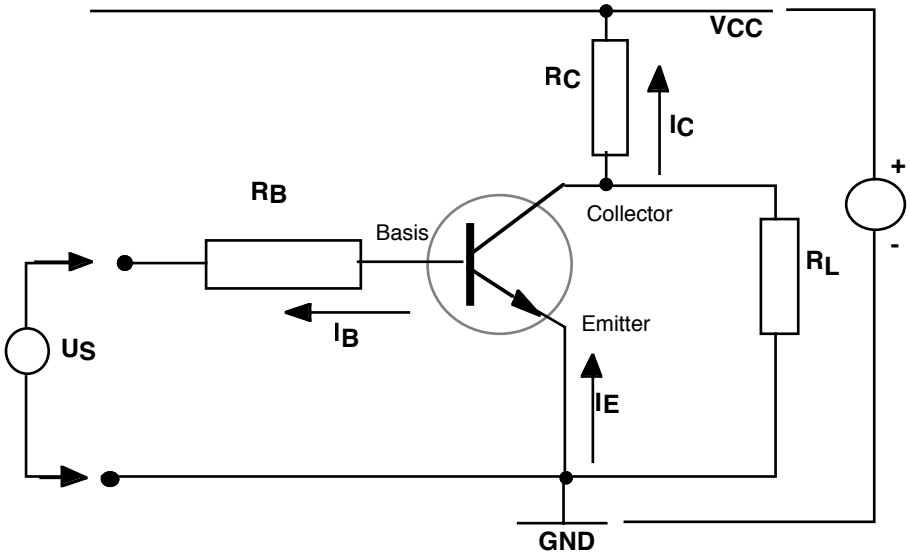


Abb. 2.18 Emitter Grundschaltung

Notation:
V_{CC} : Betriebsspannung (Collector Circuit)
GND : Null-Potential (Erde, Ground)
U_S : Steuerspannung

[9] z.B. eine Lampe, ein Motor oder ein weiteres Schaltelement.

R_B : Basiswiderstand
R_C : Kollektorwiderstand
I_B : Basisstrom
I_E : Emitterstrom
I_C : Kollektorstrom
R_L : Lastwiderstand

In Abb. 2.19 a - c sind die wesentlichen Kennlinien eines Standard npn-Transistors (BC108) zusammengestellt. Abb. 2.19 a zeigt die Stromverstärkung, d.h. das Verhältnis des Basisstroms zum Kollektorstrom. Wir sehen, daß eine etwa 250-fache Verstärkung vorliegt. Zur Bestimmung der Spannungsverstärkung betrachten wir zunächst das Verhältnis zwischen

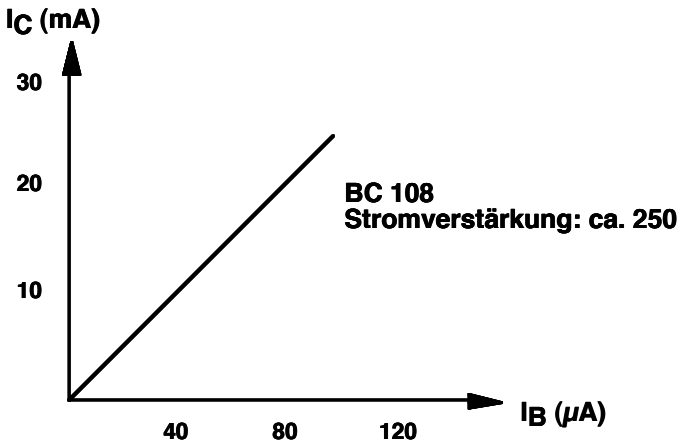


Abb. 2.19 a.) Stromverstärkung eines Transistors

Basisstrom und Basis-Emitter-Spannung (Abb. 2.19 b.). Wir sehen die typische Diodenkennlinie (vgl Abb. 2.15), d.h. der Basisstrom steigt bei ca. 0,7 V steil an. Setzen wir Abb. 2.19 a.) und b.) zueinander in Beziehung sehen wir, daß bereits eine sehr kleine Differenzspannung addiert zur Schleusenspannung einen steil ansteigenden Basisstrom verursacht und damit (bei linearer 250-facher Stromverstärkung) einen entsprechend großen Kollektorstrom.

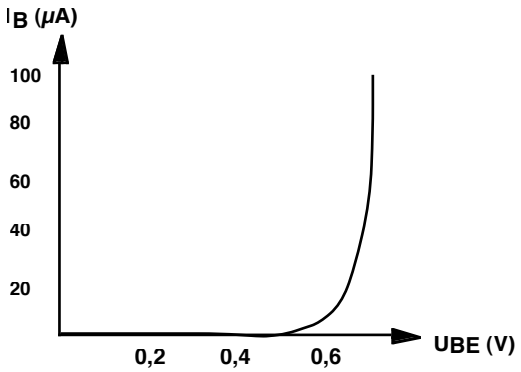


Abb. 2.19 b.) Abhängigkeit des Basisstroms von der Emitter-Basis Spannung

Zusammengefaßt sind die Kennlinien in Abb. 2.19 c.). Man kann sehen, daß die kleine Spannungsänderung an der Basis einen entsprechend großen Strom im Kollektorkreis hervorruft. Die Spannungsverstärkung hängt von der Betriebsspannung und vom Kollektorwiderstand ab. Sie liegt typischerweise bei ca. 1000.

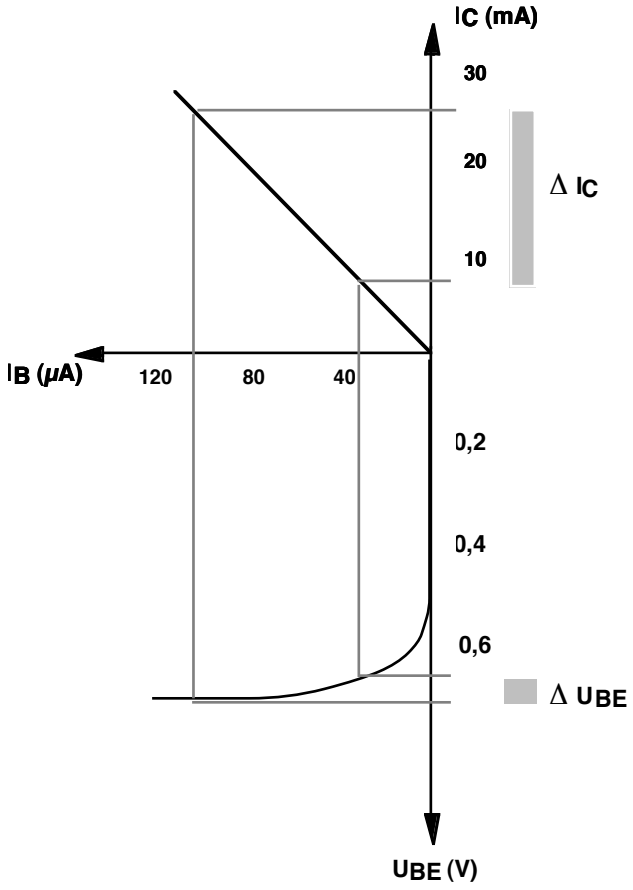


Abb. 2.19 c.) Bezug zwischen Basisspannung und Kollekorstrom

2.4 Der Feldeffekttransistor

Obwohl der Transistor, der auf dem vorgestellten Leitungsprinzip von Elektronen und Löchern beruht und deshalb auch bipolarer Transistor genannt wird, den Ausgangspunkt für die Revolution der Computertechnologie darstellt, spielt seine Technologie bei der heutigen Hochintegration keine Rolle mehr. Sein wesentlicher Nachteil besteht darin, daß für seine Steuerung im Emitter-Basis Kreis ein, wenn auch geringer, Strom fließen muß. Bei tausenden oder heute Millionen von Transistoren auf einem einzigen Chip wäre der Stromverbrauch zu hoch. Ein Schalter, der diesen Nachteil nicht hat und der einen Strom leistungslos[10] nur durch

[10] Da kein Strom fließt, ist auch die Leistung, die aus dem Produkt von Strom und Spannung gebildet wird, bis auf einen unbedeutenden Rest gleich Null.

ein elektrisches Feld steuern kann, wurde mit dem ***Feldeffekttransistor*** (FET) gefunden. Das Prinzip des FETs ist in Abb. 2.20 skizziert.

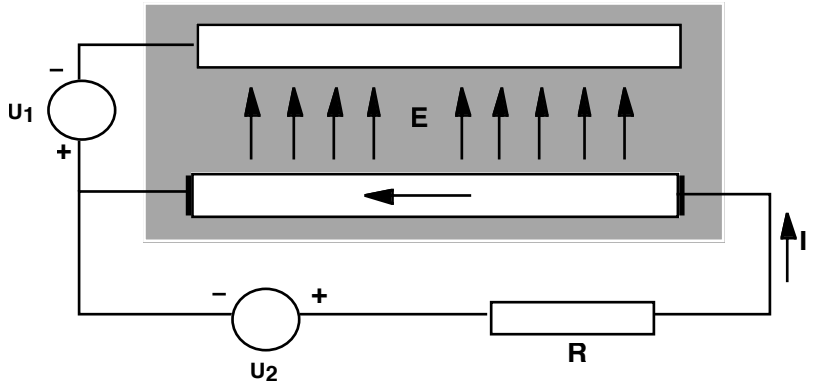


Abb. 2.20 Prinzip des Feldeffekttransistors (Verdrängungstyp, Depletion Type)

Der FET besteht aus zwei elektrisch leitenden Platten, die, etwa wie bei einem Kondensator, parallel zueinander stehen und voneinander isoliert sind. Durch die Spannung U_1 wird ein elektrisches Feld E zwischen diesen Platte erzeugt. Durch die Spannungsquelle U_2 fließt über den Widerstand R und den Widerstand, den die untere Platte bildet, ein Strom I. Die Ladungsträger für diesen Strom sind Elektronen. Wird nun die Spannung U_1 entsprechend erhöht, wirkt das elektrische Feld auf die untere Platte und verdrängt dort die Elektronen. Die Folge davon ist eine Verarmung an Ladungsträgern und damit verbunden eine Erhöhung des Widerstandes für den unteren Stromkreis.

Da die obere Platte vollständig isoliert ist, kann kein Strom fließen, die Steuerung des Widerstands der unteren Platte ist also völlig leistungslos. Da für das Prinzip des FETs nur eine Art von Ladungsträgern, nämlich Elektronen, eine Rolle spielen, nennt man den FET auch ***unipolaren Transistor***. Da die Steuerung auf einer Verdrängung von Elektronen beruht, wird dieser Typ von FETs als ***Verarmungstyp (Depletion Type)*** bezeichnet.

Das Prinzip der FETs wurde bereits um die Jahrhundertwende patentiert. Es fehlte aber ein Material, mit dem sich der FET technisch realisieren ließ. Metall war als Plattenmaterial nicht geeignet, da es eine zu gute Leitfähigkeit besitzt und der Effekt des Feldes nicht groß genug ist. Isolatoren wiederum haben eine zu geringe Leitfähigkeit.

Erst mit der Verfügbarkeit geeignet dotierter Halbleitermaterialien konnte man FETs realisieren. Durch die entsprechende Dotierung kann man den Widerstand durch die Verunreinigung mit freien Ladungsträgern genau “einstellen“.

Abb. 2.21 deutet die technische Realisierung eines FETs an.

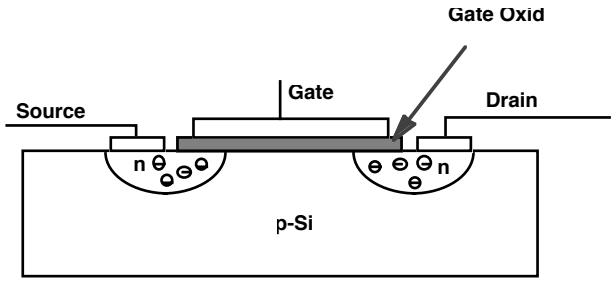


Abb. 2.21 FET Anreicherungstyp (Enhancement Type)

Source (Quelle) und Drain (Senke) sind n-dotierte (d.h. mit freien Elektronen angereicherte) Mulden in einem Basissubstrat aus p-dotiertem Silizium (p-Si). Das Gate (Gitter) wird durch eine sehr dünne Isolationsschicht vom Substrat isoliert. Als Isolation wird oft Metalloxyd verwendet, weshalb ein entsprechender FET auch als ***MOS-FET*** *(Metal-Oxyd-Silicon-FET)* bezeichnet wird. Sind Source und Drain n-dotierte Halbleiter (Substrat p-dotiert), spricht man vom ***NMOS FET***, umgekehrt heißt ein FET, der p-dotierte Source und Drain hat (Substrat n-dotiert), ***PMOS FET***.

Abb. 2.22 zeigt Sperr- und Durchlaßmodus eines FETs bei äußerer Beschaltung. Legt man eine Spannung zwischen Source und Drain, passiert zunächst nichts. Der Übergang zwischen Drain und Substrat wirkt wie eine Diode in Sperrichtung (Abb. 2.22 a).

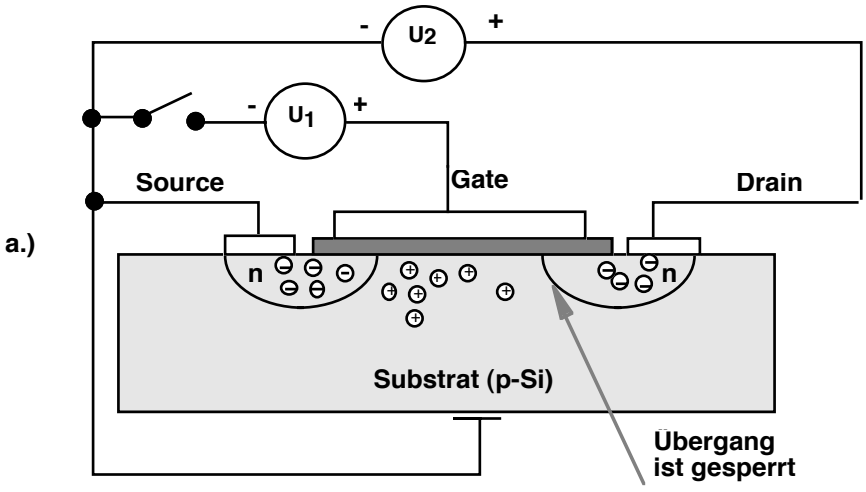


Abb. 2.22 a FET im Sperrzustand

Zwischen den beiden p-Mulden befinden sich positive Ladungsträger des Substrats. Legt man nun eine Steuerspannung U_1 an das Gate (Abb. 2.22 b), werden die positiven Ladungsträger verdrängt und gleichzeitig eine dünne Zone mit negativen Ladungsträgern angereichert, der

sogenannten *Kanal*, wird gebildet. Nun können, bedingt durch das elektrische Feld von U_2, Elektronen über diesen Kanal transportiert werden, wobei der negative Pol von U_2 als Elektronenquelle, der positive Pol als Senke fungiert. Wegen der Anreicherung von Ladungsträgern wird dieser FET als ***Anreicherungstyp*** *(Enhancement Type)* bezeichnet.

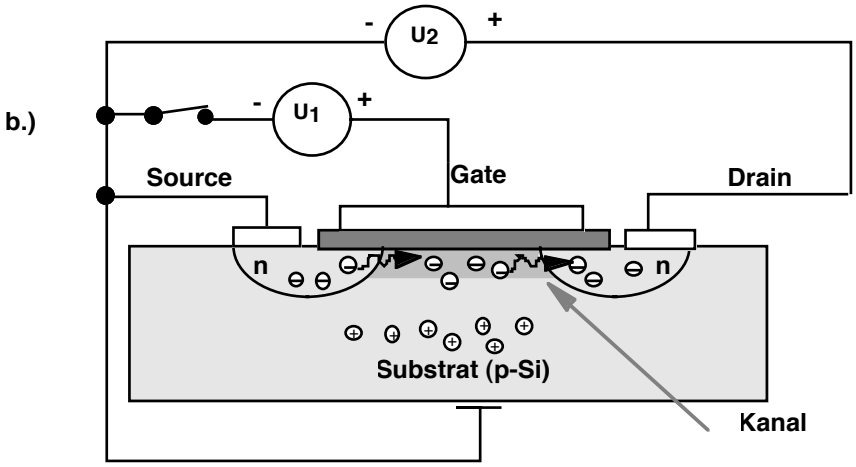


Abb. 2.22 b FET im Durchlaßzustand

Da das Gate vollständig isoliert ist, fließt kein Steuerstrom. Der FET wird also leistungslos gesteuert. Abb. 2.23 zeigt das Schaltsymbol für einen MOS-FET (wobei der Kreis oft weggelassen wird). Trotz der völlig anderen physikalischen Grundprinzipien des FETs ist die Wirkungsweise und auch die Basisbeschaltung dem des bipolaren Transistors sehr ähnlich.

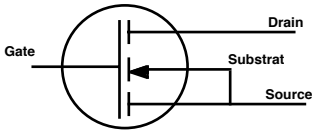


Abb. 2.23 FET Schaltsymbol

Abb. 2.24 zeigt die Grundschaltung eines Inverters, der mit N-MOS FETs aufgebaut ist.

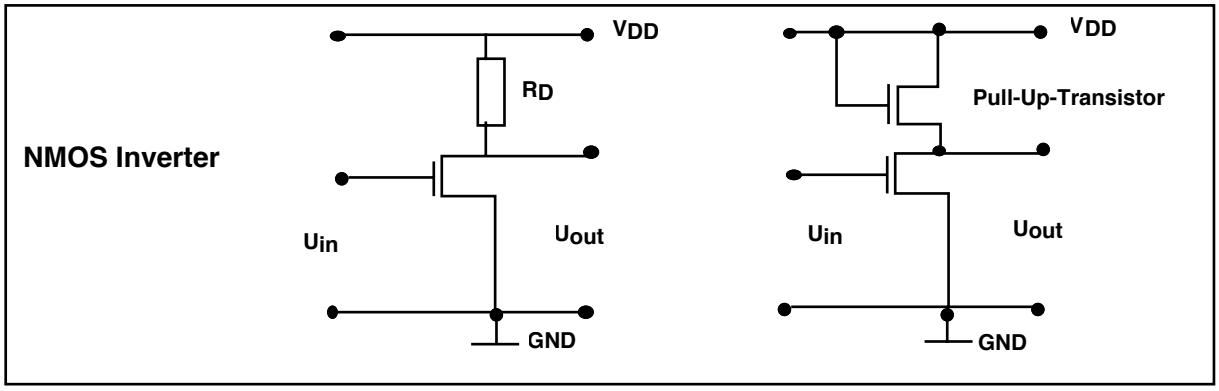


Abb. 2.24 Grundschaltung eines FET-Inverters

Sie entspricht einer Emitter-Grundschaltung beim bipolaren Transistor. Der Substratanschluß, der im Schaltsymbol (Abb. 2.23) gezeigt ist, wird oft im Schaltbild weggelassen, da er standardmäßig mit dem Source-Anschluß verbunden ist. Der Drain Widerstand R_D wird meist aus Gründen einer einfacheren Realisierung als sogenannter Pull-Up-Transistor ausgeführt, der die Aufgabe der Widerstands übernimmt.

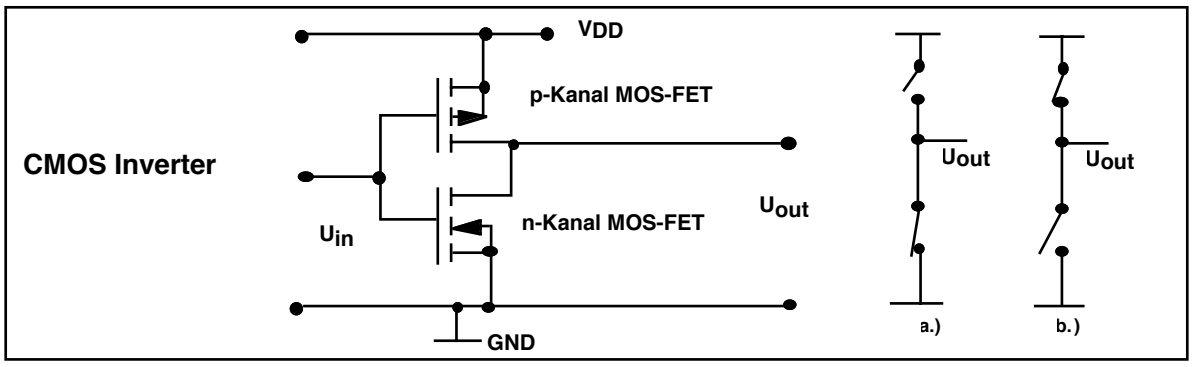


Abb. 2.25 Schaltung eines CMOS Inverters

Der Leistungsverbrauch einer solchen Schaltung resultiert nur noch aus dem Strom, der im Durchlaßfall des FETs durch R_D fließt und in Wärme umgesetzt wird. Dies ist zwar ein sehr geringer Leistungsverbrauch, der sich aber im Fall der Hochintegration millionenfach zu einem problematischen Leistungsverbrauch addiert.

Eine Technologie, die diesen Nachteil nicht hat, ist die sogenannte ***CMOS-Technologie*** (CMOS: Complementary MOS). Ein CMOS Inverter ist aus einem NMOS FET und einem PMOS FET, d.h. komplementären FETs aufgebaut. Seine Schaltung ist in Abb. 2.25 angegeben.

Liegt am Eingang dieser Schaltung eine positive Spannung U_{IN}, leitet der NMOS FET (auch n-Kanal MOS-FET genannt). Der PMOS FET ist gesperrt, weshalb kein Strom zwischen GND und V_{DD} fließen kann. Am Ausgang liegt eine Spannung U_{OUT} = 0 Volt gegenüber GND. Dieser Zustand ist in Abb. 2.25 (a) dargestellt, in der die FETs als Schalter abstrahiert wurden. Umgekehrt, liegt am Eingang eine Spannung von 0 V, leitet der PMOS FET und der NMOS FET ist gesperrt, so daß auch in diesem Fall kein Strom fließt. Am Ausgang liegt eine Spannung U_{OUT}= V_{DD} (Abb. 2.25 (b). Ein CMOS Gatter verbraucht deshalb selbst keinen Strom, wenn es sich in einem der beiden Zustände befindet. Lediglich beim Umschalten von einem in den anderen Zustand wird Strom verbraucht.

Heute sind (fast) alle Hochleistungs-Mikroprozessoren in CMOS-Technologie aufgebaut. Wir werden nun kurz die Auswirkungen der Miniaturisierung betrachten.

2.5 Auswirkung der Miniaturisierung elektronischer Bauelemente

Zwei Entwicklungen führen zu immer komplexeren integrierten Schaltkreisen:
1. die Verkleinerung der Bauelemente auf einem Chip
2. die Größe der nutzbaren Chipfläche.

Abb. 2.26 zeigt die Entwicklung und einen Ausblick bei der Strukturverkleinerung am Beispiel von Speicherchips (Quelle: Computer Zeitung 26/1996). Man kann sehen, daß eine projektierte Strukturgröße von ca. 100nm (nm: Nanometer) zu einer Speichergröße von ca. 1 GBit führt. Die stetige Verkleinerung der Strukturen führt allerdings immer näher an eine physikalische Grenze, unter der die bisherige Halbleitertechnologie nicht mehr angewendet werden kann. Bei der Verkleinerung um eine weitere Größenordnung, um an Strukturgrößen von 10 nm heranzukommen, verlieren die Ladungsträger ihre bis dahin geltenden Eigenschaften und Quanteneffekte werden wirksam. Eine völlig neue Technologie wird benötigt[11]. Die Grenzen der bisherigen Halbleitertechnologie werden deshalb, folgt man den Prognosen, in ca. 10 Jahren erreicht.

Allgemein erzielt man bei linearer Verkleinerung von 1 auf 1/sk (sk: Skalierungsfaktor der Komponenten):

Leitungs-Laufzeiten:	von	t	auf	t/sk
Ströme:	von	I	auf	I/sk
Anzahl der Bauelemente	von	Z	auf	$sk^2 * Z$

[11] Die grundlegenden Prinzipien der sogenannten "Quantentransistoren" sind bereits bekannt. Allerdings fehlt eine Technologie, um diese Prinzipien umzusetzen. Man kann die Situation mit der des FETs vergleichen, dessen physikalisches Prinzip ca. 80 Jahre vor seiner technischen Realsierung bekannt war.

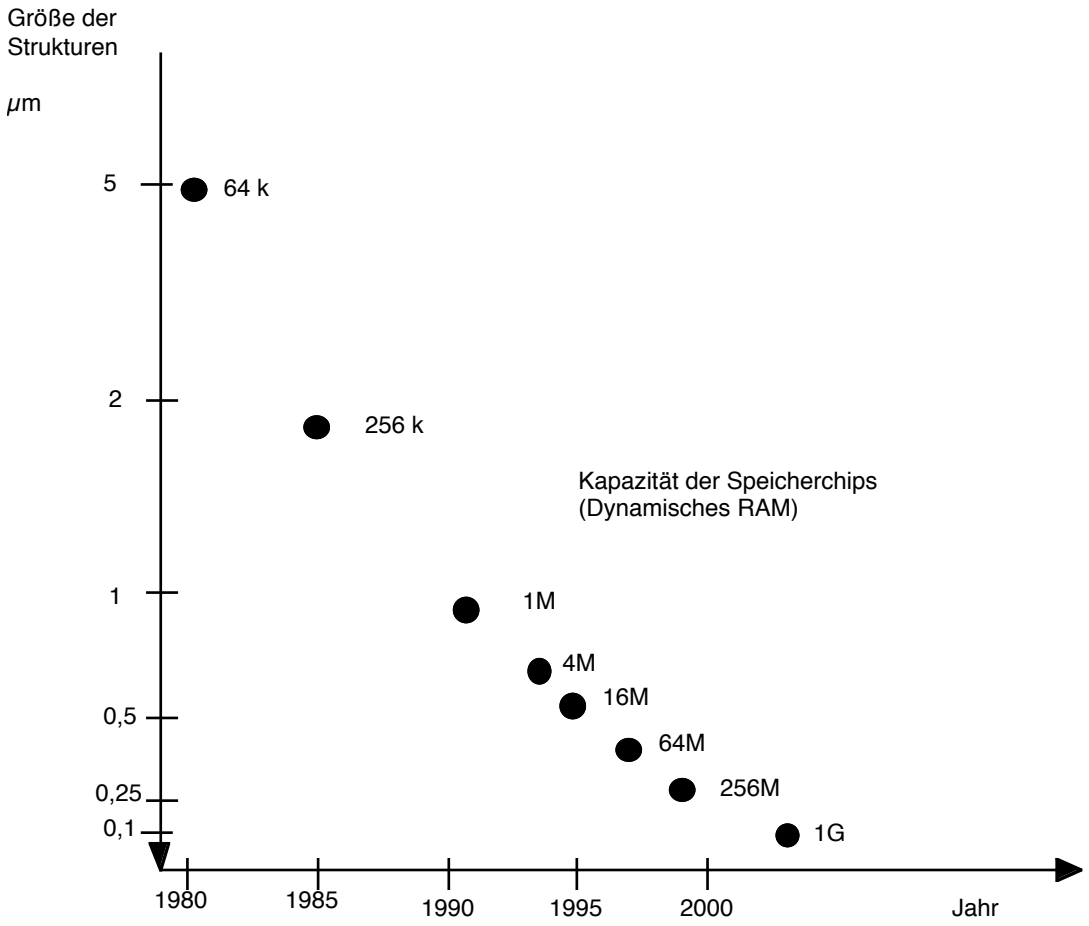


Abb. 2.26 Strukturverkleinerung und Kapazität von DRAM

Der Vorteil, den man durch die Verkleinerung erzielt, liegt zunächst in den kürzeren Abständen zwischen den Komponenten und damit verbunden mit kleineren (lokalen) Verzögerungszeiten. Auch der Strom, den die einzelnen Komponenten verbrauchen, wird etwa linear reduziert. Dadurch, daß die Anzahl der Bauelemente auf derselben Chipfläche bei linearer Verkleinerung aber quadratisch wächst, steigt der Gesamtleistungsverbrauch pro Fläche und damit die entwickelte Wärme an. Da wir gesehen haben, daß die Temperatur für ein korrektes Verhalten der Halbleiter in einem bestimmten Bereich gehalten werden muß, sind aufwendige Gehäuse- und Kühltechniken für hochintegrierte Chips erforderlich. Außerdem muß die elektrische Leistung auch an die entsprechenden Stellen transportiert werden, was das Problem der ausreichenden Dimensionierung der Versorgungsleitungen nach sich zieht.

Die Verkleinerung der Abstände und die kürzeren Verzögerungszeiten gelten im wesentlichen nur für lokale, benachbarte Verbindungen. Für weitreichende Vebindungen dagegen wird die Situation schlechter, was in Abb. 2.27 verdeutlicht ist.

Beim Entwurf von Prozessoren ist daher darauf zu achten, daß beliebige globale Verbindungen möglichst vermieden werden. Die Realisierung der Gesamtfunktionalität durch kooperierende, teilweise autonome Funktionseinheiten ist notwendig.

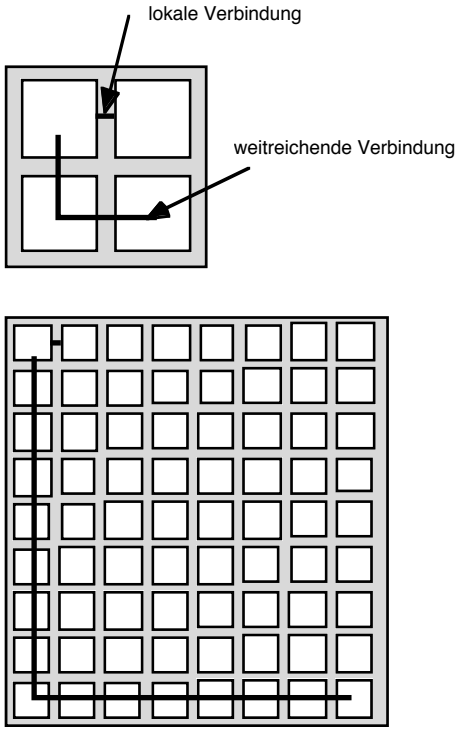


Abb. 2.28 Auswirkung der Hochintegration auf globale Verbindungen.

Die Seitenlänge der Komponenten wird um den Faktor 2 verkleinert, während die Seitenlänge des Gesamtchips um den Faktor 2 vergrößert wird.

Eine Ordnung der Funktionseinheiten, welche Lokalität berücksichtigt und häufig miteinander kommunizierende Einheiten auch in räumlicher Nähe plaziert, hilft das Problem zu mildern. Trotzdem müssen häufig globale Signale, wie z.B. der Systemtakt, gleichmäßig über den ganzen Chip verteilt werden. Dieses Problem wird durch den immer schneller werdenden Prozessortakt, der in den nächsten 5 Jahren zwischen 500 und 1000 Mhz liegen wird, verschärft.

Die Rechnerorganisation muß unter diesen Randbedingungen optimale Strukturen entwickeln.

3 Technische Grundkomponenten zur Realisierung von Rechnern

3.1 Logikfamilien

3.2 Programmierbare Logikkomponenten

Wir haben bei der Behandlung der digitalen Logik die Realisierung der Gatter und Flip-Flops nicht weiter betrachtet. Wenn man auf der logischen Ebene ein Schaltnetz entwirft, erhält man das, was im Englischen treffend als "Random Logic" bezeichnet wird, also eine beliebig irreguläre Struktur von logischen Gattern und Flip-Flops. Integrierte Schaltungen der ersten Generation realisierten eine geringe Anzahl dieser Gatter oder Flip-Flops auf einem Chip, die dann entsprechend miteinander verschaltet werden mußten. Mit den Fortschritten in der Integrationstechnik wurde es möglich, auch komplexere Standardstrukturen wie Addierer, Register, Kodierer und Dekodierer bis hin zu kompletten ALUs auf einen Chip zu bringen. Diese Standardchips waren aus vielen Gründen unpraktisch. Insbesondere konnte man keine Ein-Chip-Lösungen für Spezialfunktionen, wie z.B. einfache Steueraufgaben, realisieren, obwohl der Integrationsgrad dafür ausgereicht hätte. Einen Ausweg stellen die programmierbaren Logikkomponenten dar. Die Grundlage der programmierbaren Logikkomponenten ist dadurch gegeben, daß man beliebige logische Funktionen aus nur wenigen Grundfunktionen bauen kann. Jede logische Funktion läßt sich als Summe von Produkten (disjunktive Normalform) oder Produkt von Summen (konjunktive Normalform) darstellen. Dazu sind nur die Grundfunktionen +, •, und die Negation erforderlich.

3.2.1 Einfache Komponenten

Nehmen wir an, unser erstes Ziel sei die Entwicklung eines Universalgatters, das alle 16 zweistelligen boolschen Funktionen realisieren kann, die in Abb. 3.1 angegeben sind. Für dieses Gatter soll physikalisch eine Struktur vorgegeben werden, der nachträglich die spezifische logische Funktion aufgeprägt werden kann. Wir gehen von einer Darstellung der logischen Funktionen als Summe von Produkten aus.

a	b	f_0	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	f_9	f_A	f_B	f_C	f_D	f_E	f_F
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

f_1: UND (AND)
f_6: EXOR
f_7: ODER (OR)
f_E: NAND
f_8: NOR

Abb. 3.1 Die 16 zweistelligen boolschen Funktionen

Es gibt für die zwei Variablen a und b vier Minterme. Durch Kombination der Minterme können wir jede der 16 Funktionen darstellen.

Abb. 3.2 zeigt eine erste mögliche Lösung für unser Universalgatter.

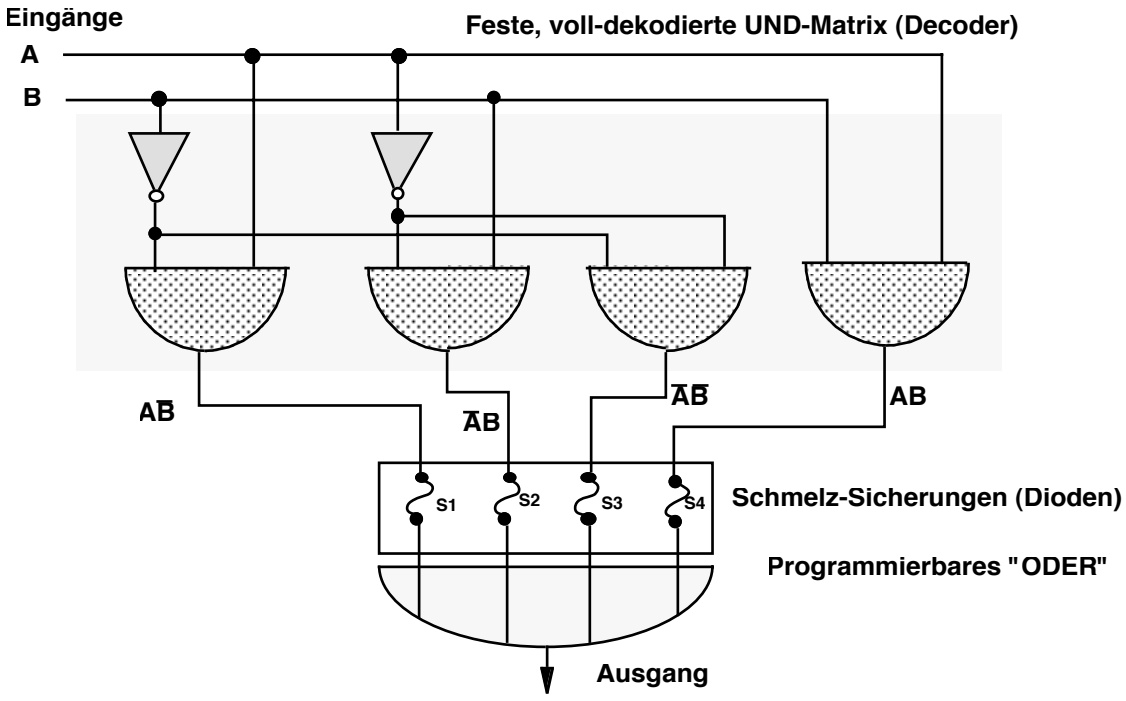


Abb. 3.2 Ein programmierbares Universalgatter (PROM-Ansatz)

Ein Eingangsdekodierer, der aus zwei Invertern und 4 UND-Gattern besteht, hat für jeden Minterm genau einen Ausgang. Welcher dieser Ausgänge für eine bestimmte logische Funktion relevant ist, wird durch ein programmierbares ODER-Gatter bestimmt. Die Ausgänge des Dekoders sind über Schmelzsicherungen mit den Eingängen des ODER-Gatters verbunden. Will man z.B. die logische Funktion f_C = A'B' + A'B realisieren, muß man S1 und S4 zerstören. Die intakten Eingänge des ODER-Gatters liefern genau dann eine 1 am Ausgang, wenn die entsprechende Belegung der Eingänge A und B gegeben sind. Die Schmelzsicherungen könnte man auch als Speicherelemente eines Festwertspeichers (Programmable Read-Only Memory (PROM)) auffassen. Eine intakte Sicherung repräsentiert eine '1', eine zerstörte Sicherung eine '0'. Durch den Eingangsdekoder wird für jede mögliche Belegung genau ein Festwertspeicherplatz, in diesem Fall eine Sicherung, ausgewählt. Der entsprechende Wert wird über das ODER-Gatter zum Ausgang propagiert.

Einen alternativen Ansatz, der einen programmierbaren Eingangsdekoder (UND-Gatter) und ein festes ODER-Gatter besitzt, ist in Abb. 3.3 gezeigt. Wieder ist der Ausgangspunkt, daß die Funktion durch zwei Minterme dargestellt werden kann. Allerdings kann man nun durch Zerstören der Sicherungen bestimmen, welcher Minterm durch das jeweilige UND-Gatter realisiert werden soll. Für die logische Funktion f_C = A'B' + A'B müssen wir im rechten UND-Gatter die Sicherungen S1 und S3 zerstören, im linken UND-Gatter S5 und S8. Schaltungen, die so arbeiten, werden als Programmable Array Logic (PAL) bezeichnet.

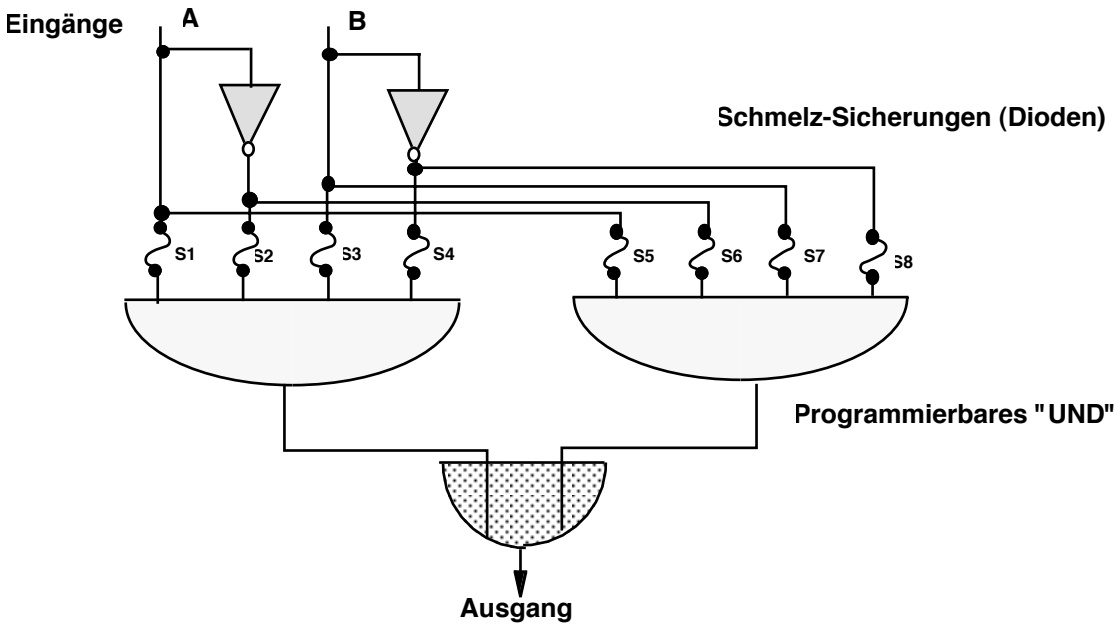


Abb. 3.3 Programmierbares Gatter (PAL-Ansatz)

Festzuhalten bleibt, daß der PROM-Ansatz auf einer festen UND-Matrix und einer programmierbare ODER-Matrix beruht, während ein PAL eine programmierbare UND-Matrix und eine feste ODER-Matrix besitzt. Der Begriff "Matrix" wird deutlich, wenn man Abb. 3.4 betrachtet, in der das Universalgatter in der üblichen PAL-Notation gezeigt wird.

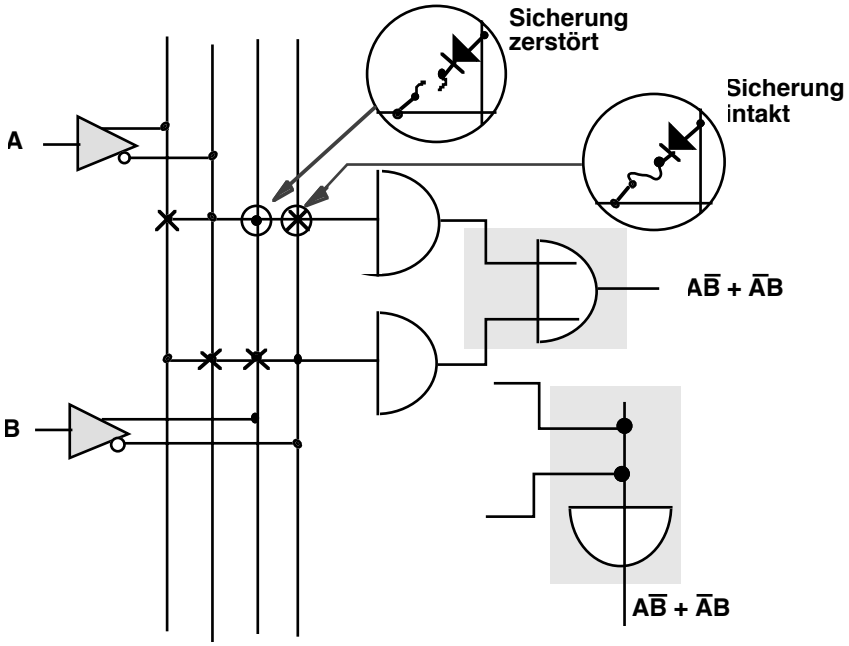


Abb. 3.4 PAL Darstellung 1

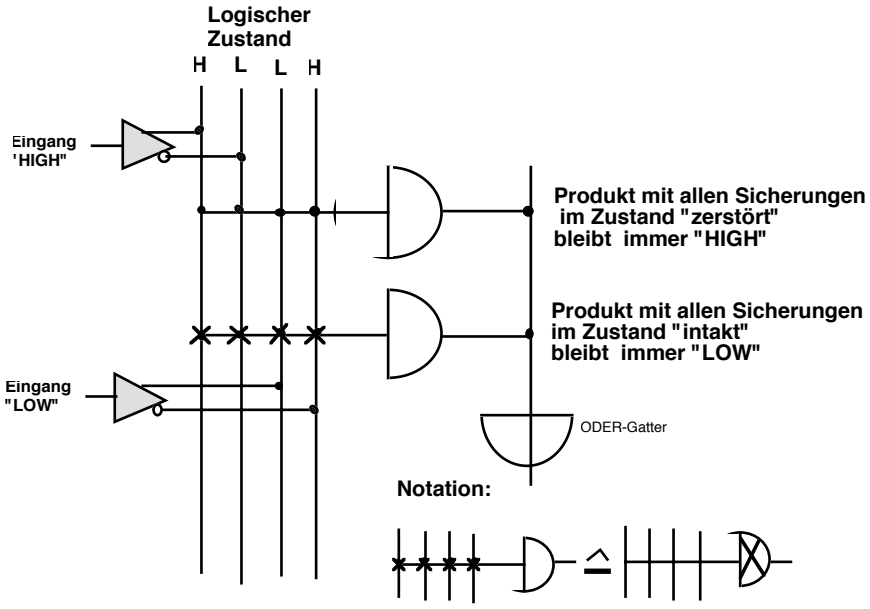


Abb. 3.5 PAL Darstellung 2

Eine intakte Sicherung wird durch ein Kreuz markiert. Alle nicht markierten Sicherungen werden durch einen Programmiervorgang zerstört. Man sieht, daß die Sicherungen zur Programmierung in Matrixform angeordnet sind. Eine Zeile der Matrix stellt jeweils die Eingänge eines Mehrfach-UND-Gatters dar. Durch eine Spalte wird ein individueller Eingang auf alle verfügbaren Mehrfach-UND-Gatter gelegt. Abb. 3.5 zeigt die vereinfachte Notation, wenn alle Sicherungen intakt oder wenn alle Sicherungen zerstört sind.

Normalerweise hat eine programmierbare Komponente natürlich wesentlich mehr Eingänge als unser Universalgatter. In Abb. 3.6 ist ein Beispiel für die Realisierung eines einfachen kombinatorischen Schaltnetzes durch ein PAL angegeben.

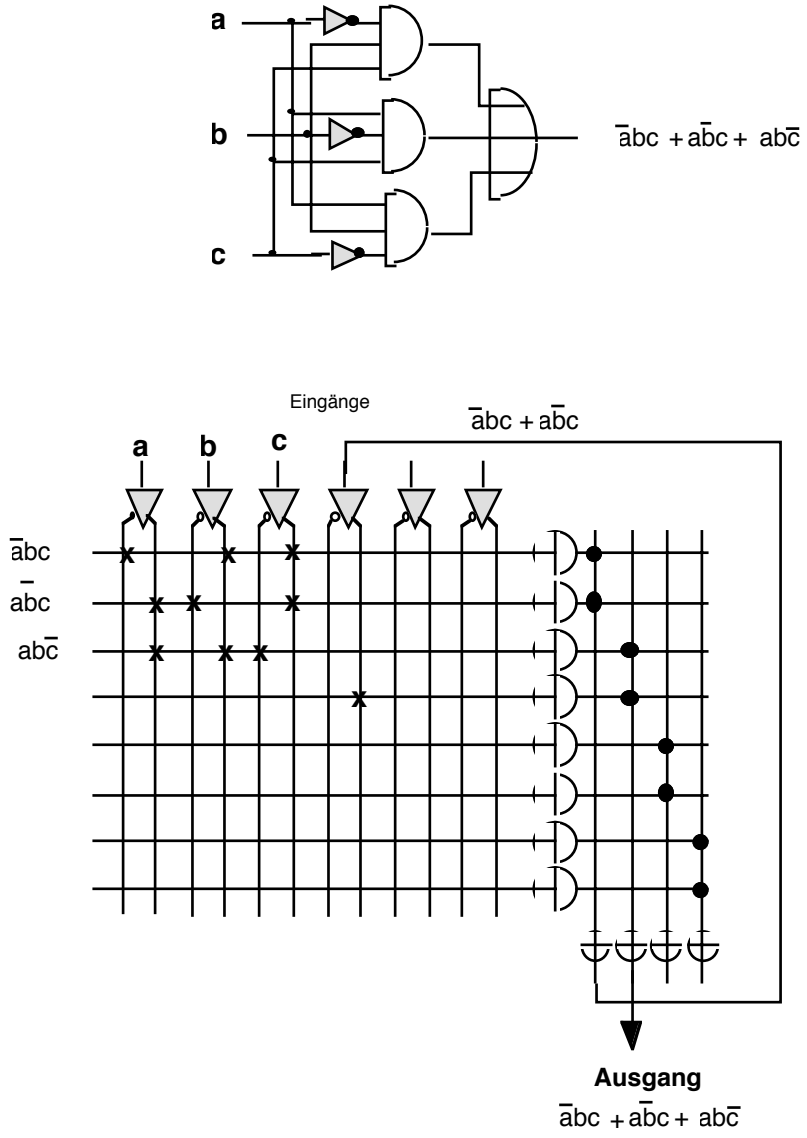


Abb 3.6 Realisierung eines kombinatorischen Schaltnetzes durch ein PAL

Da in dem angenommenen Typ eines PAL jeweils nur 2 UND-Gatter auf ein ODER-Gatter geführt werden, unser Schaltnetz aber 3 UND-Terme besitzt, müssen wir einen PAL-Ausgang auf einen PAL-Eingang zurückführen. Die mit "x" gekennzeichneten Kreuzungspunkte der UND-Matrix stellen Verbindungen dar. Alle anderen Sicherungen wurden zerstört.

Abb. 3.7 stellt die Struktur eines PALs und eines PROMs mit mehreren Eingängen gegenüber. Beide Komponenten haben in diesem Beispiel 8 Mehrfach-UND-Gatter. Die Anzahl der UND-Gatter in einem PAL bestimmt, wieviele einzelne Produkte die logische Gleichung, dargestellt als Summe von Produkten, haben kann. Die Anzahl der Eingänge bestimmt, wieviele Variablen insgesamt verarbeitet werden können. Da bei einem ROM jede Eingangsbelegung genau eines der acht möglichen Produkte darstellt (da die UND-Matrix ja ein fester 1-aus-2^n (n= Anzahl der Eingangsvariablen) Dekodierer ist), kann es bei 8 Mehrfach-UND-Gattern maximal 3 Eingänge geben. Braucht man z.B. 4 Variablen, um eine Funktion zu realsieren, müssen 2^4, d.h. 16 Mehrfach-UND-Gatter verfügbar sein. Will man, wie im PAL Beispiel, 6 Variablen verarbeiten, wären 64 UND-Gatter notwendig. Offensichtlich ist es dagegen bei einer PAL-Struktur eher möglich, viele Eingangsvariablen zu verarbeiten.

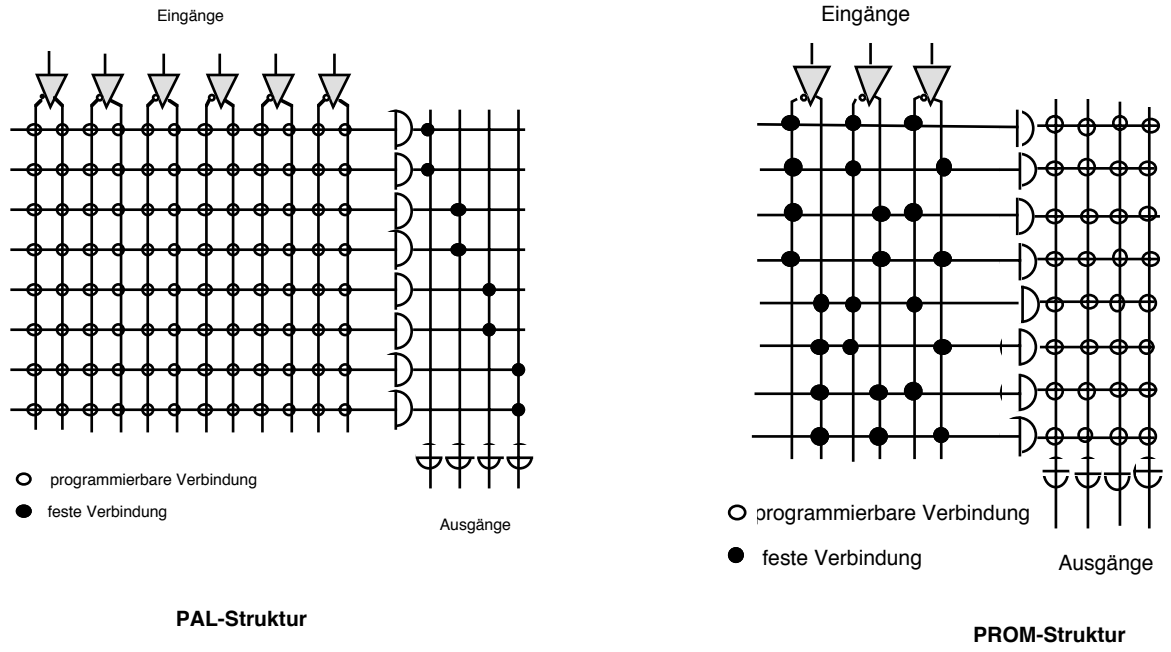


Abb. 3.7 Strukturen programmierbarer Komponenten

Allgemein ist es so, daß ein PAL immer dann geeignet ist, wenn eine relativ große Zahl von Variablen verarbeitet werden muß, aber die Funktion durch eine disjunktive Normalform mit relativ wenigen Produkttermen dargestellt werden kann. Das bedeutet, daß viele Eingangsbelegungen auf dieselbe Ausgangsbelegung abgebildet werden. Ein PROM dagegen

wird dann eingesetzt, wenn potentiell **jeder** Kombination der Eingangsvariablen eine individuelle Ausgangsbelegung zugeordnet werden soll. Ein Beispiel dafür ist die Umsetzung eines Codes, wobei jeder Eingangsbelegung genau eine Ausgangsbelegung zugeordnet werden soll, die sich von allen anderen Ausgangsbelegungen unterscheidet.

Programmierbare Komponenten, welche die Eigenschaften beider Ansätze vereinen, sind die sogenannten PLA (Programmable Logic Arrays). Sie besitzen sowohl eine programmierbare UND-Matrix als auch eine programmierbare ODER-Matrix. Allerdings ist ihre Programmierung schwierig und in den meisten Fällen wiegt der Vorteil der Flexibilität diesen Nachteil nicht auf.

3.2.2 Vom PAL zum Gate Array - Erweiterungen der programmierbaren Komponenten

PALs der bisher eingeführten Form können kombinatorische Schaltungen realisieren. Für sequentielle Schaltwerke müssen Flip-Flops hinzugefügt werden. Abb. 3.8 zeigt eine typische Konfiguration. Intern sind bei diesem PAL-Typ die Flip-Flop Ausgänge auf die UND-Matrix zurückgeführt, so daß man ohne äußere Beschaltung sequentielle Automaten in einem Chip realisieren kann.

Ein weiterer Schritt zu mehr Flexibilität sind sogenannte Makrozellen im Ausgang des PALs (z.B. bei dem Chip 22V10). Sie ermöglichen:
- die zusätzliche Invertierung von Ausgangssignalen,
- die Umgehung der Flip-Flops am Ausgang für rein kombinatorische Funktionen und
- die alternative Definition eines Eingangs oder Ausgangs auf demselben Pin des PALs.

Zusätzlich kann die Programmierung des 22V10 gelöscht und mehrfach neu durchgeführt werden. Solche Komponente werden daher auch als GAL (Generic Array Logic) bezeichnet.

Mit höherem Integrationsgrad entsteht die Notwendigkeit, eine stärkere Strukturierung und Partitionierung des Chips vorzunehmen. Der Chip wird dann in Zellen aufgeteilt, die ihrerseits die Funktionalität vollständiger PALs haben, die untereinander durch ein programmierbares, globales Verbindungsnetz verbunden sind.

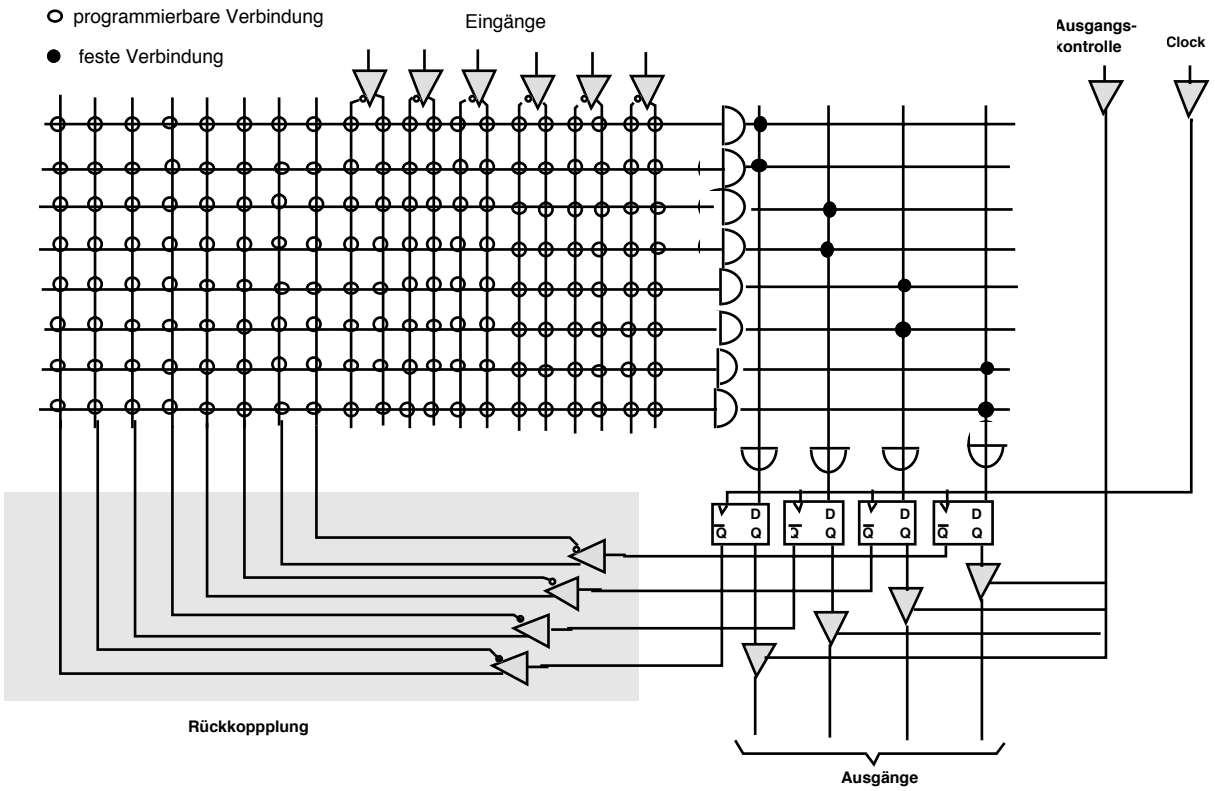


Abb. 3.8 PAL mit Flip-Flops

Die in Abb. 3.9 angegebene LCA-Familie (LCA : Logic Cell Array) von Xilinx ist ein Beispiel für diese Klasse programmierbarer Komponenten. Sie erreicht eine Komplexität von ca. 20000 Gattern und hat bis zu 240 Eingänge. Ähnlich sind die Programmierbaren Gate Arrays (PGA), das "High End" der vorgefertigten programmierbaren Logik. Der Unterschied zu LCAs ist eine noch höhere Flexibilität bei der Definition beliebiger logischer Funktionen und einem noch flexibleren Verbindungsnetzwerk.

Komponente	Eingänge	Ausgänge	Komplexität (Gatter)	Besonderheiten	Design Entry
PAL (22V10)	22	10	300-500	"High-End" PAL	PAL Assembler direkte Progr. VLSI-Entwurfswerkz.
PLD/LCA XILINX 400X	64-240	64-240	2000-20000	reprogrammierbar	VLSI-Entwurfswerkz.
FPGA ACT 12XX	82 - 140	82-140	2500-8000	"Antifuse"-progr.	VLSI-Entwurfswerkz.
			6250-20000 PLD/LCA äquivalente Gatter 70-210 TTL Äquivalente		

3.9 Komplexität programmierbarer Komponenten

Im folgenden werden die wichtigsten Begriffe und programmierbaren Komponenten (PLD = Programmable Logic Devices) zusammengestellt. Es muß allerdings betont werden, daß dem Erfindungsreichtum der Firmen, die solche Komponenten herstellen, bezüglich einer Namensgebung ihrer Produkte kaum Grenzen gesetzt sind:

- ***Maskenprogrammierbare Komponenten***: bestehen aus einer Matrix von vorgefertigten Zellen (z.B. Gattern oder Speicherzellen), die aber noch nicht miteinander verschaltet sind. Die Verbindung dieser Zellen erfolgt als Schritt im Produktionsprozeß durch eine sogenannte Maske, welche die Verbindungsstruktur festlegt.
- ***Anwenderprogrammierbare Komponenten*** (Field Programmable Devices) werden vollständig gefertigt und gekapselt. Die Programmierung ist nicht Teil des Fertigungsprozesses, sondern erfolgt vor Ort (im Feld) durch den Anwender mit einem entsprechenden Programmiergerät. Dabei wird ein Teil der Verbindungsstruktur zerstört, so daß nur die erwünschten Verbindungen bestehen bleiben.

- ***Programmable Logic Arrays (PLA)*** bestehen aus einer programmierbaren UND-Matrix, gefolgt von einer programmierbaren ODER-Matrix.
- ***Field Programmable Logic Arrays (FPLA)*** wie PLA, aber durch den Anwender programmierbar.
- ***Programmable Array Logic (PAL)*** bestehen aus einer programmierbaren UND-Matrix und einer festen ODER-Matrix.
- ***Generic Array Logic (GAL)*** sind elektrisch löschbare PLDs, die auch manchmal E^2PLD genannt werden.
- ***Field Programmable Gate Arrays (FPGA)*** bestehen aus einer programmierbaren UND-Matrix, bei der zusätzlich die Eingangs- und Ausgangspolarität programmiert werden kann, d.h. ob sie in invertierter oder nicht invertierter Form vorliegt. Dadurch kann ein Gatter der UND-Matrix jedes Grundgatter UND, ODER, NAND oder NOR realisieren und hat daher eine hohe Flexibilität.
- ***Field Programmable Logic Sequencer (FPLS)*** ist eine Erweiterung der FPLAs durch Register.

3.2.4 Programmierung

Die Programmierung umfaßt bei PALs die Generierung der entsprechenden Belegungen der programmierbaren Diodenmatrizen, d.h. die Erstellung der sogenannten "Fuse Map" (FM), und den anschließenden physikalischen Vorgang des "Brennens", d.h. der Zerstörung der

gewünschten Dioden in der Matrix. Bei der Generierung der FM kann man von unterschiedlichen Abstraktions- und Beschreibungsebenen ausgehen. Zum Beispiel kann man ein Schaltnetz als eine Menge boolscher Gleichungen spezifizieren, wie wir es bisher getan haben. Das Entwurfswerkzeug erzeugt dann aus dieser Beschreibung die Diodenmatrix. Bei anderen programmierbaren Komponenten werden Speicher zur Definition der Verbindungsstruktur und der logischen Funktionen eingesetzt.

Als weiteres Beispiel der Umsetzung einer booleschen Gleichung in eine PAL-FM soll der Binärsequenzdecodierer dienen, den wir im Teil "Digitale Logik" kennengelernt haben. Abb. 3.10 zeigt das Schaltbild des als Moore-Automat realisierten Binärsequenzdekodierers.

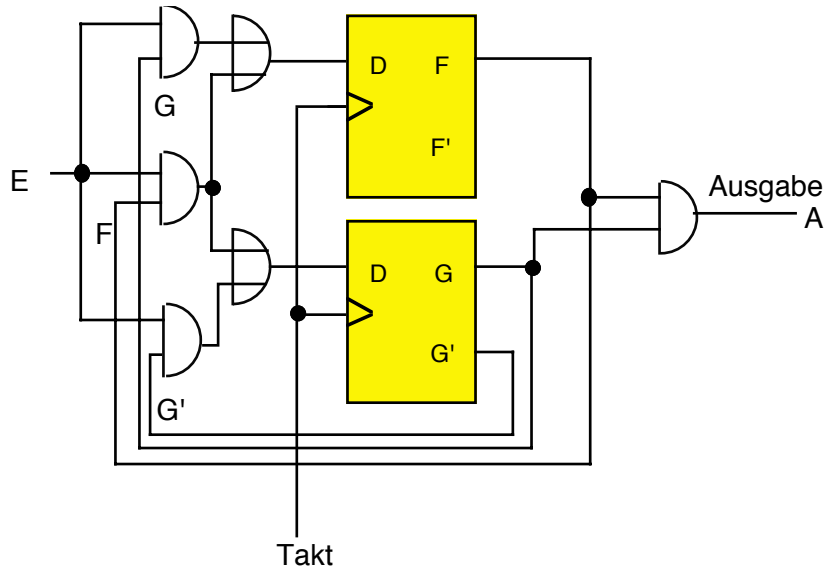


Abb. 3.10 Schaltbild des Binärsequenzdekodierers

Aus dem Eingang E und den internen Variablen F, G, und G‘ muß ein neuer Zustand generiert werden, der eine Belegung des Ausgangs A erzeugt. Die entsprechenden boolschen Gleichungen lauten:

DF = EG + EF
DG = EF + EG‘
A = FG

Als Grundlage für die Realisierung wird das PAL aus Abb. 3.8 verwendet. Für den Binärsequenzdecodierer werden zwei Flip-Flops benötigt und ein direkter Ausgang, der nicht über ein Flip-Flop geschaltet wird. In realen PALs mit Flip-Flops sind beide Arten von

Ausgängen vorhanden. Wir modifizieren das PAL aus Abb. 3.8 so, daß zwei Ausgänge über Flip-Flops und zwei direkte Ausgänge verfügbar sind. Abb. 3.11 zeigt die Struktur des programmierten PALs. Es wird ein Eingang und ein Ausgang benötigt. Die Ausgänge der Flip-Flops F, G und G‘ sind intern auf die Diodenmatrix rückgekoppelt.

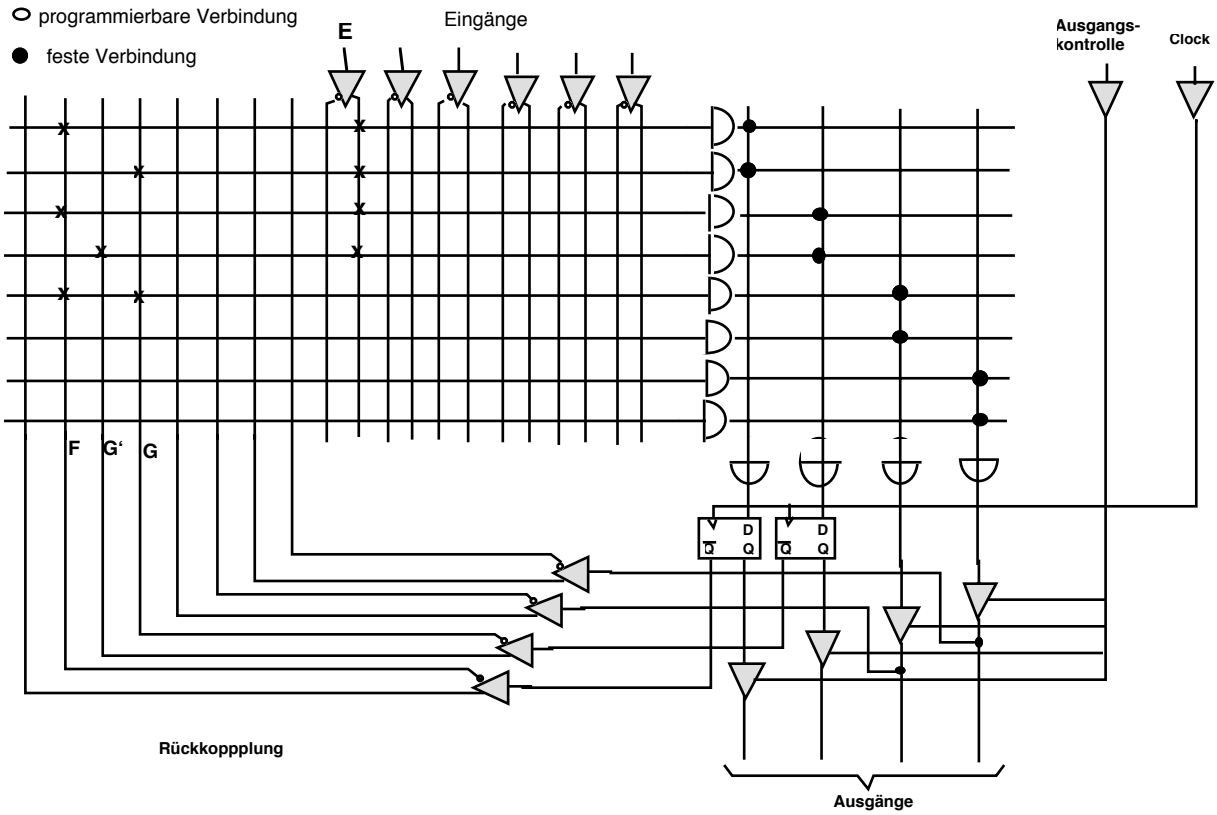


Abb. 3.11 Realisierung des Binärsequenzdekodierers mit einem PAL

4 Entwurf eines einfachen Prozessors

Ziel dieses Kapitels ist der Entwurf eines sehr einfachen Prozessors. An diesem Beispiel soll die Arbeitsweise eines Prozessors auf der Logikebene nachvollziehbar sein.

4.1 Einführung

Der Prozessor oder synonym die CPU ist die Einheit eines Rechners, die Daten nach einem vom Anwender spezifizierten Programm manipuliert. Das Programm wird als eine Folge von Anweisungen oder Befehlen für die CPU formuliert. Die CPU besteht daher:
1.) aus einer Komponente, welche die Befehle liest, interpretiert und die korrekte Folge der Befehlsabarbeitung einhält, kurz, einer ***Kontrolleinheit,*** die den Ablauf des Programms steuert. Die Kontrolleinheit wird auch als ***Steuerwerk*** bezeichnet.
2.) aus einer Komponente, welche die Operationen auf den Daten, die durch die Befehle spezifiziert werden, durchführt. Diese Einheit wird als ***Datenpfad*** bezeichnet. Abb. 4.1 zeigt diese beiden Grundkomponenten der CPU. Der Datenpfad wird auch als ***Rechenwerk*** bezeichnet.

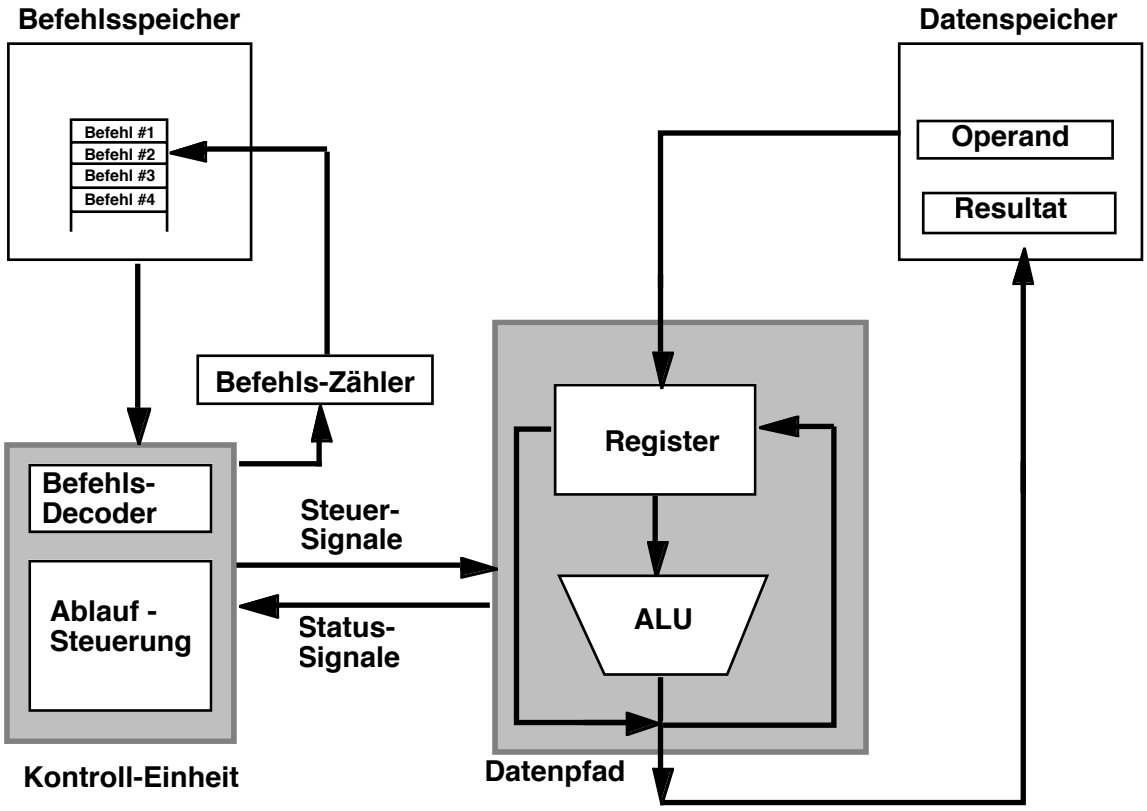


Abb. 4.1 Kontrolleinheit und Datenpfad einer CPU

Die Kontrolleinheit besteht im wesentlichen aus einem Befehlsdecoder und einer Ablaufsteuerung. Befehle werden aus dem Befehlsspeicher gelesen und dekodiert. Dabei gibt der Befehlszähler (auch als Programmzähler bezeichnet) an, welcher Befehle im nächsten Schritt bearbeitet werden soll. Die Ablaufsteuerung erzeugt, entsprechend dem auszuführenden Befehl, die Steuer- oder Kontrollsignale, um Daten zu lesen, im Datenpfad zu bearbeiten und die Resultate zu speichern. Bei der Bearbeitung der Daten werden vom Datenpfad Statussignale generiert, die wiederum den Ablauf des Programms beeinflussen können, wie z.B. eine Programmverzweigung, die abhängig davon ist, ob das Resultat einer Berechnung größer oder kleiner einem bestimmten Wert ist. Der Datenpfad besteht aus einer ALU, die im einfachsten Falle arithmetische und logische Grundoperationen ausführen kann und einem Registersatz, in dem Daten bei der Verarbeitung zwischengespeichert werden. Eine spezielle Ausprägung von Kontrolleinheit und Datenpfad wird im Teil "Digitale Logik" bereits mit der Entwicklung einer ALU, eines Registersatzes und einer mikroprogrammierbaren Steuereinheit eingeführt. Was hier neu hinzukommt, ist die Steuerung dieser Komponenten durch ein Anwendungsprogramm. Anwendungsprogramme werden meist in einer Hochsprache geschrieben.

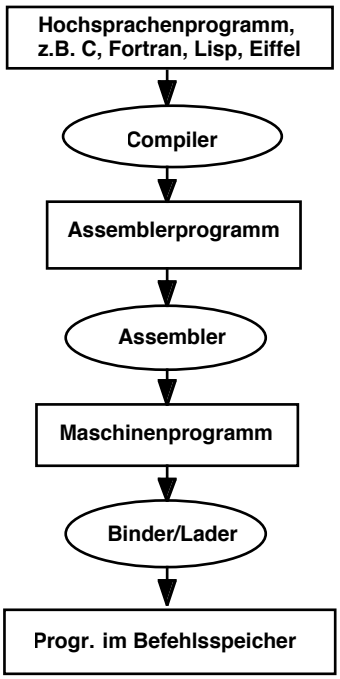


Abb. 4.2 Von der Hochsprache zum interpretierbaren Maschinenprogramm

Eine CPU benötigt jedoch die Befehle in einer Form, in der sie für die Kontrolleinheit interpretierbar sind. Dies ist in der Regel nicht direkt die Notation der Hochsprache[1]. Das

[1] Es gibt Spezial-CPUs, die auf eine bestimmte Sprache optimiert sind. Allerdings ist dies allgemein nicht unbedingt wünschenswert, da die CPU universell für viele Anwendungen einsetzbar sein soll. Die Beschränkung auf eine bestimmte Hochsprache stellt in diesem Fall eine Einschränkung dar.

Anwenderprogramm der Hochsprache muß deshalb übersetzt werden in eine Form, welche die Kontrolleinheit "versteht". Abb. 4.2 zeigt die Übersetzungsvorgänge bis das Programm in geeigneter Form im Befehlsspeicher steht. Die von der CPU direkt interpretierbaren Befehle werden als ***Maschinenbefehle*** bezeichnet. Die Menge der Maschinenbefehle bildet den (Maschinen-) ***Befehlssatz*** der CPU.

Heutige CPUs beruhen auf dem Prinzip des ***Stored Program Computer***. Dieser Ansatz beruht auf zwei Annahmen, die letztlich zwei Seiten derselben Münze sind:
1.) Die Befehle der CPU werden als Binärzahlen codiert.
2.) Die Befehlsfolgen (Programme) werden in einem Schreib-/Lesespeicher abgelegt.

Punkt 1 besagt, daß Befehle wie Daten codiert werden und nicht, wie in Rechnern der allerersten Generation (etwa im ENIAC), eine Folge von Befehlen auf einem Steckbrett mit Kabeln und Schaltern festgelegt wird. Dadurch ist die Repräsentation von Befehlen grundsätzlich von Daten nicht zu unterscheiden. Welche Binärzahlen als Befehle und welche als Daten zu interpretieren sind, kann nicht aus der Repräsentation eines Programmes selbst geschlossen werden. Es muß außerhalb der Repräsentation eines Programmes genau bekannt sein, welche Binärzahlen Befehle und welche Daten darstellen.

Punkt 2 ist eine Konsequenz aus Punkt 1. Wenn Programme schon wie Daten repräsentiert werden, liegt es nahe, sie auch im selben Medium, nämlich einem Schreib-/Lesespeicher, abzulegen. Die Eigenschaft, daß man beliebige Programme in den Speicher laden kann, ist die Grundlage für die Universalität des Rechners.

In Abb. 4.1 sind der Befehlsspeicher und der Datenspeicher getrennt. Tatsächlich haben aber die heutigen Rechner einen gemeinsamen Schreib-/Lesespeicher für Befehle und Daten.

4.2 Ein einfacher Befehlssatz

With any computer, on-line or not, the merits of different possible instructions can be argued about endlessly.

James Martin, 1967

Wie in der Einführung bereits erwähnt, stellt der Maschinenbefehlssatz die Schnittstelle zwischen Hardware und Software dar und definiert die Rechnerarchitektur. Er definiert sowohl die Operationen, die ausgeführt werden können, als auch die verwendbaren Register des Datenpfads und der Steuereinheit sowie die Arten der Adressierung. Der Maschinenbefehlssatz ist die Zielsprache bei der Compilierung einer Hochsprache in die direkt vom Rechner interpretierbaren Anweisungen.

Zunächst sollen die verschiedenen Arten von Befehlen betrachtet werden. Sie können grob in folgende Klassen unterteilt werden:

- Befehle, die sich auf den Datenpfad beziehen und Funktionen definieren, die zur Manipulation von Daten zur Verfügung stehen. Beispiele für diese Befehlsklasse sind in Tab. 4.1 angegeben.
- Befehle, die zur Steuerung des Programmflusses notwendig sind. Dazu gehören bedingte und unbedingte Programmverzweigungen. Tab. 4.2 gibt dazu Beispiele an.
- Speicher-Befehle (Load, Store), Ein/Ausgabe - Befehle

Datenpfadbezogene Befehle:	
• Arithmetische Befehle:	Beisp.: Add, Sub, Mult, Div, 2er-Komplement
• Logische Befehle:	Beisp.: AND, OR, XOR, Komplement
• Shift- Befehle:	Beisp.: Shift, Rotate
• Vergleichsbefehle:	Beisp.: CMP (A = B, A ≠ B, A < B, A > B, A < B, A > B)
• Testbefehle:	Beisp.: A = "all ones", A = 0, A = 0, Overflow, Underflow
• Transferbefehle:	Beisp.: Move Reg a, Reg b

Tab. 4.1 Datenpfadbezogenen Befehle

Kontrollflußbezogene Befehle: Programmverzweigungen (Sprungbefehle)	
• Unbedingter Sprung	JMP, BRA
• Bedingter Sprung	JMP (Bedingung), BRA (Bedingung),
• Unterprogrammsprung	JSR, RTS

Tab. 4.2 Befehle zur Steuerung des Kontrollflusses

Als nächstes wollen wir uns überlegen, in welcher Form ein Maschinenbefehl dargestellt werden kann, d.h. wir wollen das sogenannte ***Maschinenbefehlsformat*** festlegen. Gehen wir zunächst von der Klasse der arithmetischen und logischen Operationen aus. Wir müssen im Maschinenbefehl eine Operation und mindestens ein Argument oder einen Operanden spezifizieren. Abb. 4.3 zeigt den grundlegenden Aufbau eines Maschinenbefehls. Der Operationscode ist eine Binärzahl, die eine Codierung für einen Maschinenbefehl darstellt. Diese "dichte" Codierung stellt den Befehlssatz (im Gegensatz z.B. zu einer textuellen Namensgebung) mit möglichst wenig Bitstellen dar.

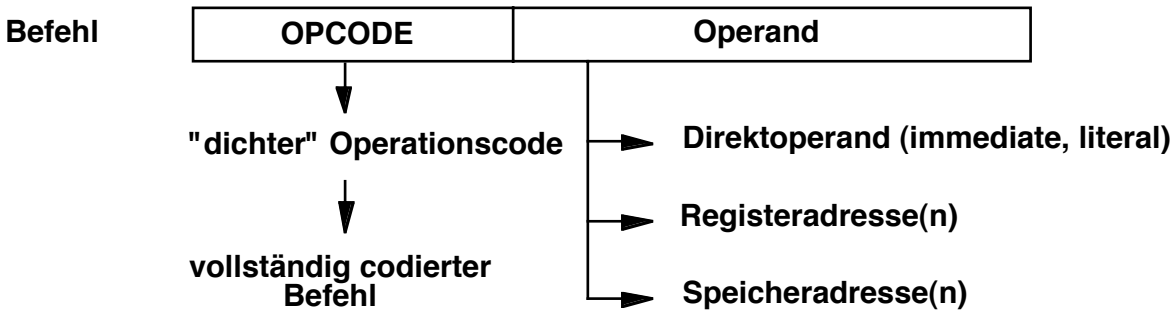


Abb. 4.3 Format eines Maschinenbefehls

Zur Spezifikation der Operanden im Operandenfeld gibt es verschiedene Möglichkeiten. Nehmen wir zum Beispiel den Additionsbefehl. Eine normale Addition benötigt zwei Operanden. Diese Operanden könnten z.B. beide direkt im Operandenfeld angegeben sein. Man spricht dann auch von ***Direktoperanden*** (auch als immediate operand oder als literal bezeichnet). Da üblicherweise aber mehrere arithmetische Operationen hintereinander ausgeführt werden, ist es wahrscheinlich, daß einer oder beide Operanden Ergebnisse vorheriger Operationen sind und z.B. bereits in Registern oder im Speicher stehen. Dann müßten zur Spezifikation der Operanden zwei Register- oder Speicheradressen angegeben werden. Eine andere Möglichkeit besteht darin, daß das Ergebnis einer arithmetisch/logischen Operation immer in einem bestimmten Register steht. Dieses Register muß deshalb nicht extra spezifiziert werden, so daß nur der zweite Operand angegeben werden muß.

Die Zielkonflikte bei der Festlegung eines Befehlsformates sind vielfältig und die entsprechende Entwurfsentscheidung wirkt sich auf die Leistungsfähigkeit einer CPU aus. Wir werden in späteren Kapiteln immer wieder darauf zurückkommen und die Alternativen diskutieren. Für unseren Modellrechner nehmen wir an, daß ein Operand bei einer zweistelligen Operation immer in einem festen Register steht und nur der zweite Operand explizit im Befehl spezifiziert werden muß. Das Befehlsformat entspricht dann genau dem

Format in Abb. 4.3. Um eine möglichst große Einfachheit zu erreichen, kann der Operand auch nur auf eine einzige Weise spezifiziert werden: als Adresse im (Daten-)Speicher.

Um einen konkreten Maschinenbefehlssatz festlegen zu können, müssen wir aber noch weitere Randbedingungen unseres Prozessors betrachten. Ein entscheidender Parameter ist die ***Wortbreite der Speicherschnittstelle***. Die Wortbreite der Speicherschnittstelle bestimmt, wieviel Information in einem Speicherzugriff zwischen Prozessor und Speicher transferiert werden kann. Natürlich wäre der erste Ansatz, diese Schnittstelle möglichst breit zu machen, um einen möglichst großen Informationsfluß zwischen Prozessor und Speicher zu ermöglichen. Allerdings haben wir es bei dieser Entscheidung mit einer der schwergewichtigen Kosten-/Nutzen-Entscheidungen in der Rechnerarchitektur zu tun. Zwischen der Wortbreite des Speicher und der Breite der Verarbeitungseinheiten des Prozessors besteht nämlich eine enge Wechselwirkung und sie sind daher so aufeinander abgestimmt, daß die Verarbeitungseinheiten des Prozessors genau ein Speicherwort parallel verarbeiten können. Da die Verwendung von Verarbeitungseinheiten, die große Wortbreiten parallel verarbeiten können, teuer und je nach Anwendungsbereich nur bis zu einer gewissen Größe auch sinnvoll ist, liegen die Wortbreiten der Prozessor/Speicherschnittstelle meist zwischen 8-Bit (bei einfachen Microcontrollern) und 64 Bit (bei den neueren hochintegrierten CPUs).

Da Befehle und Daten in ein- und demselben Schreib/Lesespeicher stehen, muß die Wortbreite der Speicherschnittstelle auch für das Befehlswort berücksichtigt werden. Für unseren Modellrechner wählen wir eine Wortbreite von 16 Bit. Nun muß die Aufteilung dieses Wortes in ein Feld für den Operationscode und ein Feld zur Adressierung des Operanden aufgeteilt werden. Mit 16 Bit für das Operandenfeld können insgesamt 64 k Worte adressiert werden. Jedes Bit, das wir für die Codierung der Befehle brauchen, halbiert diesen Wert. Als einen Kompromiß zwischen der Größe des Speicheradreßraums und einem Befehlssatz, der die wesentlichen arithmetisch/logischen Operationen enthält, nehmen wir eine Aufteilung vor in ein:

4-Bit Feld für 16 Operationscodes und
12-Bit-Feld zur Adressierung von 4 k Speicher.

Wir könnten im Rahmen unseres 16-Bit-Wortes auch andere Aufteilungen vornehmen, aber es sollte klarwerden, daß hier eine Entwurfsentscheidung getroffen werden muß, die immer auf einen Kompromiß hinausläuft.

Tab. 4.3 zeigt die Operationen unseres Modellrechners und beschreibt deren Wirkung.

OPC	Mnem.	Operand	Beschreibung
0000	HLT		HLT hält den Computer an. Kann auch manuell eingegeben werden. Nach HLT kann der Rechner nur manuell gestartet werden. Fortsetzung beim nächsten Befehl.
0001	JMA	addr	(Jump on Minus) Bedingter Sprung. Wenn das Ergebnis einer Berechnung negativ ist. Die Adresse "addr" wird in den Befehlszähler geladen. Der nächste Befehl wird von "addr" genommen.
0010	JMP	addr	Unbedingter Sprung. Die Adresse "addr" wird in den Befehlszähler geladen. Der nächste Befehl wird von "addr" genommen.
0011	JSR	addr	Unterprogrammsprung. Die Adresse, die im Befehlszähler enthalten ist, wird ins Register A geladen. Die Adresse "addr" wird in den Befehlszähler geladen. Der nächste Befehl wird von "addr" geladen.
0100	CSA		(Copy Switch Register to A) Der Zustand der Schalter wird in das Register A geladen.
0101	RAL		(Rotate A Left) Zyklischer Links-Shift. Der Inhalt von Register A wird um 1 Stelle nach links rotiert. Ringshift : Bit $A_0 \leftarrow$ Bit A_{15}
0110	INP		INPUT
0111	OUT		OUTPUT
1000	NOT		Komplementbildung des Inhalts von Register A.
1001	LDA	addr	Laden des Registers A von Speicheradresse addr.
1010	STA	addr	Speichern des Registerinhalts auf Speicheradresse addr .
1011	ADD	addr	Inhalt des Speicherwortes mit Adresse adr wird auf den Inhalt des Registers A addiert. Der ursprünglich Inhalt von A wird mit dem Ergebnis überschrieben.
1100	XOR	addr	Inhalt des Speicherwortes mit Adresse adr wird mit dem Inhalt des Registers A durch excl. ODER verknüpft. Der ursprüngliche Inhalt von A wird mit dem Ergebnis überschrieben.
1101	AND	addr	Inhalt des Speicherwortes mit Adresse adr wird mit dem Inhalt des Registers a durch log. UND verknüpft. Der ursprünglich Inhalt von A wird mit dem Ergebnis überschrieben.
1110	IOR	addr	Inhalt des Speicherwortes mit Adresse adr wird mit dem Inhalt des Registers A durch log. ODER verknüpft. Der ursprünglich Inhalt von A wird mit dem Ergebnis überschrieben.
1111	NOP		(No Operation) Der Befehlszähler wird um 1 erhöht.

Tab. 4.3 Befehlssatz des Modellrechners

Die Spalten der Tab. 4.3 beinhalten:

- die Codierung des Befehls, d.h. die zugeordnete Binärzahl.
- den (textuellen) Namen des Befehls, auch mnemotechnische Bezeichnung genannt.
- den zugehörigen Operanden, der hier immer durch eine Adresse spezifiziert wird.
- die Beschreibung des jeweiligen Befehls.

Wir haben bisher das Befehlsformat festgelegt. Da Befehlsworte und Datenworte im selben Speicher stehen und daher die gleiche Länge haben, ist die Festlegung des Datenformates in einfacher Weise als 16-Bit-Wort gegeben, wobei die höchstwertige Stelle bei arithmetischen Operationen als Vorzeichen interpretiert wird. Der darstellbare Zahlenbereich reicht daher von -2^{15} bis $2^{15}-1$. Abb. 4.4 faßt Befehls- und Datenformat des Modellrechners zusammen.

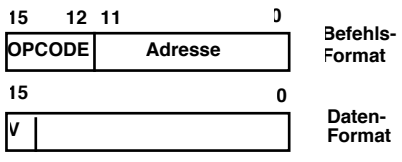


Abb. 4.4 Befehls- und Datenformat des Modellrechners

Zur Manipulation der Daten und zur Steuerung des Programmes benutzen die Befehle Register. Folgende Register können direkt durch die Befehle des Rechners gelesen bzw. manipuliert werden:

1.) Der ***Akkumulator*** (Register A). In dem Modellrechner ist der Akkumulator das einzige *allgemeine* Register. Alle arithmetischen und logischen Befehle arbeiten auf diesem Register.
2.) Das ***Schalterregister*** (Switch Register: SWR) besteht aus 16 Schaltern, die z.B. an der Frontplatte eines Rechners angebracht sind. Sie können manuell gesetzt und vom Rechner abgefragt werden[2].
3.) Der ***Programmzähler*** (Befehlszähler) wird beim Unterprogrammsprung(JSR, Jump Subroutine) in den Akkumulator geladen, durch den Befehl RTS (Return from Subroutine) wird der Inhalt des Akkumulators in den Programmzähler geladen.
4.) Das ***Halt-Flip-Flop*** wird durch den Befehl HLT gesetzt. Es kann nur manuell zurückgesetzt werden.

Die Register, die durch den Befehlssatz gelesen oder verändert werden können, bilden zusammen mit den Formaten für Daten und Befehle das ***Programmiermodell*** des Rechners. Das Programmiermodell beschreibt den Rechner auf der für den Programmierer relevanten Abstraktionsebene. Abb. 4.5 zeigt das einfache Programmiermodell des Modellrechners.

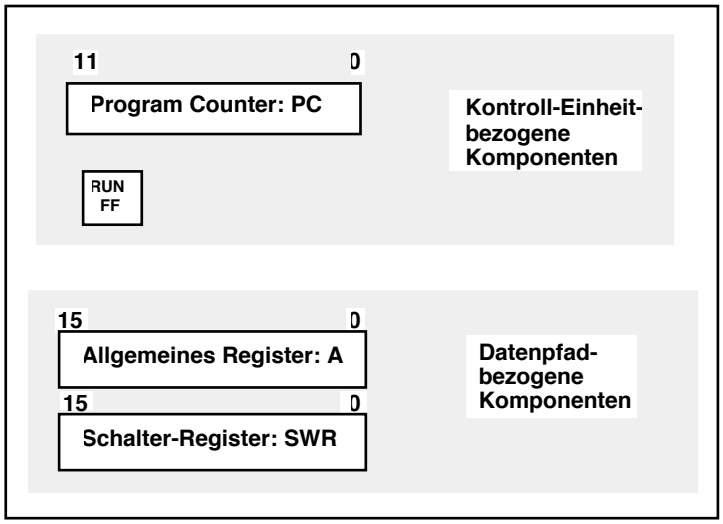


Abb. 4.5 Register im Programmiermodell des Modellrechners

[2] In heutigen Rechern gibt es kein SWR mehr. In Verbindung mit Lämpchen oder Leuchtdioden, welche den Zustand ausgewählter Register anzeigten, diente es in früheren Rechnergenerationen zum einen der Initialisierung des Rechners, zum anderen wurde es zu Diagnose- und Debuggingzwecken benutzt. Heute werden Rechner aus einem ROM initialisiert. Diagnose und Debugging werden durch komfortablere, programmgestützte Mechanismen unterstützt. Diese setzen allerdings mindestens voraus, daß gewisse Treiberprogramme zur Kommunikation mit dem Rechner in einem ROM installiert sind.

Ein kleines Beispielprogramm, das eine variable Anzahl zyklischer Rechtsshifts ausführt, ist in Abb. 4.6 a gezeigt. Abb. 4.6 b zeigt die Organisation der zugehörigen Daten im Speicher.

Sp.- Adr	OPC	OP-Adr.	Sprung-Marke	Mnemonic		Kommentar
00010000	1001	10000001		LDA	AR	Lade Anzahl der Rechts-Shifts in Register a
00010001	1000	----------		NOT		Komplementieren von a
00010010	1011	10000010		ADD	"1"	2-Komplement
00010011	1011	10000100		ADD	"16"	Berechne : 16 + (-AR)
00010100	1010	10000011		STA	TMP	Speichere Anzahl der Links-Shifts
00010101	1001	10000000	NEXT:	LDA	D	Lade Datum
00010110	0101	----------		RAL		Links-Shift
00010111	1010	10000000		STA	D	Speichere Datum
00011000	1001	10000010		LDA	"1"	Lade Konstante "1"
00011001	1000	----------		NOT		Komplementieren von a
00011010	1011	10000010		ADD	"1"	2-Komplement
00011011	1011	10000011		ADD	TMP	Decrementiere Anzahl der L-Shifts
00011100	1010	10000011		STA	TMP	Speichere verbleibende Anzahl der L-Shifts
00011101	0001	00011111		JMA	DONE	Bed. Sprung, wenn alle L-Shifts ausgeführt wurden
00011110	0010	00010101		JMP	NEXT	Unbed. Sprung zum Anfang der Schleife
00011111	0000	----------	DONE:	HLT		

Abb. 4.6 a. Programm zur Ausführung zyklischer Rechtsshifts

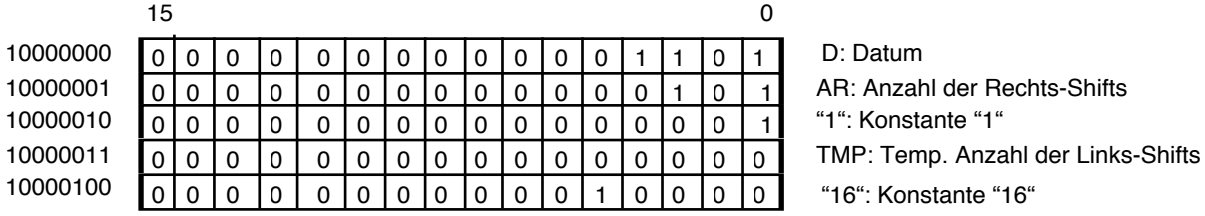


Abb. 4.6 b. Organisation der Daten im Speicher

Da wir als Befehl nur den zyklischen Linksshift um 1 Stelle zur Vefügung haben, müssen wir den zyklischen Rechtsshift um eine beliebige Anzahl ($<2^{15}-1$) von Stellen durch ein Programm simulieren. Im Modellrechner kann in einem Befehl kein Direktoperand angegeben werden. Deshalb müssen wir alle Konstanten im Speicher ablegen. Da nur ein einziges Register im Prozessor zur Vefügung steht, müssen wir auch alle Variablen im Speicher ablegen. Abb. 4.6.b. zeigt die Organisation dieser Daten im Speicher.

In Abb. 4.6 a ist in der ersten Spalte die Speicheradresse angegeben. Das Programm belegt 15 aufeinanderfolgende Speicherplätze. Die nächste Spalte gibt die binäre Repräsentation des Programms an. Die mnemotechnische Darstellung und ein Kommentarfeld sind in den folgenden Spalten dargestellt.

Im ersten Teil des Programms (Zeile 1-5) wird die Anzahl der notwendigen Linksshifts berechnet. Dann folgt eine Schleife (Zeile 6-13), in der die berechnete Anzahl von Linksshifts ausgeführt wird.

Ein weiteres Programmierbeispiel (Abb. 4.7 a und 4.7 b) zeigt die Unterprogrammtechnik unseres einfachen Prozessors. Wir nehmen an, daß wir z.B. das Programmfragment, das den zyklischen Rechts-Shift ausführt, als Unterprogramm benutzen wollen.

```
Sp.- Adr     OPC  OP-Adr.   Sprung-   Mnemo.          Kommentar
                            Marke
-------------------------------------------------------------------------
11010000     ------------              ---------       ------------------------
11010001     0011 01010000             JSR   RAR       Unterprogr.-Sprung zu RAR
11010010
                                       CONTINUE
                  :
                  :

Unterprogramm
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
01010000     1010 11100000   RAR:      STA   TMP1      Speichern der Rücksprungadresse
01010001          :
                  :                    AUSFÜHRUNG DES UNTERPROGRAMMS
                  :
01xxxx00     1001 11100000             LDA   TMP1      Lade Rücksprungadresse
01xxxx01     1110 11100001             IOR   "0010"    Generiere Sprungbefehl
01xxxx10     1010 01000011             STA   adr+1
01xxxx11     0010 11010010             JMP   RETURN    Rücksprung
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

Abb. 4.7 a Unterprogrammtechnik im Modellrechner

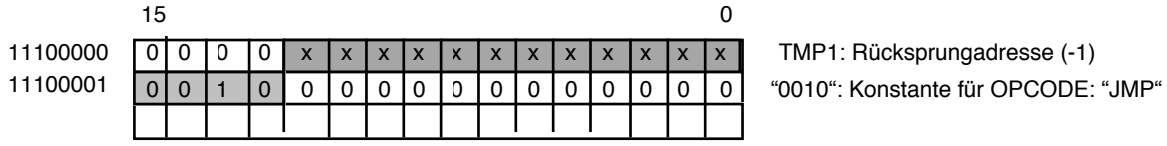


Abb. 4.7 b Organisation der Daten im Speicher

Beachtenswert sind drei Eigenschaften:

1.) Die Rücksprungadresse muß explizit gerettet werden, da der Unterprogrammsprung lediglich den aktuellen Inhalt des Programmzählers (aktuelle Adresse +1) in den Akkumulator A transferiert.

2. Der Prozessor stellt zwar einen Unterprogrammsprung JSR zur Verfügung, aber keinen expliziten Rücksprung (meist als RTS: Return from Subroutine bezeichnet).

3.) Der Befehl zum Rücksprung wird durch sogenannten ***selbstmodifizierenden Code*** realisiert. Da wir nur absolute Adressen im Befehl angeben können, müssen wir den Rücksprungbefehl dadurch generieren, daß wir den Operationscode des Sprungbefehls mit der gespeicherten Rücksprungadresse durch IOR kombinieren und den so synthetisierten Befehl an der Stelle im Programm wieder abspeichern, an der der Rücksprung erfolgen soll. Hier zeigt sich, welche Möglichkeiten sich ergeben, wenn man Befehle wie Daten repräsentiert.

4.3 Vom Programmiermodell zum arbeitsfähigen Prozessor - Die interne Struktur

Die Abstraktionsebene des Programmiermodells brauchen wir, wenn wir den Rechner benutzen wollen. Wenn wir einen Rechner konstruieren wollen, müssen wir auf eine tiefere Ebene gehen. Wir werden die Funktionsweise des Modellrechners auf der logischen Ebene kennenlernen. Dazu werden wir uns die Speicherschnittstelle, die Kontrolleinheit und den Datenpfad genauer ansehen. Abb. 4.8 zeigt diese Komponenten unseres Modellrechners.

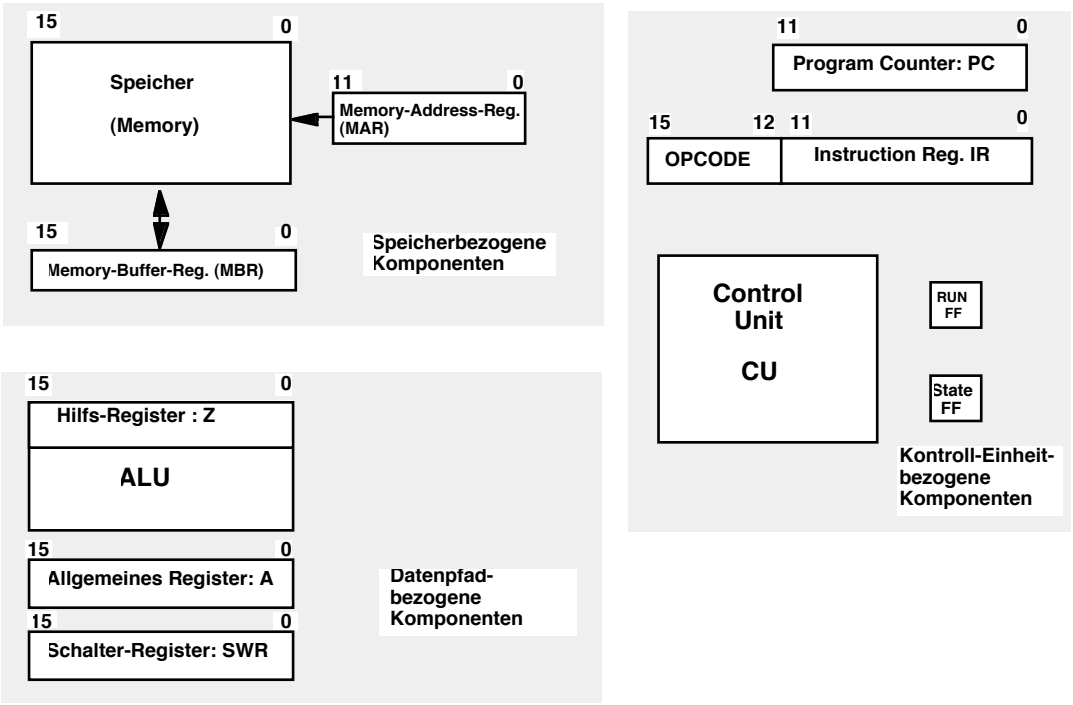


Abb. 4.8 Komponenten des Modellrechners

4.3.1 Die Speicherschnittstelle

In einem Speicherzyklus muß der Prozessor zunächst eine Adresse liefern, die während des gesamten Speicherzyklus anliegen muß. In einem Schreibzyklus muß zusätzlich auch das Wort, das in den Speicher geschrieben werden soll, während des Speicherzyklus anliegen. Bei einem Lesezyklus steht das gewünschte Wort erst mit einer gewissen Verzögerung an der Schnittstelle zur Verfügung. Da der Speicher sowohl zum Schreiben als auch zum Auslesen eines Wortes länger braucht als die Zeit, in der der Prozessor eine elementare Operation ausführen kann[3], sind zwei Pufferregister vorgesehen:

1.) Das Speicher-Adreß-Register (MAR : Memory Address Register), in das die Adresse zu Beginn des Speicherzyklus geschrieben wird.

[3] Wir werden später sehen, daß ein Befehls aus mehreren elementaren Operationen des Prozessors besteht.

2.) Das Speicher-Puffer-Register (MBR : Memory Buffer Register). Bei einer Schreiboperation legt der Prozessor ein Datenwort hier ab, so daß es durch den (langsamen) Schreibvorgang im Speicher unter der Adresse abgespeichert wird, die im MAR spezifiziert ist. Beim Lesen stößt der Prozessor den Lesevorgang an und kann später das adressierte Wort aus dem MBR auslesen.

Durch MBR und MAR sind Prozessor und Speicher bezüglich ihrer Zykluszeiten weitgehend entkoppelt.

4.3.2 Die Komponenten des Datenpfads

Der Datenpfad besteht aus der ALU, dem allgemeinen Register A (Akkumulator), einem Hilfsregister Z und dem bereits erwähnten Schalterregister SWR. Die zweistelligen arithmetischen und logischen Befehle haben alle die Form: < OPCODE, addr >. Sie setzen voraus, daß der eine Operand in A steht, der zweite Operand muß aus dem Speicher von Adresse "addr" gelesen wird. Nach unserer Betrachtung in 4.3.1 können wir davon ausgehen, daß dieser Operand im MBR zur Verfügung steht. Da die ALU rein kombinatorisch aufgebaut werden soll, müssen beide Operanden während der Verarbeitungszeit an den Eingängen anliegen. Das Ergebnis der Operation soll in A verfügbar sein. Damit das Ergebnis der Operation nicht einen der Operanden in A überschreibt, ist das Hilfsregister Z vorgesehen. Während der Befehlsausführung wird der Operand aus A nach Z transferiert, damit das Ergebnis in A gespeichert werden kann.

Das SWR wurde in der Diskussion zu Abb. 4.5 erklärt.

4.3.2 Die Komponenten der Kontrolleinheit

Die Kontrolleinheit besteht aus:
1.) dem Programmzähler,
2.) dem Befehls- oder Instruktionsregister,
3.) dem RUN/HLT Flip-Flop,
4.) dem State-Flip-Flop,
5.) dem Automaten, der die der sequentiellen Kontrolle realisiert.

In der Phase des Prozessors, in der eine neue Instruktion gelesen wird, enthält der ***Programmzähler*** die Adresse dieser Instruktion. Ist die Instruktion gelesen, wird der Programmzähler auf die Adresse der Instruktion gesetzt, die als nächste ausgeführt werden soll. Über den Programmzähler wird also die Sequenzierung der Instruktionen bei der Abarbeitung eines Programms gesteuert.

Das ***Instruktionsregister*** speichert die Instruktion während ihrer gesamten Abarbeitung. Das ist notwendig, damit die Kontrolleinheit die Felder der Instruktion auswerten kann.

Das ***RUN/HLT Flip-Flop*** zeigt den entsprechenden Zustand des Prozessors an. Es wurde weiter oben bereits erwähnt.

Das ***Zustands-Flip-Flop "SF" (State-Flip-Flop)*** zeigt an, in welcher Phase sich der Prozessor befindet. Es wird weiter unten näher erläutert.

Der Automat für die ***sequentielle Kontrolle*** steuert den gesamten Ablauf der Instruktionsabarbeitung des Prozessors. Ihm werden wir uns jetzt genauer widmen.

4.3.4 Eine Sprache zur Beschreibung der Abläufe im Prozessor

Zur Bearbeitung einer Instruktion, z.b. einem "ADD addr", braucht der Prozessor mehrere Schritte. Die entsprechende Instruktion muß aus dem Speicher geholt werden, dekodiert werden, der Operand muß gelesen werden usw. Zur genauen Beschreibung dieser prozessorinternen Vorgänge wollen wir nun eine einfache Sprache einführen. Grundsätzlich können wir die Vorgänge auf verschiedenen Ebenen beschreiben. Wir könnten z.B. die Logikebene wählen und ein vollständiges Schaltbild des Prozessors entwickeln. Allerdings wollen wir den Prozessor zunächst auf einer höheren Ebene, nämlich auf der sogenannten ***Register-Transfer-Ebene***, beschreiben. Wesentlich für unsere Betrachtungen sind:

- die Beschreibung der Struktur des Prozessors und
- die Beschreibung des Verhaltens des Prozessors.

Die ***Strukturbeschreibung*** auf der Register-Transfer-Ebene umfaßt z.B. die vorhandenen Register, die arithmetisch/logischen Einheiten und die dazugehörige Verbindungsstruktur (die wir bis jetzt noch nicht betrachtet haben). Die Strukturbeschreibung werden wir wie bisher durch intuitive Blockschaltbilder vornehmen.

Die ***Verhaltensbeschreibung*** beschreibt das Zusammenspiel der Komponenten bei der Erfüllung einer bestimmten Aufgabe. Die Zustandsdiagramme, die wir zur Beschreibung von Automaten eingeführt haben, sind eine Form der Verhaltensbeschreibung. Die Sprache, die wir nun einführen, dient der Verhaltensbeschreibung und wird wegen der Ebene, die sie beschreibt als ***Register-Transfer Sprache*** bezeichnet. Sie wurde eingeführt von: T.C. Bartee, I.L. Lebow, I.S. Reed: "Theory and Design of Digital Machines". Ihre Grundelemente sind:

Register	$\mathbf{R_{n-0}}$	bezeichnet die Bitstellen n,n-1,. . . , 0 des Registers R
Transfer:	←	A ← B bezeichnet den Transfer des Inhalts von Register B nach A
Speicher:	**M[addr]**	bezeichnet den Inhalt der Speicherzelle mit der Adresse "addr"
Bedingungen:	**B:**	z.B. R = 0: A ← B, CP7 • RAL: A ← Z

Beispiele:

$a \leftarrow b$	Inhalt von Register b wir nach Register a transferiert.
$a_{3-0} \leftarrow b_{7-4}$	Inhalt der Stellen 4-7 von Register b werden auf die Stellen 0-3 in Register a übertragen.
$a \leftarrow SUM(a,b)$	Die Summe aus den Registerinhalten von a und b wird nach a übertragen
$a \leftarrow SUM(a,1)$	Zum Registerinhalt von a wird 1 addiert. Das Ergebnis wird nach a übertragen.
$a \leftarrow a+b$	Auf die Registerinhalte von a und b wird der Operator "+" angewandt. Das Ergenbis wird nach a übertragen.
$a \leftarrow M[567]$	Speicherwort an der Adresse 567 wird in Register a transferiert.
$R = 0: A \leftarrow B$	Wenn die Bedingung R=0 gilt, wird der Inhalt von Register B nach A übertragen.
$C_n \bullet CLR: a \leftarrow 0, b \leftarrow 0, c \leftarrow 0$	Wenn der Takt C_n anliegt und der Befehl "CLR", werden die Register a, b, c auf 0 gesetzt .
$C_n \bullet HLT:$ Run-FF $\leftarrow 0$	Wenn der Takt C_n anliegt und der Befehl "HLT" wird das Run-FF auf 0 gesetzt.
$C_n \bullet JMP: PC \leftarrow IR_{11-0}$	Wenn der Takt C_n anliegt und der Befehl "JMP" wird der Inhalt des Operandenfelds des IR in den PC transferiert.

Diese einfache Register-Transfer Sprache, die wir als RTL (Register-Transfer-Language) abkürzen, soll nun verwendet werden, um die Abarbeitung einzelner Instruktionen im Prozessor zu beschreiben.

4.3.5 Detaillierter Ablauf der Maschinenbefehle

Abb. 4.10 zeigt den detaillierten Ablauf der Maschinenbefehle des Modellrechners in RTL. Der Ablauf ist in zwei Hauptphasen unterteilt:

- die ***Instruktions-Holphase IF*** (Instruction Fetch Cycle)
- die ***Instruktions-Ausführungsphase EX*** (Instruction Execute Cycle).

In der IF-Phase wird die Instruktion aus dem Speicher gelesen und dekodiert. Einfache Instruktionen, die keine weiteren Operanden benötigen, können in dieser Phase sogar vollständig ausgeführt und abgeschlossen werden. Kompliziertere Operationen, insbesondere arithmetisch/logische Operationen mit zwei Operanden, benötigen eine zweite Phase, die EX-Phase. Die Länge jeder dieser Hauptphasen entspricht einem Speicherzyklus, d.h. der Zeit, die

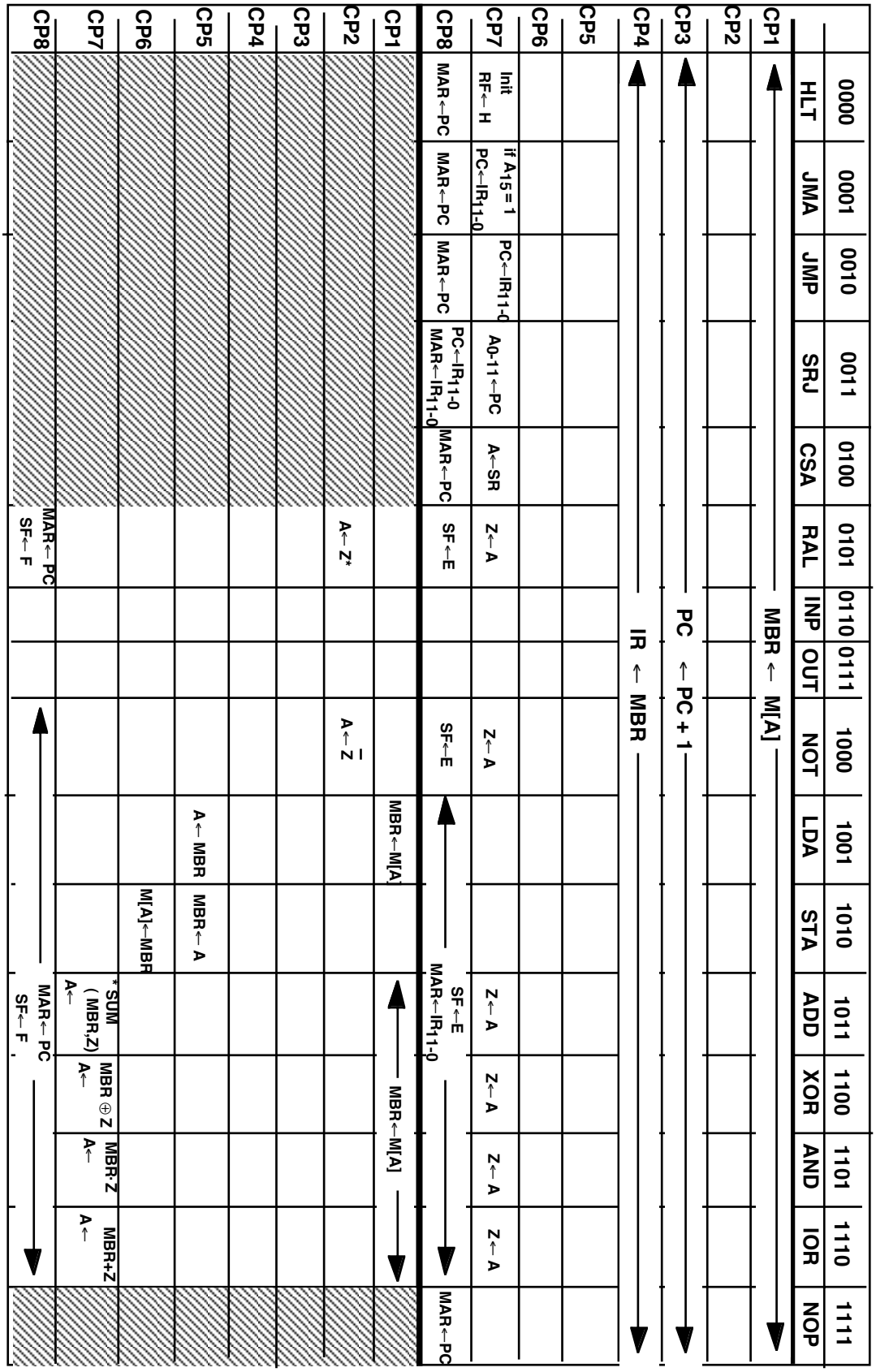


Abb. 4 10 Detaillierter Ablauf der Maschinenbefehle

der Speicher braucht, um zwei aufeinanderfolgende Zugriffe auszuführen. Die Hauptphasen sind weiter in jeweils 8 sogenannte ***Prozessortakte*** unterteilt. Jeder Takt entspricht der Zeit, die der Prozessor braucht, um eine ***Elementaroperation*** (**ELOP**) durchzuführen. Das Zustands-Flip-Flop **SF** speichert ab, in welcher der beiden Phasen sich der Prozessor befindet. Wird zur Ausführung einer Instruktion die EX-Phase benötigt, wird am Ende der IF-Phase SF ← E ausgeführt und das SF in den "Execute"-Zustand gesetzt. Im nächsten Takt führt der Prozessor dann ELOPs in der EX-Phase aus. Am Ende der EX-Phase wird durch SF← F angezeigt, daß der Prozessor sich beim nächsten Takt wieder in der IF-Phase befindet.

Instruktionsholphase (IF-Phase):

Die ersten ELOPs in der IF-Phase sind für alle Operationen gleich:

MBR ← M[A]	Der Inhalt der Speicheradresse A wird ins MBR transferiert.
PC ← PC+1	Der Programmzähler wird um 1 erhöht und zeigt nun auf die folgende Instruktion.
IR ← MBR	Die Instruktion wird aus dem MBR in das Instruktionsregister geladen.

Die Wartezyklen, in denen keine Prozessoraktivität stattfindet, dienen der Synchronisation mit dem langsameren Speicherzyklus.

Einige Befehle, insbesondere die, welche keinen zweiten Operanden oder ALU-Aktivitäten benötigen, können vollständig in der IF-Phase abgearbeitet werden. Dies gilt für HLT, NOP, CSA, und die Sprungbefehle JMP, JMA und SRJ. Es ist zu beachten, daß bei diesen Befehlen im letzten Prozessorzyklus (CP8) eine neue Adresse in das MAR geladen, und dadurch schon der neue Speicherzyklus vorbereitet wird. Während bei HLT, NOP und CSA der Programmzähler einfach inkrementiert wird, muß bei den Sprungbefehlen die Zieladresse des Sprungs, die im Operandenfeld der Instruktion (IR_{11-0}) steht, aus dem IR in das MAR geladen werden.

Bei den Befehlen, zu deren Ausführung die EX-Phase benötigt wird, wird in CP8 das SF in den Zustand E (Execute) gesetzt. Wird ein zweiter Operand benötigt, wird das Operandenfeld der Instruktion, das die Adresse enthält, in das MAR geladen, um den neuen Speicherzyklus zu initiieren.

Instruktionsausführungsphase (EX-Phase):

In der EX-Phase werden die arithmetisch/logischen Operationen, sowie Speicherbefehle LOAD/STORE und Ein/Ausgabebefehle ausgeführt.

• ALU-Instruktionen

Operanden für ALU-Instruktionen stehen in den Registern A und MBR. Da A auch das Zielregister für alle ALU-Instruktionen ist, d.h. der jeweilige in A gespeicherte Operand während der Ausführung der Operation überschrieben würde, wird bereits in der IF-Phase der Operand, der in A steht, in das Hilfsregister Z transferiert (Z ← A in CP7).

• LOAD/STORE-Instruktionen

Die Wirkungsweise der LOAD/STORE-Instruktionen kann aus Abb. 4.10 abgeleitet werden.

• Ein/Ausgabe - Instruktionen

Für den Modellrechner wird eine sehr einfache Ein/Ausgabe (EA) angenommen. Die EA-Geräte werden einfach wie Speicher behandelt. Der Prozessor stellt lediglich die Adresse im Operandenfeld der Instruktion für die Adressierung des externen Gerätes bereit und überträgt am Ende der EX-Phase beim OUT-Befehl den Inhalt des Akkumulators zum EA-Gerät, bzw. liest er bei der INP-Instruktion einen Wert vom externen Gerät in den Akkumulator [4].

4.4 Der Datenpfad

Die Komponenten des Datenpfads wurden in 4.3.2 eingeführt. In diesem Kapitel soll die Verbindungsstruktur entworfen werden, die notwendig ist, um die ELOPs aus Abb. 4.10 zu realisieren. Darüberhinaus wird die ALU im Detail betrachtet.

4.4.1 Die Verbindungsstruktur der CPU

Zum Transfer der Registerinhalte muß eine Verbindungsstruktur zur Verfügung gestellt werden. Die Verbindungen zwischen den einzelnen Registern wird durch Busse realisiert[5]. Ein ***Bus*** verbindet mehrere Teilnehmer, die über den Bus Informationen austauschen. Er wird gekennzeichnet durch:

1.) ein physikalisches Transportmedium. In unserem Modellrechner sind das eine Menge paralleler Leitungen, auf denen jeweils ein Bit eines Registers übertragen wird.

2.) ein Protokoll, das festlegt, wer Nachrichten auf den Bus legen darf und in welchem Format diese Nachrichten übertragen werden. Im Fall unseres Modellrechners wird der Bus durch die entsprechenden ELOPs gesteuert. Das Protokoll ist sehr einfach und wird in Zusammenhang mit Abb. 4.11 erläutert.

---

[4] Eine ausführlichere Diskussion der EA-Schnittstelle wird in Kapitel 8. beschrieben.

[5] Siehe Skript: "Digitale Logik" .

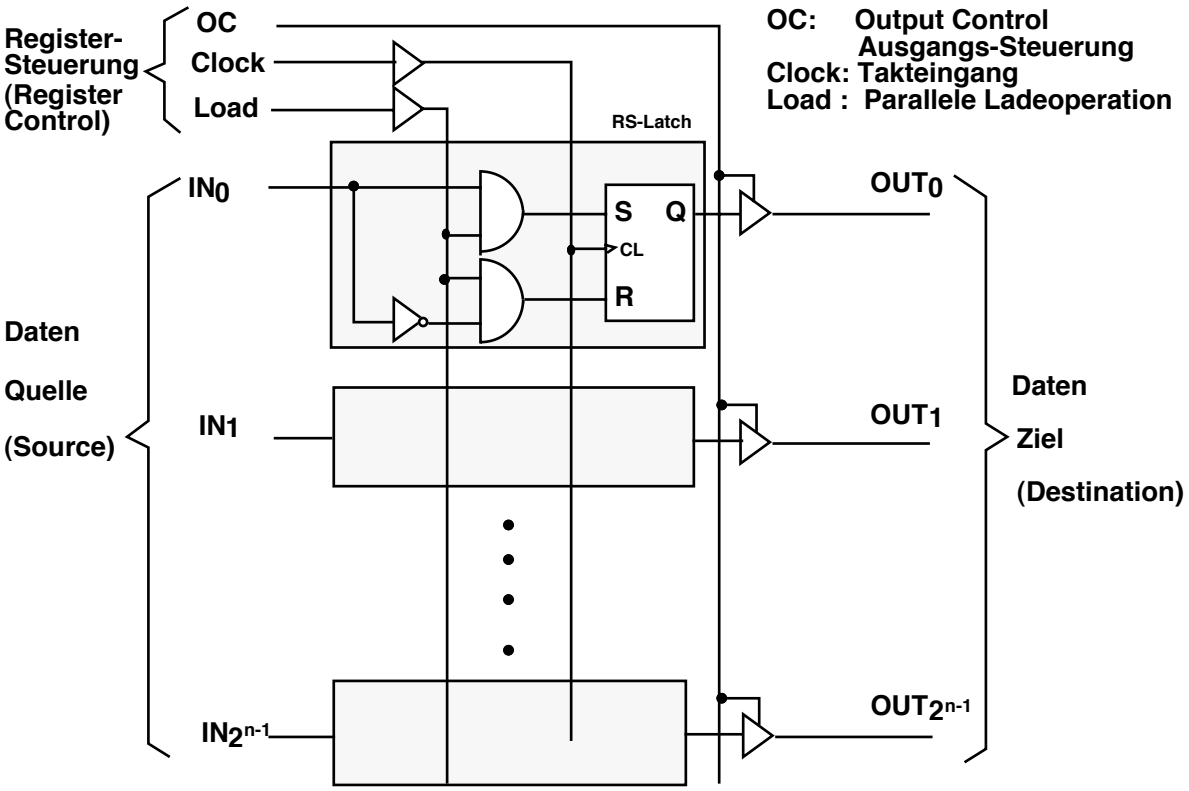


Abb. 4.11 Busanbindung eines Registers

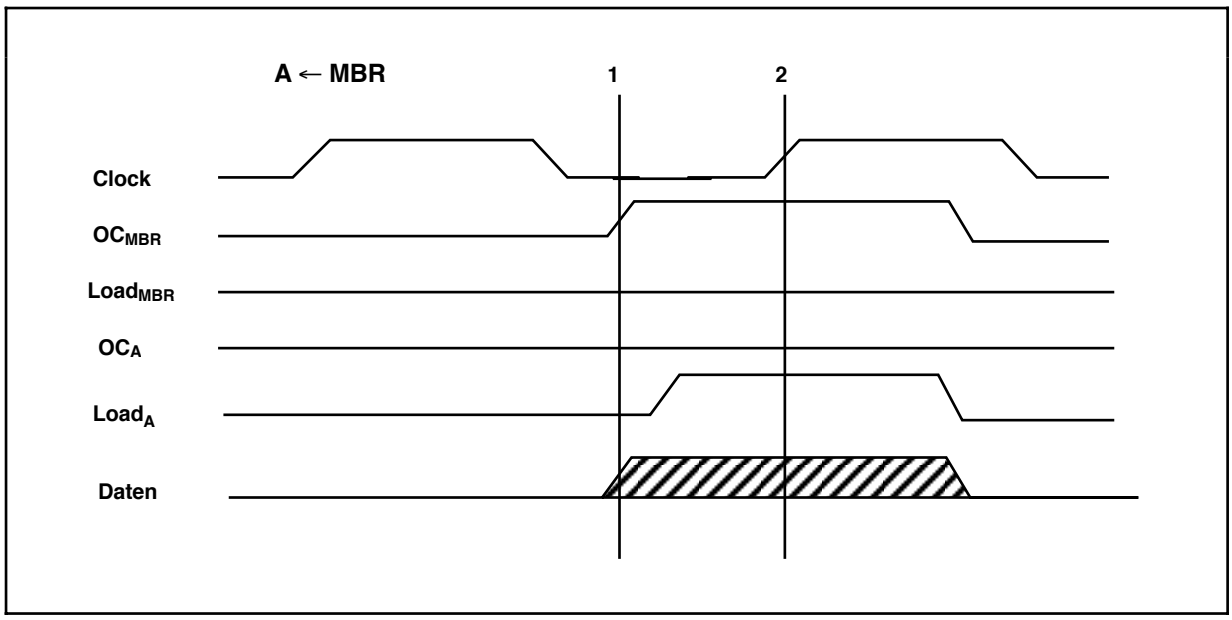


Abb. 4.12 Zeitablauf des Busprotokolls

Die Funktionsweise der Busanbindung eines Registers ist in Abb. 4.11 dargestellt. Die RS-Latches bilden zusammen das Register. Soll das Register geladen werden, wird die Leitung

Load aktiviert und gleichzeitig die entsprechende Information auf den Bus (d.h. an die Eingänge $IN_0, IN_1, \ldots IN_{2^{n-1}}$) gelegt. Durch den Takt (Clock) werden die Werte auf dem Bus in das Register übernommen.

Soll der Inhalt des Registers auf den Bus gelegt werden, wird die Leitung OC aktiviert. Dadurch werden die Ausgänge der Tri-State-Puffer (OUT_0, OUT_0, $OUT_{2^{n-1}}$) vom hochohmigen Zustand auf den Wert von Q gesetzt. Abb. 4.12 zeigt den zeitlichen Ablauf des Datentransfers am Beispiel der ELOP: A ← MBR. Zum Zeitpunkt "1" werden die Bustreiber am Ausgang des MBR aktiviert und dadurch der Inhalt der Latches des MBR auf die Busleitungen gelegt. Im Zeitdiagramm werden dazu nicht alle Datenleitungen mit ihren Belegungen einzeln dargestellt, sondern sie werden als "Daten" zusammengefasst. Die Steuerleitung $Load_A$ wird aktiviert und damit liegen die Werte auf den Busleitungen an den Latches des A-Registers an. Zum Zeitpunkt "2" werden sie dann mit der steigenden Flanke des Taktimpulses in das A-Register übernommen.

In unserem Modellrechner sind zwei verschiedene Bussysteme realisiert:

- Der Adreßbus
- Der Datenbus

Darüberhinaus gibt es für spezielle Zwecke feste Verbindungen:

- Speicher - MBR
- MAR - Speicher
- MBR - ALU
- OPCODE-Feld des IR - Steuereinheit (CU)

Da sie nicht mehrere Teilnehmer verbinden, werden sie nicht als Busse bezeichnet. Sie unterliegen auch nicht einem allgemeinen Busprotokoll, sondern jeweilig angepaßten Bedingungen. Das Zusammenwirken von Speicher, MBR und MAR wurde im Zusammenhang mit der Beschreibung der Speicherschnittstelle erklärt. Die Verbindung MBR-ALU dient der Bereitstellung des Speicheroperanden bei zweistelligen arithmetisch/logischen Operationen. Die Bedeutung der Verbindung zwischen IR und Steuereinheit wird im Abschnitt "Steuereinheit" erläutert.

Grundsätzlich ist der Ansatz, zwei getrennte Busse für Adressen und Daten bereitzustellen, eine Entwufsentscheidung, bei der der Konflikt zwischen Kosten und Geschwindigkeit zugunsten der Geschwindigkeit entschieden wurde. Sind getrennte Datenwege für Adressen und Daten vorhanden, können Adressen und Daten nebenläufig übertragen werden, was grundsätzlich die Leistungsfähigkeit der CPU erhöht. Im Modellrechner wird das allerdings nicht ausgenutzt. Dies liegt daran, daß aus Gründen der Einfachheit und Übersichtlichkeit genügend Zeit reserviert wurde, damit in einem Taktzyklus jeweils nur eine ELOP ausgeführt werden muß.

Der Adreßbus

Der Adreßbus hat eine Breite von 12 Bit, um eine Adresse parallel zu übertragen. Quellregister sind PC und IR (nur das Adreßfeld). Mögliche Zielregister sind PC, MAR und A. Abb. 4.10 zeigt den Adreßtransfer im einzelnen. Der PC enthält immer Adressen von Instruktionen. Das IR enthält bei Sprüngen die Zieladresse, bei ALU-Instruktionen eine Operandenadresse. Die Verbindung der Adreßregister zum Akkumulator wird im Unterprogrammsprung zur Abspeicherung der Rücksprungadresse benötigt (JSR • CP7: $A_{11\text{-}0} \leftarrow$ PC).

Die Daten-Selektions-Leitungen (DSL) werden benutzt, um ein externes Gerät zu selektieren.

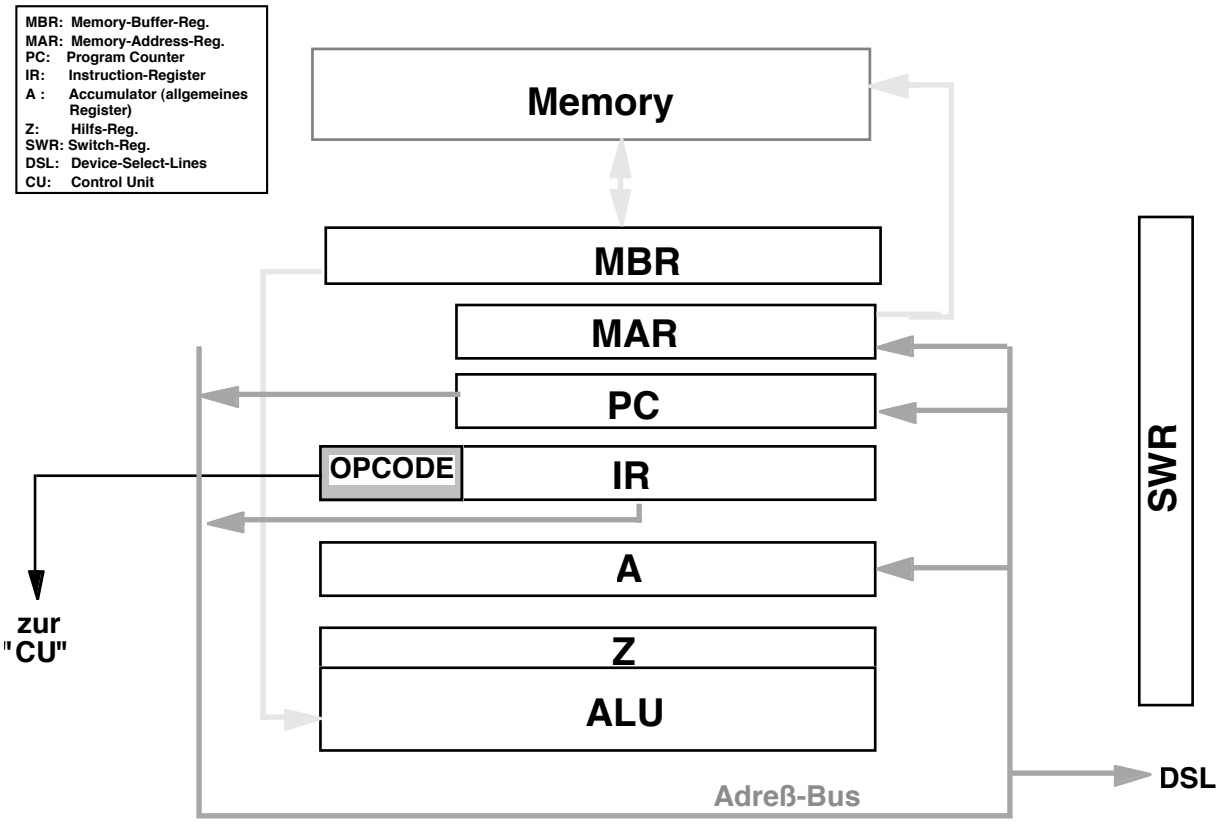


Abb. 4.13 Der Adreßbus

Der Datenbus

Die vollständige Busstruktur des Modellrechners ist in Abb. 4.14 dargestellt. Der Datenbus hat eine Breite von 16 Bit, was der Wortbreite des Speichers und der Datenregister entspricht.

- Quellregister sind: MBR, SWR und A. Außerdem wird das Ergebnis einer ALU-Operation über den Datenbus in das Register A transferiert, so daß die ALU ebenfalls als Quelle am Datenbus angeschlossen ist.
- Zielregister sind: MBR, Z und A.

Es ist zu beachten, daß der PC nicht direkt vom Datenbus geladen werden kann. Dies bedeutet, daß durch die ALU manipulierte Adressen nicht direkt in den PC geladen werden können. Auch kann die abgespeicherte Rücksprungadresse bei Unterprogrammsprüngen nicht direkt in den PC geladen werden. Adreßmanipulationen sind nur durch selbstmodifizierenden Code möglich, wie in Abb. 4.7 gezeigt wurde. Dies ist ein weiteres Beipsiel dafür, wie der Aufwand, der für die Hardwarerealisierung eines Rechners eingeplant wird, gegen den Aufwand bei der Programmierung abgewogen werden kann.

Der Datenbus dient auch zum Austausch von Daten mit externen Geräten. Die "DATA I/0"-Leitungen sind hier einfach der nach außen geführte Datenbus.

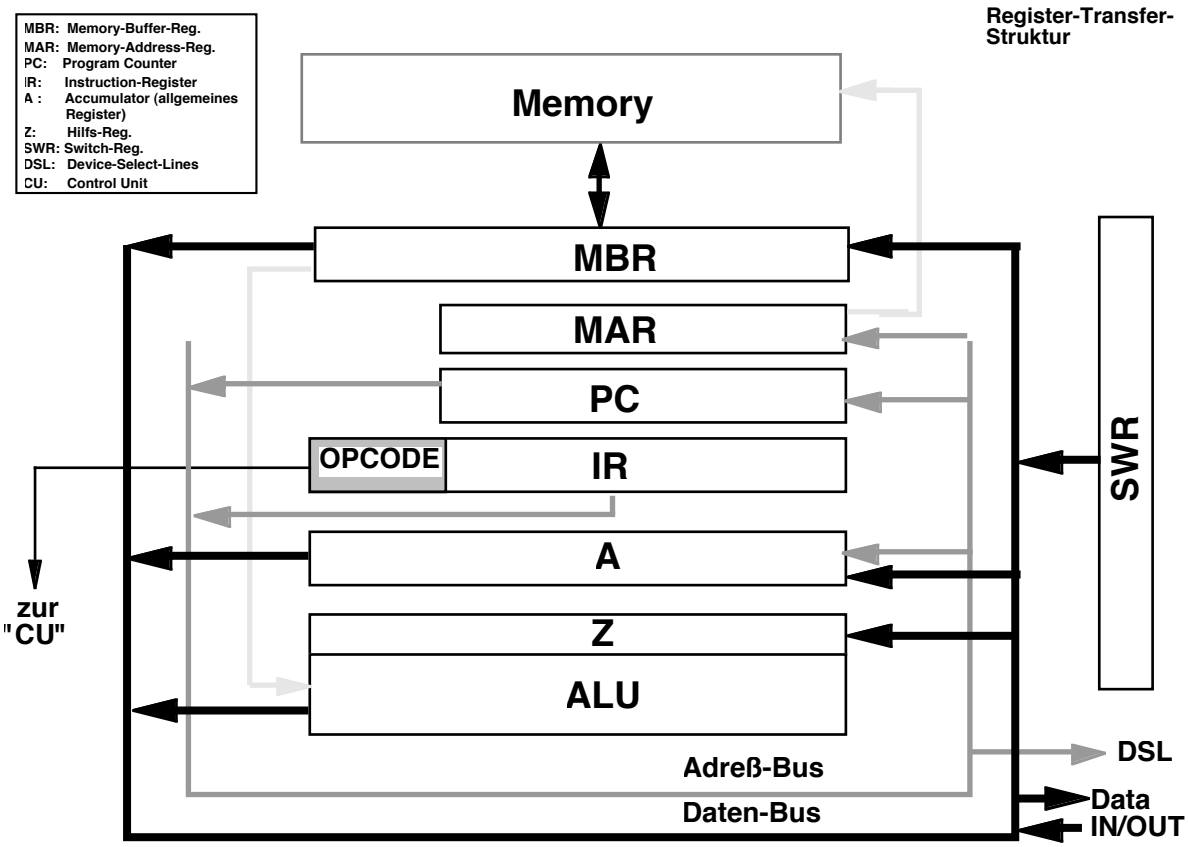


Abb. 4.14 Die vollständige Busstruktur des Modellrechners

4.4.2 Die ALU

Abb. 4.15 gibt den Aufbau der ALU an, einschließlich der Steuerleitungen zur Auswahl der Funktionen. Gezeigt wird ein einzelnes Bit der ALU.

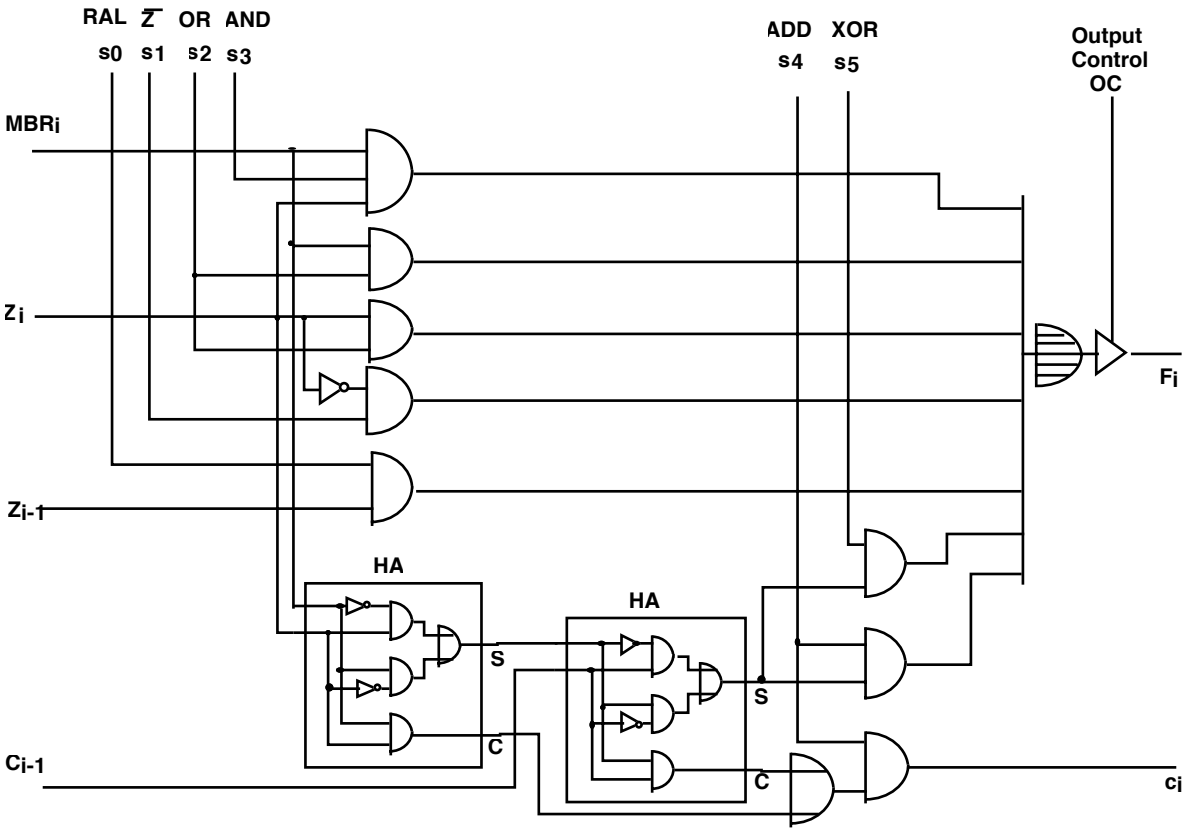


Abb. 4.15 Einfache ALU mit Steuerleitungen

Die boolschen Gleichungen für die ALU sind in Abb. 4.16.a und b angegeben, wobei die Variablen a,b, c, d folgende Zuordnung zu den in Abb. 4.15 gezeigten Eingängen der ALU haben: $MBR_i = a$, $Z_i = b$, $Z_{i-1} = c$, $C_{i-1} = d$.

$$F_i = s_0c + s_1\overline{b} + s_2a + s_2b + s_3ab + s_4a\overline{b}\,\overline{d} + s_4\overline{a}b\overline{d} + s_4\overline{a}\,\overline{b}d + s_4abd + s_5a\overline{b} + s_5\overline{a}b$$

RAL $\overline{Z}$ OR AND ADD (SUMME) XOR

Abb. 4.16 a. Gleichungen für den Funktionsausgang der ALU

$$C_i = s_4(ab + (a\overline{b} + \overline{a}b)\,d)$$
$$= s_4ab + s_4a\overline{b}d + s_4\overline{a}bd$$

Abb. 4.16 a. Gleichungen für den Übertragsausgang der ALU

Die Gesamtstruktur der ALU mit 16 Stufen ist in Abb. 4.17 skizziert. Die Bedeutung der Eingänge kann aus dem logischen Schaltbild in Abb. 4.15 entnommen werden. Der Übertrag aus der vorletzten und der letzten Stufe werden durch ein EXOR verglichen, um festzustellen, ob ein Überlauf (Overflow) vorliegt.

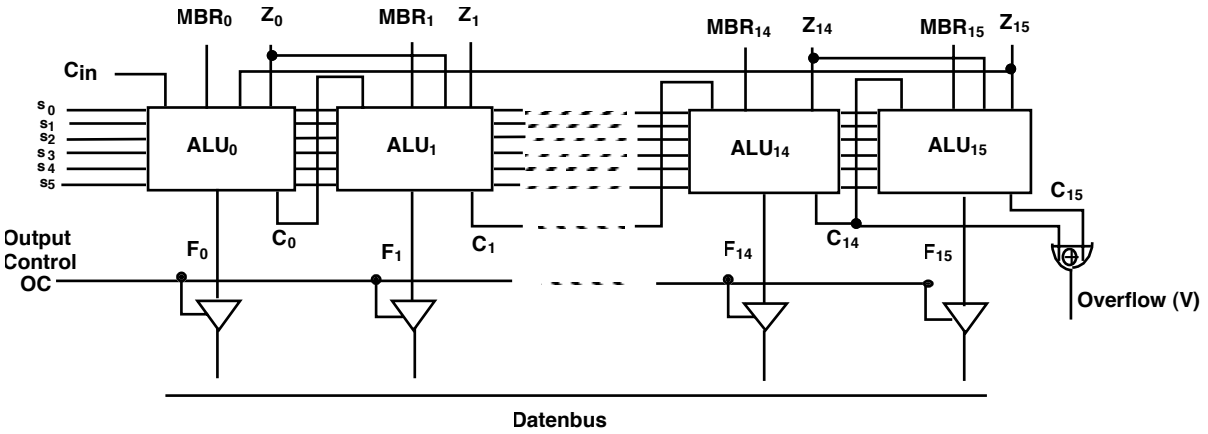


Abb. 4.17 Gesamtstruktur der ALU

4.5 Die Kontrolleinheit

Wie wir gesehen haben, ist eine Maschineninstruktion aus mehreren ELOPs aufgebaut. Die Kontrolleinheit steuert den detaillierten Ablauf der ELOPs. Grundlage für den Zeitablauf einer Maschinenoperation ist der Speicherzyklus und der Prozessortakt. In unserem Modellrechner wird ein Speicherzyklus in 8 Taktintervalle unterteilt. In jedem dieser Taktintervalle kann eine ELOP ausgeführt werden. Abb. 4.18 a. zeigt den Aufbau eines Taktgenerators, der 8 gleichlange Taktintervalle erzeugt. Er besteht aus einem ***Oszillator***, d.h. einer Komponente, die periodische Schwingungen mit der Periodendauer eines Taktintervalls erzeugt. Die Anzahl der Schwingungen/Zeiteinheit wird als Frequenz des Oszillators bezeichnet. Das Maß für die Frequenz ist Hz (Hertz) und wird mit ***f* = Schwingungen / 1 Sekunde** angegeben. Die Dauer einer Periode ist dann $p = 1/f$. Ein moderner Prozessor mit einem Takt von 200 MHz (1Mhz = 10^6 Hz) hat demnach eine Periode von 5 ns (ns = Nanosekunde = 10^{-9} Sekunden). Ein einfacher Oszillator, der mit Invertern aufgebaut ist, ist in Abb. 4.18 b. dargestellt. Durch die Rückkoppplung am ersten Inverter wird eine Schwingung erzeugt, deren Frequenz durch die zeitbestimmenden Komponenten R und C beeinflußt werden kann[6]. Der zweite Inverter dient der Impulsformung, d.h. er macht aus den unregelmäßig geformten Oszillatorschwingungen Rechteckimpulse. Das Verhältnis der Dauer des logischen 0-Pegels und des logischen 1-Pegels (das sogenannte Tastverhältnis) ist nicht "1", d.h. sie sind nicht

[6] Heute wird zur Stabilisierung des Oszillators meist ein Quartz als zeitbestimmendes Element verwendet.

gleich lang. Um zu erreichen, daß gleich lange Intervalle von jeweils 0 und 1 entstehen, ist ein JK-Flip-Flop vorgesehen.

Der Ausgang des Oszillators und der Ausgang des RUN/HLT-Flip-Flops sind an ein UND-Gatter geführt. Dadurch werden die Taktimpulse nur dann weitergeleitet, wenn sich das RUN/HLT-Flip-Flop im Zustand RUN befindet. Ist dies der Fall, gelangen die Takte des Oszillators an einen 3-Bit-Zähler. Dieser generiert zyklisch die Binärzahlen 0-7, die von einem 3-zu-8-Dekodierer zu entsprechenden Taktintervallen auf den Taktleitungen CP1, . . ., CP8 umgewandelt werden.

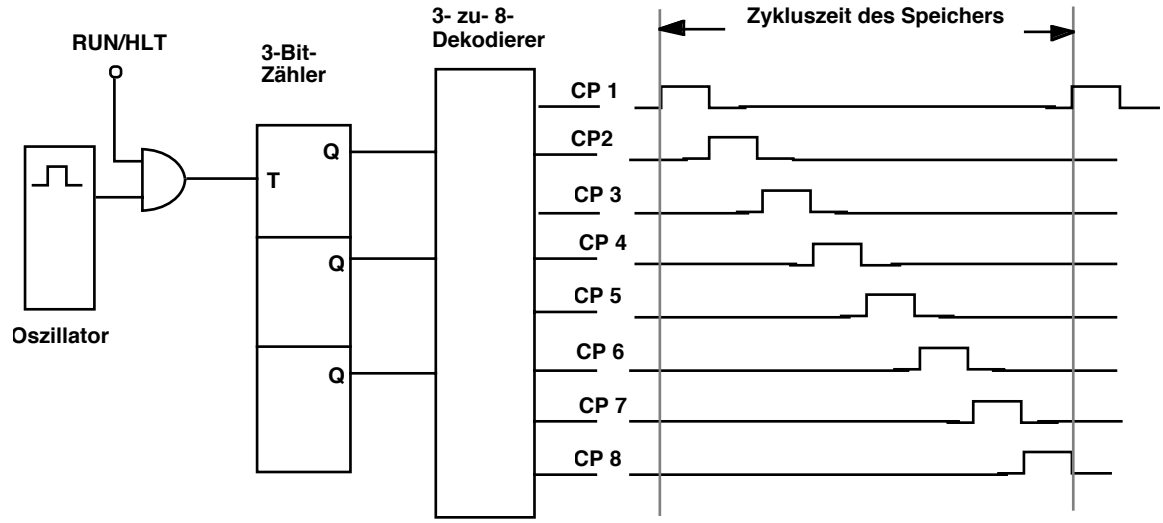


Abb. 4.18 a. Der 8-Phasen Taktgenerator

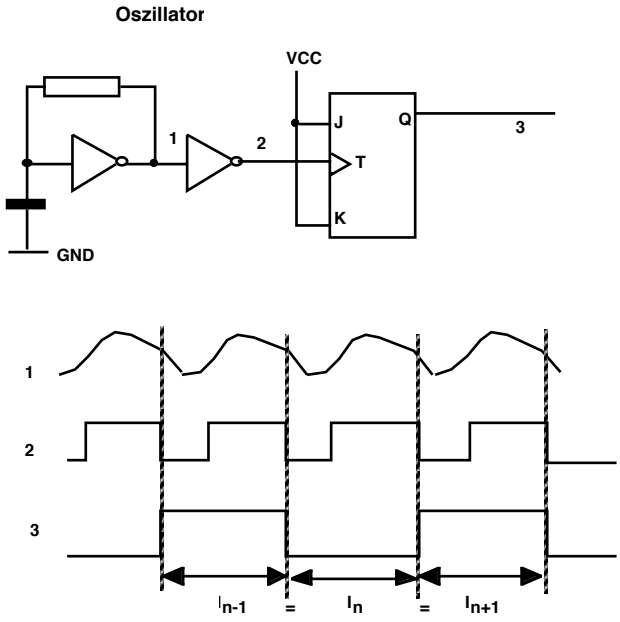


Abb. 4.18 b. Aufbau und Impulsformen des Oszillators

In Abb. 4.19 ist der schematische Aufbau der Kontrolleinheit dargestellt. Die wesentlichen Komponenten sind:

1.) der 8-Phasen Taktgenerator
2.) der Instruktionsdekoder
3.) die Status-Flip-Flops
4.) das kombinatorische Schaltnetz zu Erzeugung der Steuersignale.

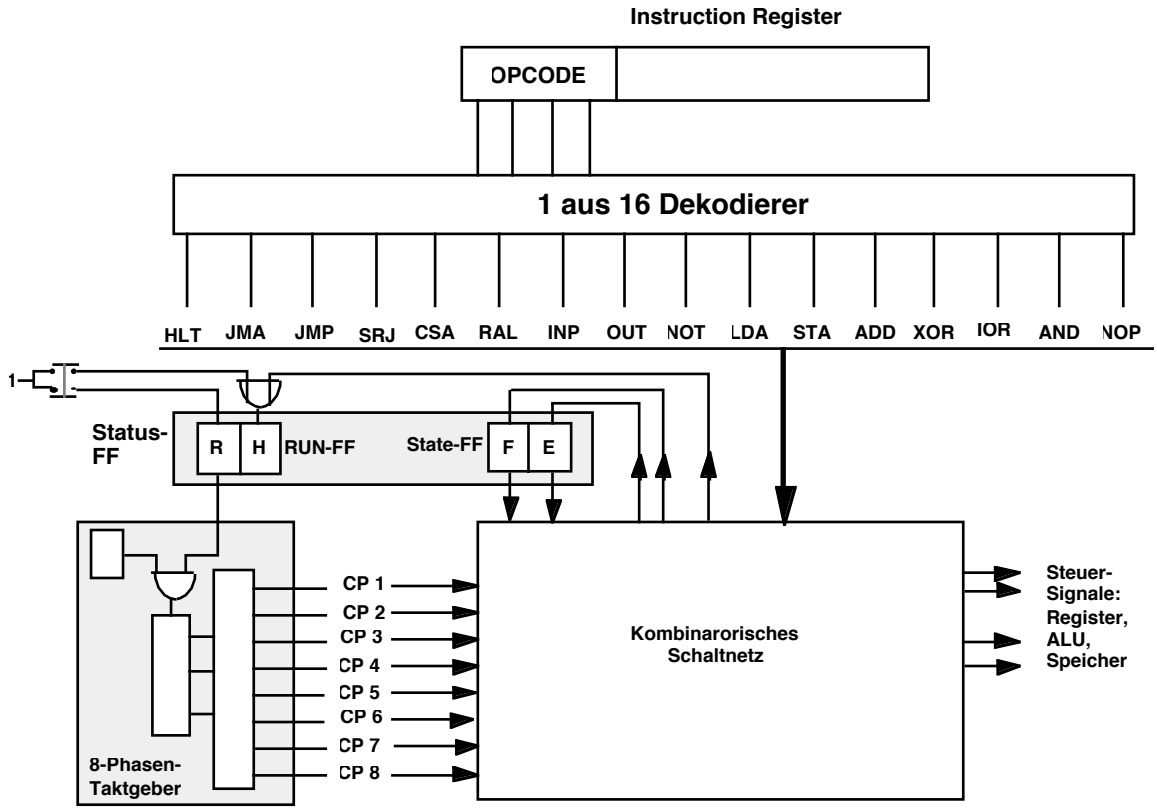


Abb. 4.19 Schematischer Aufbau der Kontrolleinheit

Wir wollen nun im einzelnen die Logik des kombinatorischen Schaltnetzes entwickeln. Die Eingangsleitungen sind:

- 16 Leitungen aus dem Instruktionsdekoder, von denen genau eine Leitung aktiviert ist und einen auszuführenden Maschinenbefehl repräsentiert.
- 8 Leitungen aus dem Taktgenerator, von denen genau eine Leitung aktiviert ist und die aktuelle Taktphase angibt.
- 2 Eingänge des Zustands-Flip-Flops SF, die angeben, ob die CPU sich in der IF- oder EX-Phase befindet.

Aus Abb. 4.10 können wir entnehmen, daß die ersten vier Taktintervalle CP1,.., CP4 für alle Instruktionen gleich sind. Bezogen auf die kombinatorische Schaltung zur Realisierung der Steuerung bedeutet das, daß man die Belegung der Steuerleitungen des Instruktionsdekoders nicht berücksichtigen muß. Abb. 4.20 gibt die entsprechende Schaltung an.

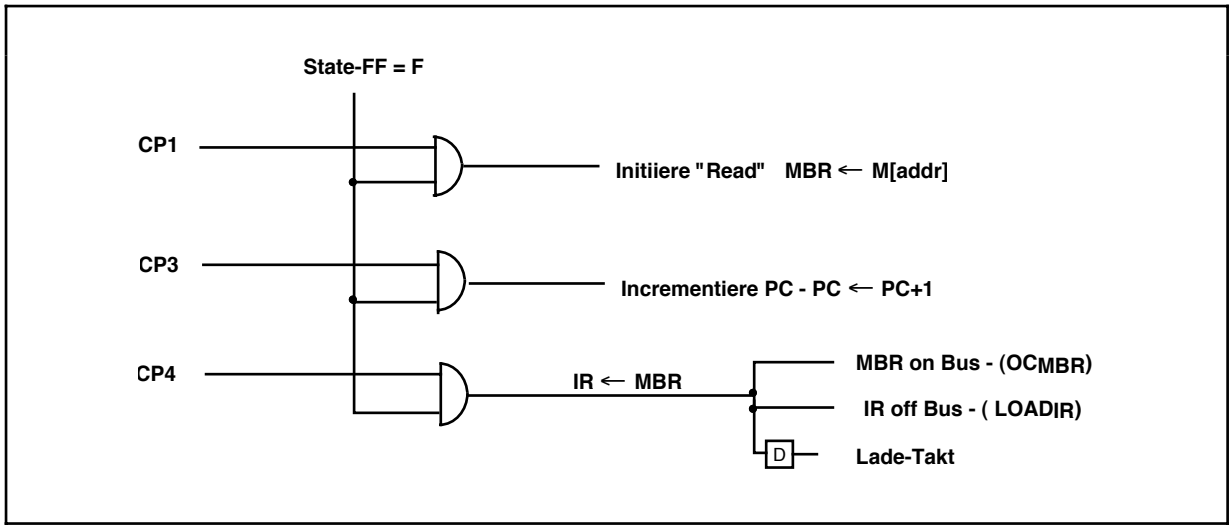


Abb. 4.20 Steuersignal während der gemeinsamen ersten Phase des IF-Zyklus

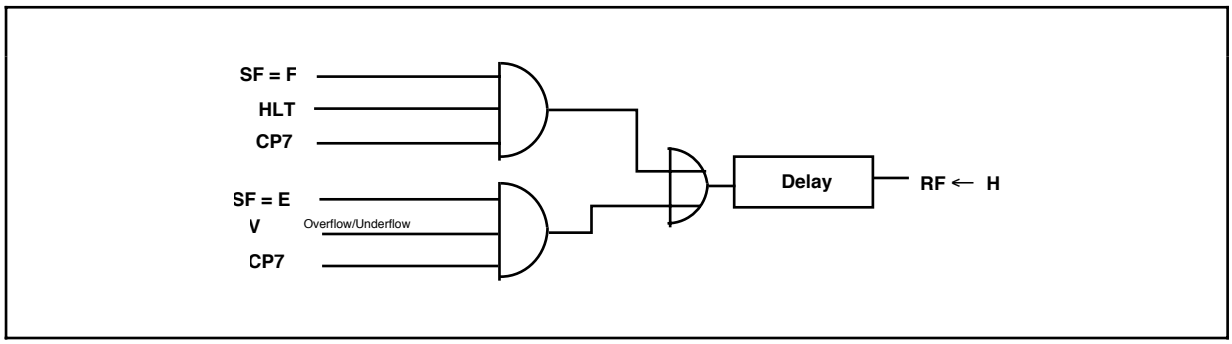


Abb. 4.21 Steuerung des RUN/HLT-Flip-Flops

Die Instruktionen HLT, NOP und CSA sowie die Sprungbefehle JMP, JMA und JSR können in der IF-Phase vollständig ausgeführt werden. Abb. 4.21 gibt die Steuerung des RUN/HLT-FF an. Abb. 4.22 zeigt die Erzeugung der Steuersignale für JMP und JMA. Bei JMA wird das Vorzeichenbit des Akkumulators (A_{15})als Sprungbedingung ausgewertet. Außerdem wird der PC mit der Adresse des Sprungziels aus dem Operandenfeld des IR geladen.

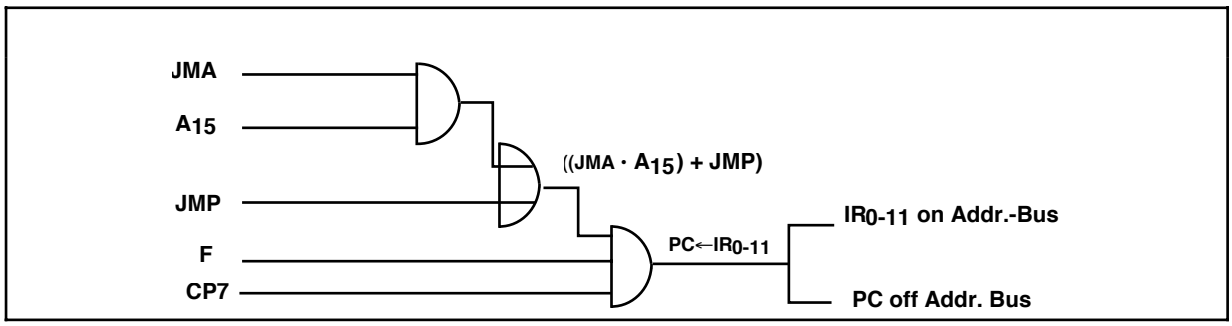


Abb. 4.22 Steuerung der Instruktionen JMP und JMA

Bei dem Unterprogrammsprung JSR wird in Taktintervall CP_7 die Adresse im PC, die als Rücksprungadresse dient, in das Register A geladen. Der PC und das MAR werden in CP_8 mit der Adresse des Sprungziels aus dem Operandenfeld des IR geladen.

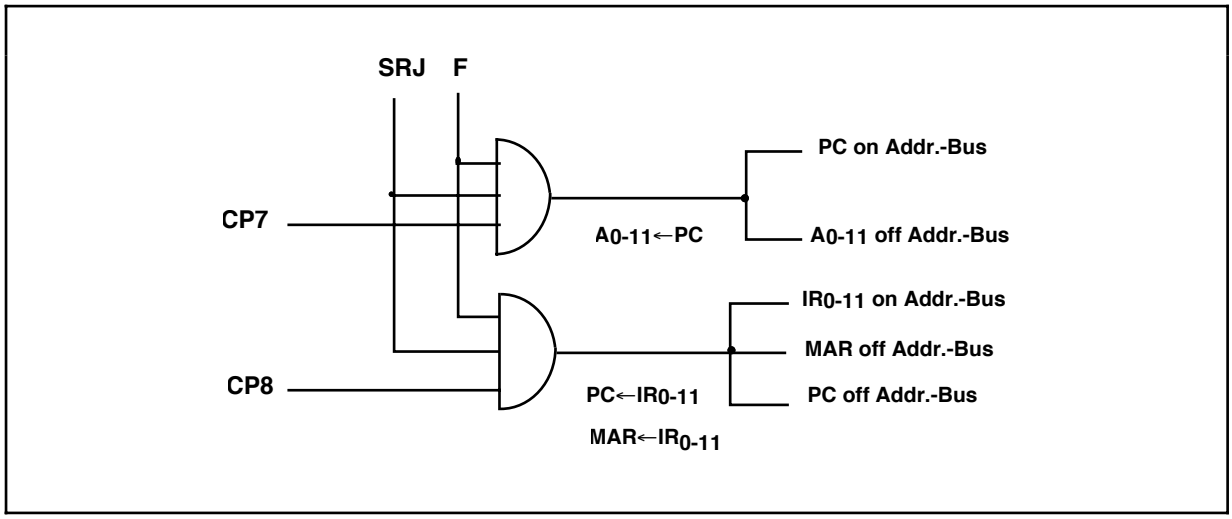


Abb. 4.23 Steuerung der Instruktion JSR

Bei Ausführung der Instruktion CSA wird der Inhalt des SWR in den Akkumulator geladen.

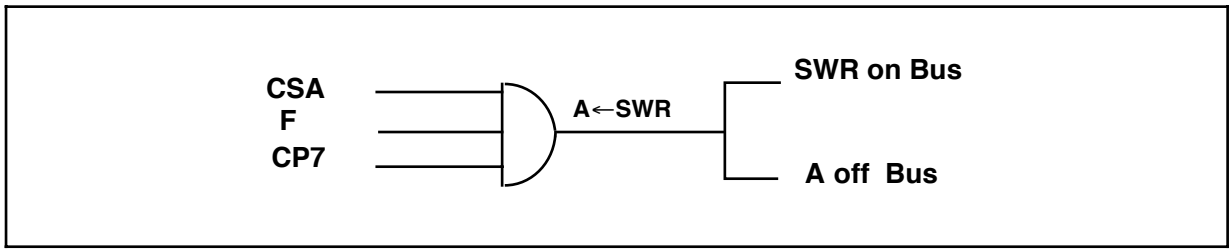


Abb. 4.24 Steuerung der Instruktion CSA

Die Steuersignale, die den IF-Zyklus für die oben genannten Instruktionen beenden sind in Abb. 4.25 angegeben.

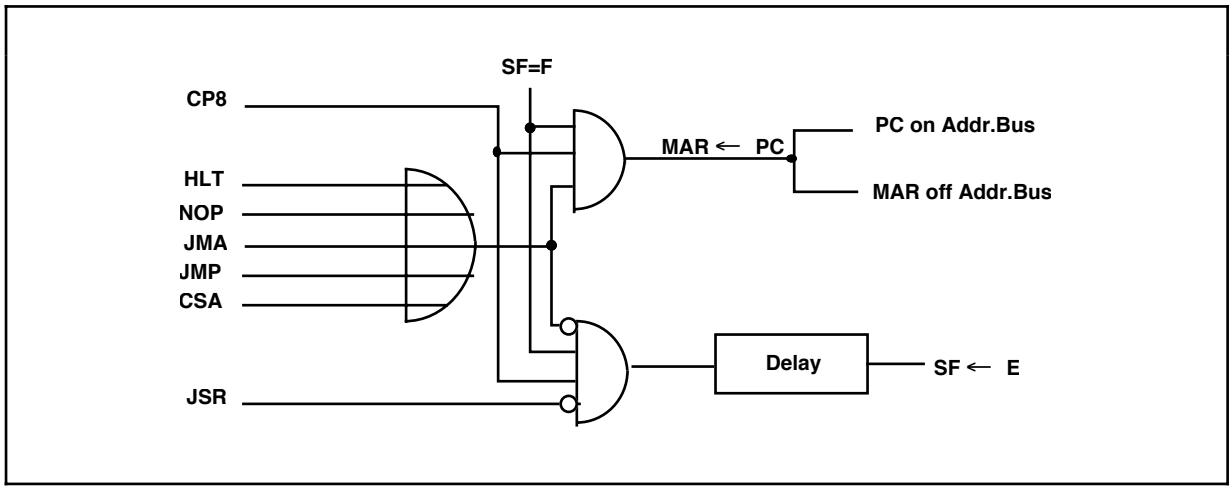


Abb. 4.25 Beendigung aller 1-Zyklus-Instruktionen

Für die Befehle, die zwei Zyklen zu ihrer Ausführung benötigen, wird zur Vereinfachung ein DTA- und ein REF-Signal erzeugt. REF wird erzeugt für alle Instruktionen, die in der EX-Phase einen Speicherzugriff durchführen. DTA wird zusätzlich für alle ALU-Instruktionen erzeugt, die einen zweiten Operanden benötigen.

Die Generierung des REF- und DTA-Signals ist in Abb. 4.26 angegeben. Abb. 4.27 zeigt die gemeinsame Aktivität aller ALU-Operationen, die einen zweiten Operanden benötigen. Hier wird der Inhalt des Akkumulators in das Hilfsregister Z übertragen, damit der Akkumulator als Zielregister der ALU-Operation zur Verfügung steht. Das DTA-Signal wird hier verwendet.

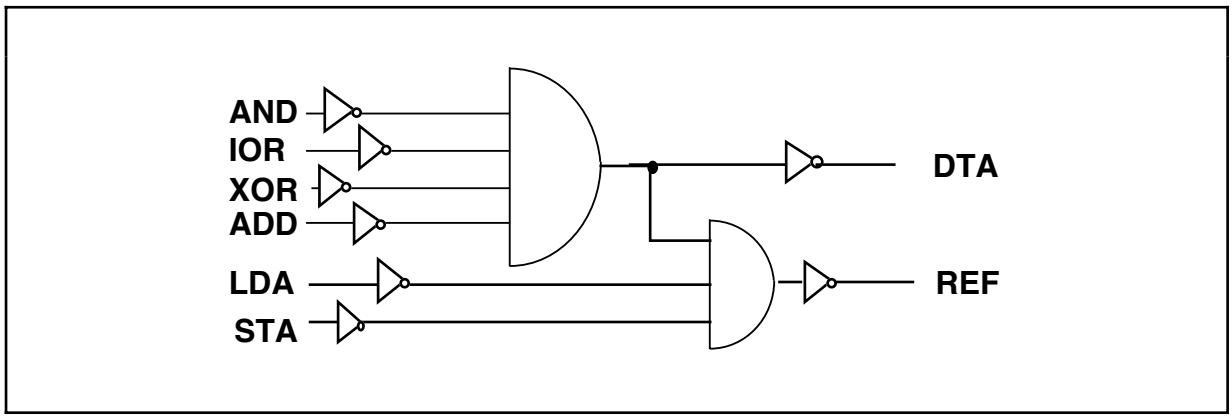


Abb. 4.26 Erzeugung der DTA und REF Steuersigale

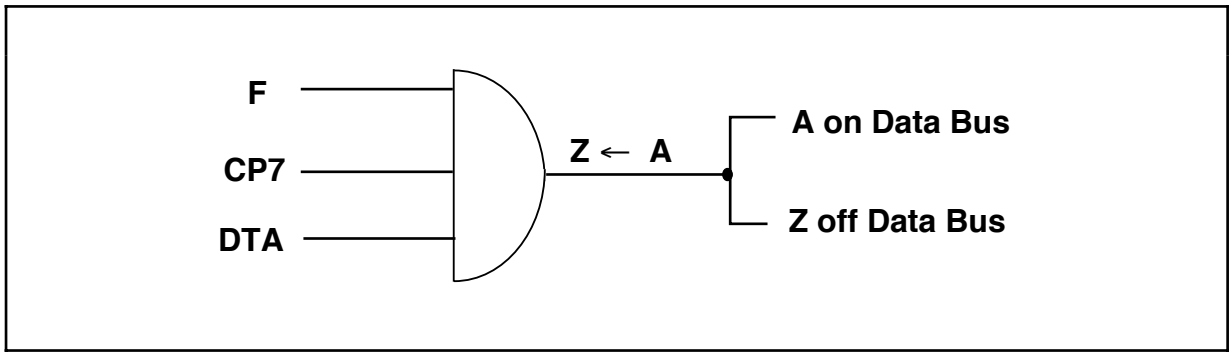


Abb. 4.27 Gemeinsame Steuersignale für ALU-Operationen

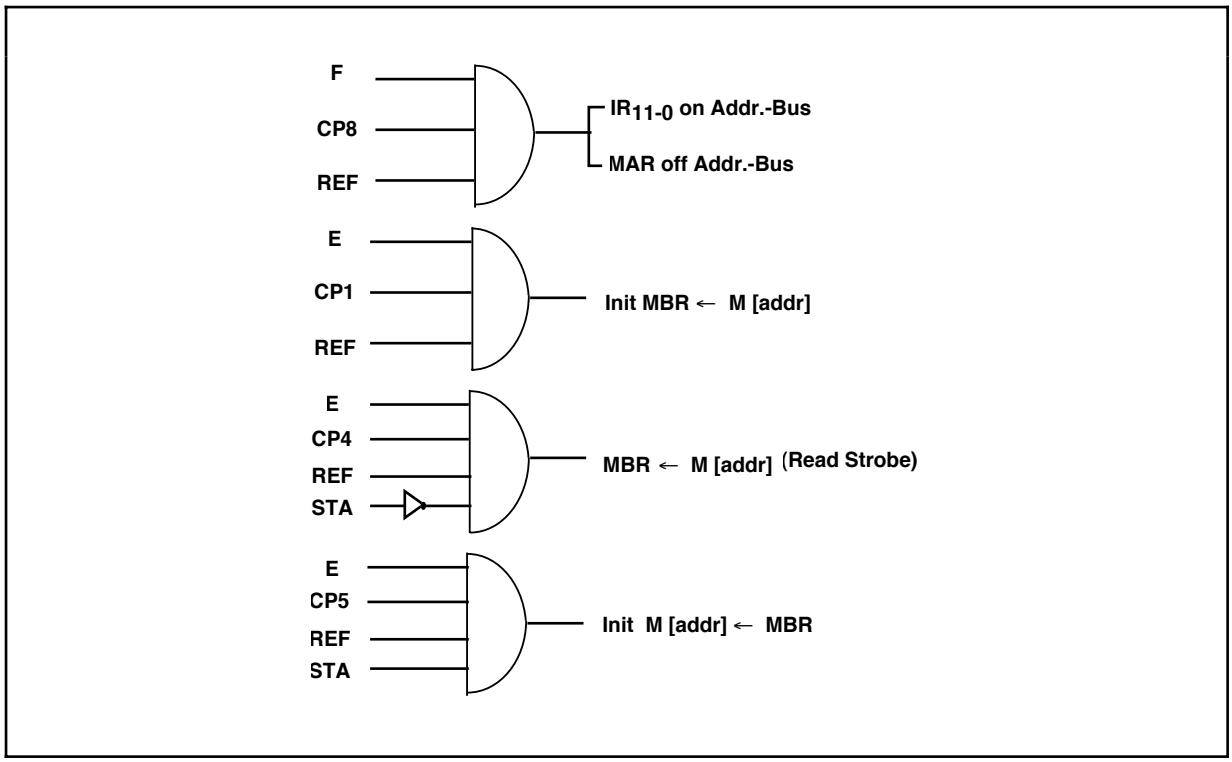


Abb. 4.28 Gemeinsame Steuersignale zum Speicherzugriff (EX)

Abb. 4.28 zeigt alle Steuersignale, die für den Speicherzugriff generiert werden müssen. Bereits in der IF-Phase wird der Speicherzugriff durch den Transfer der Adresse in das MAR initialisiert. Er wird dann in den Phasen CP_1,..., CP_4 ausgeführt. In CP_5 wird der Schreibzugriff zum Speicher (STA) durchgeführt.

Die Steuersignale zur Selektion der entsprechenden ALU-Operation können direkt aus dem Instruktionsdekoder abgeleitet und den entsprechenden ALU-Steuereingängen zugeführt werden. Abb. 4.29 zeigt dies am Beispiel der Instruktion ADD.

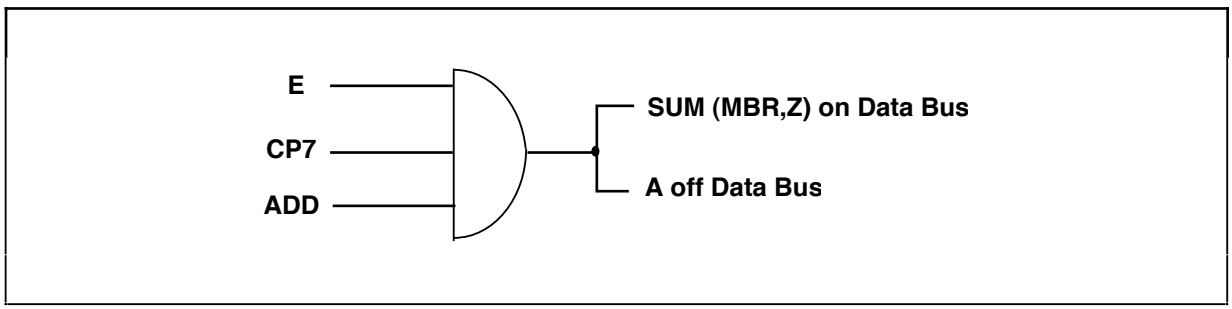


Abb. 4.29 Steuerung der ALU am Beispiel ADD

Der Abschluß der EX-Phase für alle Instruktionen ist in Abb. 4.30 dargestellt. Die Adresse der nächsten Instruktion wird aus dem PC in das MAR geladen. Das Zustands-Flip-Flop SF wird auf F gesetzt und zeigt damit die nächste IF-Phase an. Damit keine undefinierten Zustände entstehen, wird das Signal für das SF um die Zeitdauser eines halben Takts verzögert.

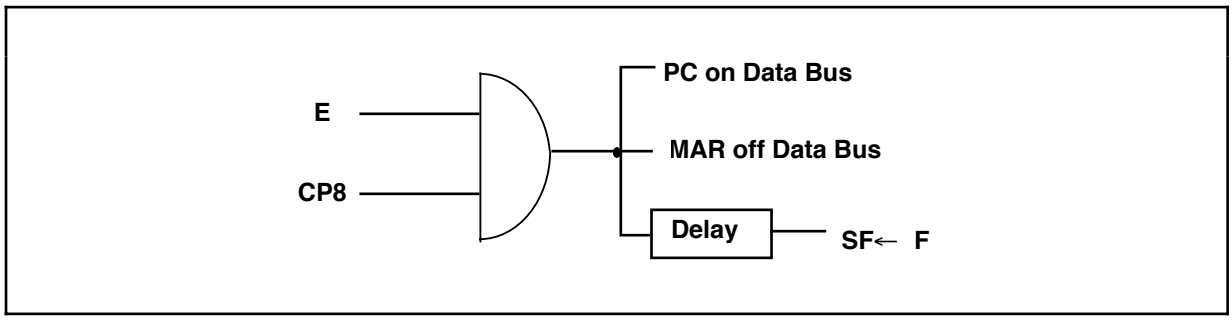


Abb. 4.30 Abschluß der EX-Phase

Damit haben wir den Entwurf der Kontrolleinheit auf der logischen Ebene abgeschlossen. Wir können uns vorstellen, daß wir die gesamte Kontrolleinheit in einer der in Kapitel 3 besprochenen komplexeren programmierbaren Komponenten realisieren können.

5 Mikroprogrammierung

... I realized that the solution was to turn the control unit into a computer in miniature by adding a second matrix to determine the flow of control at the microlevel .....
Maurice Wilkes,
Memoires of a Computer Pioneer

In unserem Modellrechner haben wir die Kontrolleinheit durch zustandsspeichernde Komponenten wie Flip-Flops und Zähler in Verbindung mit einem kombinatorischen Schaltnetz realisiert. Dies ist der Ansatz, wie er klassischerweise für einen sequentiellen Automaten verfolgt wird (Abb. 5.1). Die Wahrheitstafel der logischen Funktionen wird dabei in logische Gatter umgesetzt. Wir haben aber bereits Ansätze kennengelernt, welche die vollständige Wahrheitstafel in einem Speicher ablegen und dann durch "look-up" den Funktionswert ermitteln. Die Eingänge des Schaltnetzes dienen als Adresse in den Speicher, der Speicherinhalt repräsentiert die Ausgangsbelegung. In Kapitel 3 wurden die Vor-und Nachteile dieses Ansatzes im Zusammenhang mit dem ROM- bzw. PAL-Ansatz zur Realisierung logischer Funktionen diskutiert.

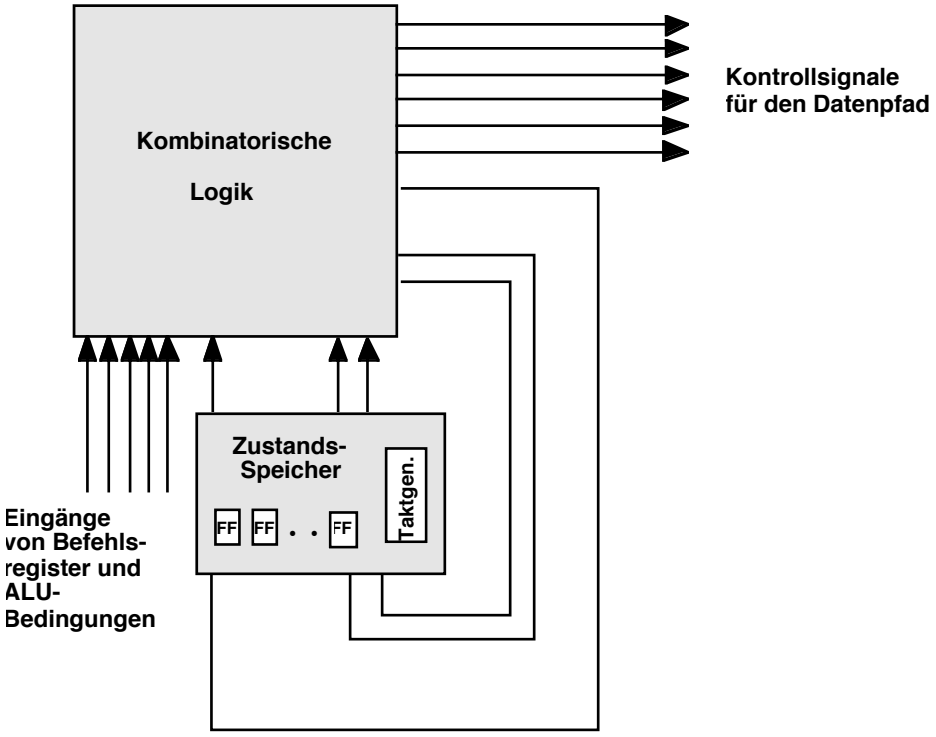


Abb. 5.1 Festverdrahtete Zustandsmaschine zur Realisierung der Kontrolleinheit

Diese Dualität von Speicher und Logik wollen wir nun weiter betrachten, indem wir alternative Ansätze zur Realisierung der Kontrolleinheit einführen. Im Zusammenhang mit der Steuerung

einer ALU haben wir im Skript "Digitale Logik" (Kapitel 6) die Grundzüge der Mikroprogrammierung kennengelernt. Diese sollen hier vertieft und auf die Kontrolleinheit einer CPU angewandt werden. Abb 5.2 stellt die Alternativen der festverdrahteten Kontrolleinheit (wie in Abb. 5.1) und der ***Mikroprogrammierten Kontrolleinheit*** (**MCU** : Microprogrammable Control Unit) gegenüber auf den Ebenen:

- grundlegende Repräsentation
- Ermittlung des Folgezustands
- Logische Repräsentation der Funktionen und
- Implementierungstechnik.

Die Spalten in Abb. 5.2 stellen den klassischen Weg des Entwurfs einer Kontrolleinheit in der jeweiligen Technik dar. Von jeder Ebene aus kann zur Erreichung der nächsten Ebene der alternative Weg verfolgt werden. Ein Beispiel haben wir bereits kennengelernt.

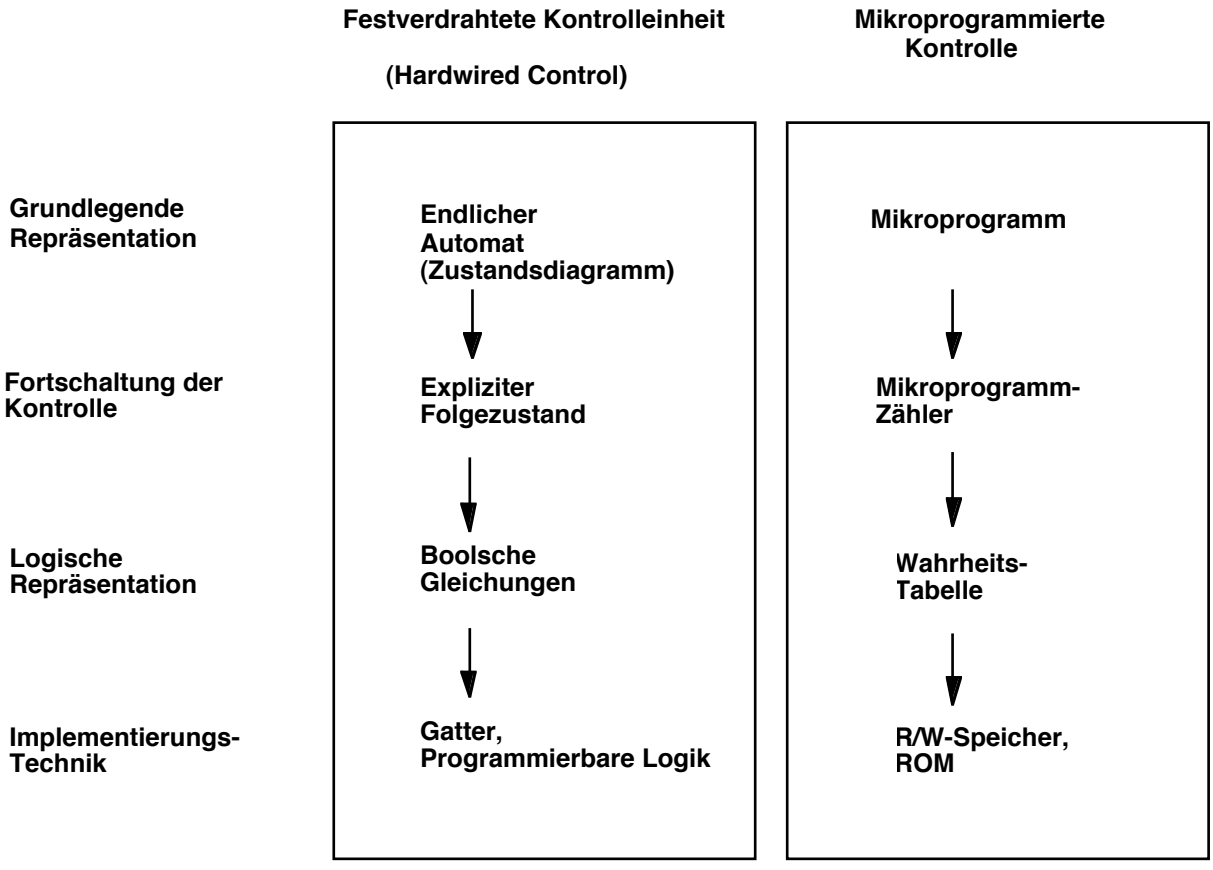


Abb. 5.2 Alternativen bei der Realisierung einer Kontrolleinheit

Der Ablauf der Maschinenbefehle und damit die Spezifikation der Kontrolleinheit unseres Modellrechners erfolgte durch elementare Operationen (ELOP), die wir als Mikroinstruktionen interpretieren können, d.h. die grundlegende Repräsentation wurde aus der rechten Spalte in

Abb. 5.2 gewählt. Die übrigen Ebenen bis zur Realisierung dagegen aus der linken Spalte von Abb. 5.2. Die Kontrolleinheit in der Darstellung als endlicher Automat ist beispielhaft für die 1-Zyklus-Instruktionen in Abb. 5.3 angegeben. Wir sehen, daß wir beim Entwurf einer Kontrolleinheit nicht auf einen bestimmten Weg festgelegt sind, sondern die für den jeweiligen Zweck geeignete Wahl treffen können.

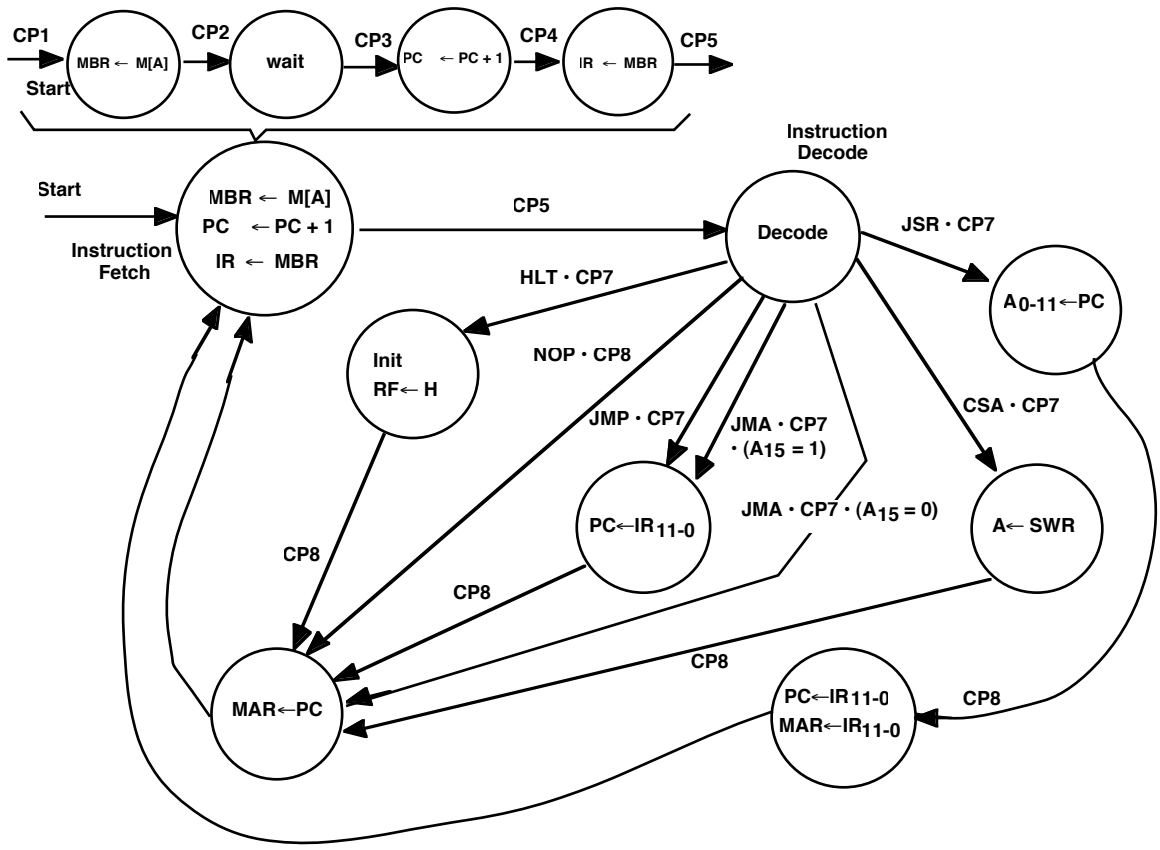


Abb. 5.3 Repräsentation der Kontrolleinheit durch einen endlichen Automaten

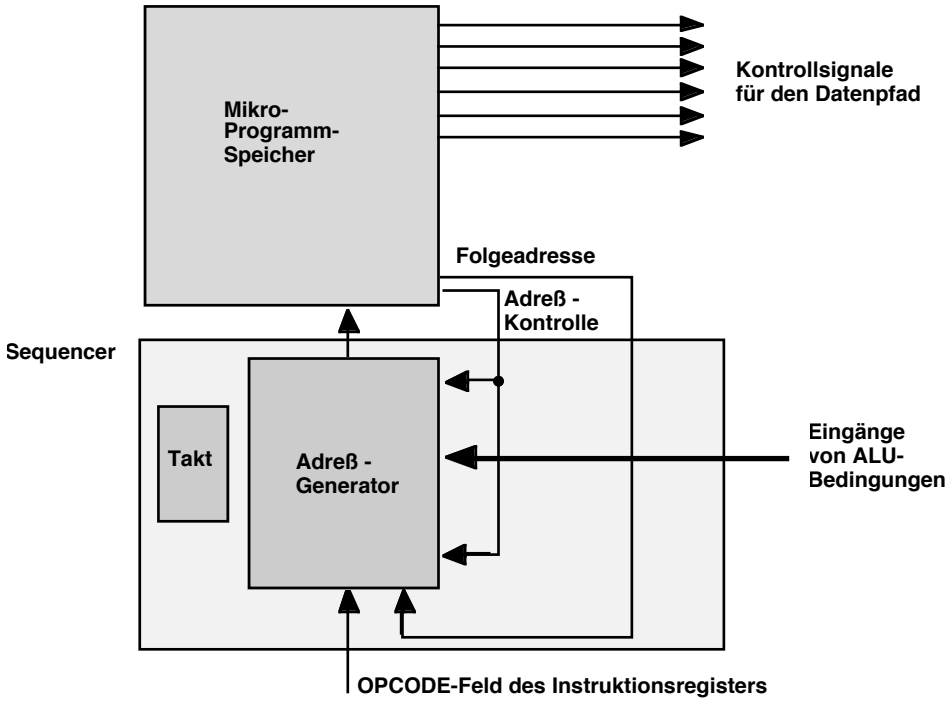
Der prinzipielle Aufbau einer mikroprogrammierten Kontrolleinheit ist in Abb. 5.4 angegeben. Der ***Mikroprogrammspeicher*** enthält das ***Mikroprogramm***, welches aus ***Mikroinstruktionen*** (**MIP: MI**cro-o**P**eration) besteht. Die Mikroinstruktionen enthalten die Information zur Erzeugung der Kontrollsignale z.B. für den Datenpfad einer CPU und Information zur Bestimmung der Adresse der nächsten MIP. Die Bildung der Adresse für die nächste auszuführende MIP ist im Detail sehr unterschiedlich für verschiedene MCUs. Grundsätzlich setzt sich diese Adresse zusammen aus:

1.) Adreßinformation aus der aktuellen MIP (Folgeadresse).

2.) Steuerinformation für den Adreßgenerator. Der Adreßgenerator wählt verschiedene Quellen aus, die zur Bildung der Adresse herangezogen werden, z.B. :

- Das OPCODE-Feld aus dem Instruktionsregister, um zu bestimmen, welche MIP-Sequenz ausgeführt werden soll.

- Zustandsinformation, z.B. aus der ALU des Datenpfads (arithmetischer Überlauf, Sprungbedingungen).



5.4 Prinzipielle Struktur einer mikroprogrammierten Kontrolleinheit

Wir wollen nun eine MCU für den Maschinenbefehlssatz unseres Modellrechners entwerfen.

5.1 Eine mikroprogrammierte Kontrolleinheit für den Modellrechner

Der Maschinenbefehlssatz unseres Modellrechners wurde bereits auf der Ebene von ELOPs beschrieben (Abb. 4.10). Wir können daher eine recht einfache Umsetzung vornehmen, indem die ELOPs jeweils als MIPs interpretiert werden. Dabei müssen wir festlegen:
1.) das Format eines Mikroprogrammwortes
2.) den Adreßfortschaltungsmechanismus.

5.1.1 Das Format eines Mikroprogrammwortes

Zunächst soll das Format eines Mikroprogrammwortes festgelegt werden. Wir wählen einen sehr direkten Weg, indem wir für jedes Steuersignal für Register, ALU, Speicher, usw. ein Bit in der MIP reservieren. Eine Steuerleitung wird aktiviert, wenn dieses Bit auf "1" gesetzt ist. Wird das Bit auf "0" gesetzt, wird die Steuerleitung deaktiviert. Deshalb wird dieser Teil des MIP auch als ***Steuerwort*** bezeichnet. Das aktuelle Mikroprogrammwort steht im

Mikroinstruktionsregister. Wir müssen also nur die Ausgänge des Mikroinstruktionsregisters mit den entsprechenden Einheiten verbinden. In Abb. 5.5 ist das Layout unseres Mikroprogrammsteuerwortes angedeutet.

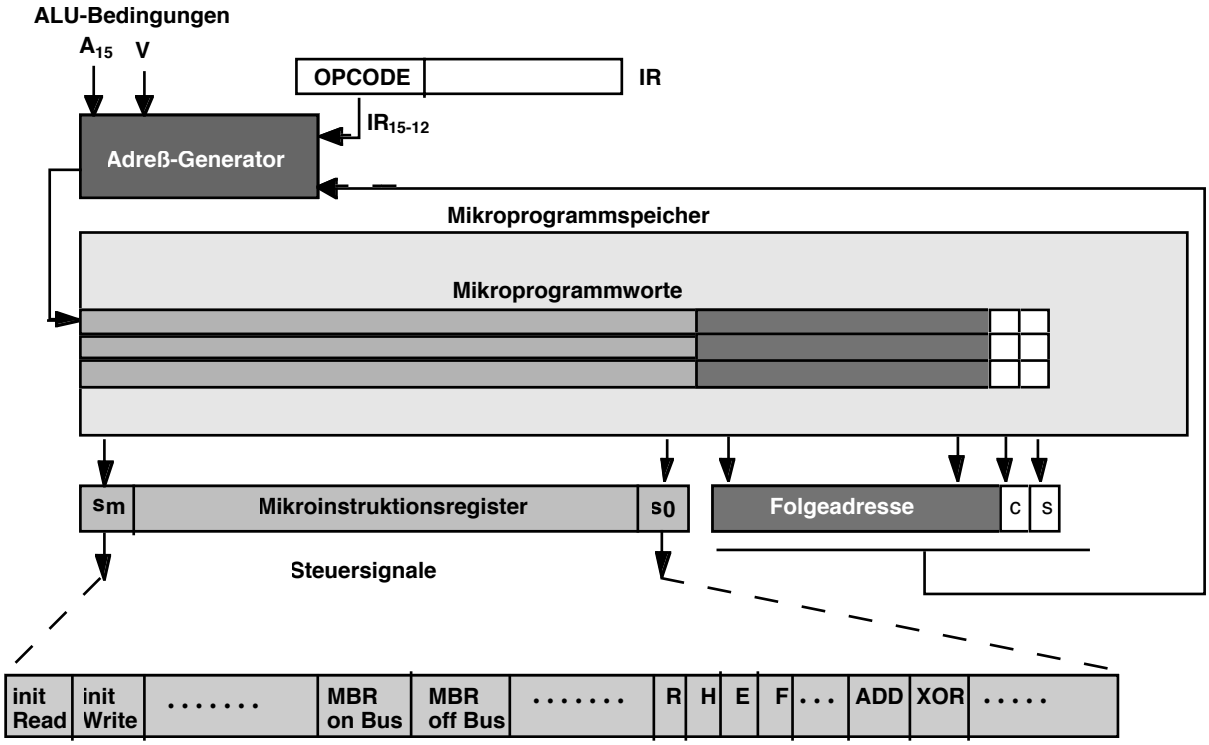


Abb. 5.5 Struktur des Mikroprogrammspeichers und des MIP-Steuerwortes

5.1.2 Adressierung des Mikroprogrammspeichers

Das Mikroprogrammwort enthält außer dem Steuerteil noch den Adreßteil, dem wir uns nun zuwenden. Auch hier wählen wir einen einfachen Weg. Der Adreßteil eines Mikroprogrammwortes enthält direkt die Adresse der nächten MIP. Allerdings muß der Modellrechner auch auf Statusinformation der ALU und auf den Inhalt des OPCODE-Feldes des IR reagieren, so daß der Ablauf des Mikroprogramms nicht immer gleich ist. Deshalb ist es notwendig:

1.) Eine bedingte Adreßfortschaltung einzuführen.
2.) Die externe Quelle der Bedingungen auszuwählen.
3.) Einen Indexmechanismus einzuführen, der es erlaubt, eine Adresse aus dem Adreßfeld der MIP und externen Adreßinformationen zu bilden.

Dazu werden 2 zusätzliche Felder im Mikrooperationswort reserviert:

c "**c**omputed next" : Die Folgeadresse wird aus der Adresse im Adreßfeld der MIP und dem OPCODE-Feld des IR_{15-12} gebildet. Dabei wird das OPCODE-Feld auf die Adresse im Adreßfeld des MIP-Wortes addiert.

s "**s**kip next on condition": Berücksichtigt wird das Vorzeichenbit des Akkumulators $A15$. Trifft die Bedingung zu, d.h. $A15 = 0$, wird die nächste Instruktion übersprungen (Folgeadresse +1). Trifft die Bedingung nicht zu, wird die nächste Instruktion von der spezifizierten Folgeadresse genommen.

Es ist zu beachten, daß keinerlei Operationscode für die MIPs definiert wurde. Die Kontrolle wird nur durch Speicherung der Steuerwortes und der Folgeadresse realisiert. Jede MIP kann als unbedingter Sprung mit der Ausgabe von Steuerinformation angesehen werden. Durch Hinzufügen weiterer Felder werden bedingte Programmverzweigungen möglich.

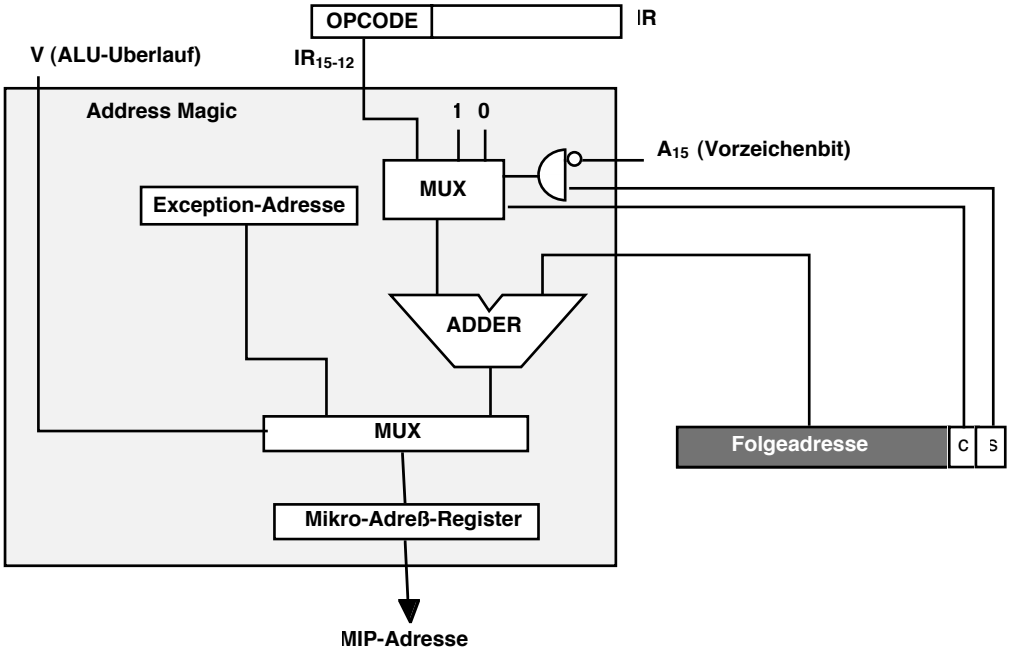


Abb. 5.6 Implementierung des Adreßgenerators

Für die MIP-Adresse gilt:

c	s	A_{15}	MIP-Adresse
0	0	x	Folgeadresse + 0
0	1	1	Folgeadresse + 0
0	1	0	Folgeadresse + 1
1	x	x	Folgeadresse+ OPCODE

x : don't care

Für die Ausnahmesituation eines arithmetischen Überlaufs muß eine gesonderte Maßnahme getroffen werden. Die CPU soll dann in den HLT-Zustand übergehen. Dies wird dadurch erreicht, daß eine feste Adresse vorgegeben wird, die im Fall eines Überlaufs selektiert wird. Der Adreßgenerator kann durch ein einfaches logisches Schaltnetz aufgebaut werden. Damit während der Adressierungsphase des Mikroprogrammspeichers die Adresse stabil anliegt, wird ein Adreßregister im Adreßgenerator vorgesehen. Abb. 5.6 gibt die Realisierung dieser Einheit an.

Abb 5.7 zeigt am Beispiel der JMA-Instruktion den grundsätzlichen Ablauf eines Mikroprogramms. Hier wird der Ablauf mit Hilfe der RTL dargestellt. Die genaue Steuerung durch die in 5.6 eingeführten Felder ist aus Gründen der Übersichtlichkeit nicht dargestellt[1].

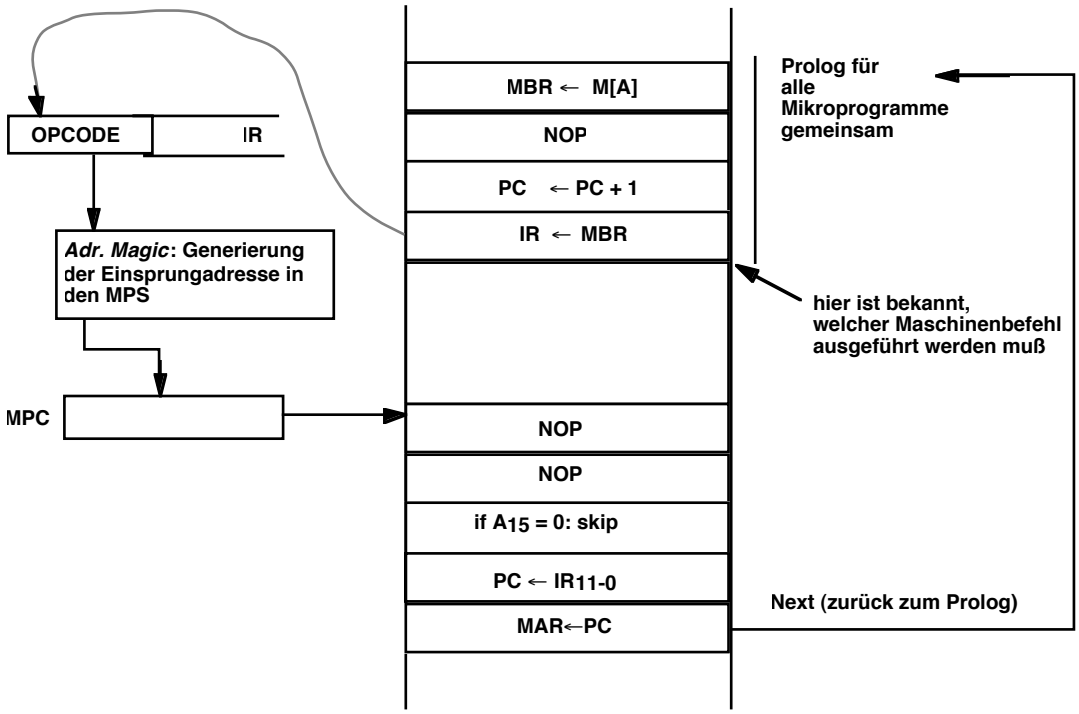


Abb. 5.7 Grundsätzliche Struktur eines Mikroprogramms am Beispiel von JMA

Alle Instruktionen haben die ersten 4 Zyklen der IF-Phase gemeinsam. In MIP 4 wird der Inhalt des MBR, der die Instruktion darstellt, in das IR transferiert. Er muß nun dekodiert werden und abhängig vom OPCODE muß das Programm verzweigen. In Abb. 5.7 ist dies für JMA gezeigt. Die Frage ist, wie die Startadresse für ein Mikroprogramm ermittelt wird, bei der ein externer Offset berücksichtigt werden muß (z.B. der OPCODE). Abb. 5.8 zeigt ein Verfahren, das auf der Indizierung einer Einsprungtabelle beruht. Der OPCODE wird dabei als Index benutzt, der auf die Basisadresse der Einsprungtabelle addiert wird.

[1] In Abb. 5.9 wird die Steuerung der Adreßfortschaltung im Detail gezeigt.

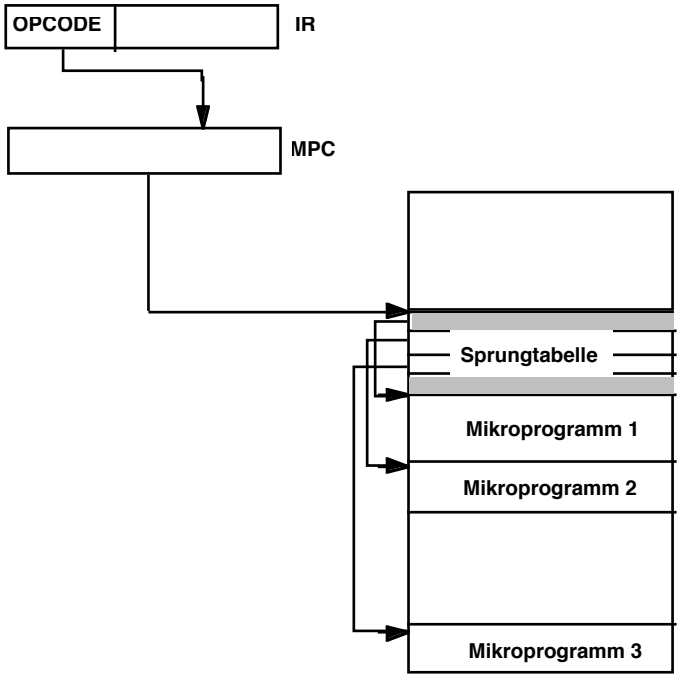


Abb. 5.8 Ermittlung der Startadresse eines Mikroprogramms

In Abb. 5.9 ist ein Mikroprogramm und seine Codierung angegeben, das auf diesem Verfahren basiert..

MPS-Adresse	Adreßauswahl c	s	Adreßteil	Steuerteil	Kommentar
00	0	0	01	Init Read	
01	0	0	02		
02	0	0	03	PC ← PC+1	
03	0	0	04	IR on Bus, MBR off Bus	
04	1	0	05		Folgeadresse + OPCODE
05	0	0	30		Startadresse für HLT
06	0	0	35		Startadresse für JMA
07	0	0	40		Startadresse für JMP
.					
.					
.					
30	0	0	31	RF ← H	
31	0	0	00	MAR off Bus, PC on Bus	Beendigung von HLT und Rücksprung
32					
33					
34					
35	0	1	36		if A_{15} = 0 THEN skip
36	0	0	37	PC off Bus, IR_{11-0} on Bus	
37	0	0	00	MAR off Bus, PC on Bus	Beendigung von JMA und Rücksprung
38					
39					
40	0	0	41	PC off Bus, IR_{11-0} on Bus	
41	0	0	00	MAR off Bus, PC on Bus	Beendigung von JMP und Rücksprung
.					
.					
.					

Abb. 5.9 Beispiel-Mikroprogramm für den Modellrechner

Wir nehmen an, daß es ein Reset gibt, der die Anfangsadresse des Mikroprogramms auf "00" setzt. Der gemeinsame Prolog der Mikroprogramme ist auf Adresse 00 - 03 gespeichert. Die Adressen 05-20 enthalten die Einsprungtabelle. In der MIP auf Adresse 04 ist das c-Bit gesetzt. Der Inhalt des OPCODE-Feldes im IR wird daher zu der im Adreßteil der MIP spezifizierten Adresse (05) addiert und bildet damit die entsprechende Adresse in der Einsprungtabelle (vgl. auch Abb. 5.6). In Abb. 5.9 werden für die Befehle HLT (0000), JMA (0001), und JMP (0010) die Realisierungen durch Mikroprogrammsequenzen gezeigt. Für den bedingten Sprung JMA wird auf Adresse 35 die ALU-Bedingung $A_{15} = 0$ abgefragt.

In Abb. 5.10 ist die Organisation des Mikroprogrammspeichers dargestellt. Insbesondere sind Felder des Steuerwortes für die Instruktionen HLT, JMA und JMP im Detail spezifiziert.

	Adr. im MPS	Adr.-Feld	Auswahl	Register-Kontrolle: MAR		MBR		IR	PC			A		Z	SWR	Status-FF: RF		SF		Speicher: Memory			ALU					
		in M-Instr.	c s	on	off	on	off	on	on	off	inc	on	off	off	on	H	R	F	E	IR	RS	WS	s0	s1	s2	s3	s4	s5
	00	01	00																	1								
	01	02	00																									
	02	03	00								1																	
	03	04	00				1	1																				
	04	05	10																									
Einsprung-Tabelle	05	30	00																									
	06	35	00																									
	•	•																										
HLT	30	31	00													1												
	31	00	00		1				1																			
	•	•																										
	•	•																										
JMA	35	36	01																									
	36	37	00					1		1																		
	37	00	00		1				1																			
	•	•																										
	•	•																										

Abb. 5.10 Layout des Mikroprogrammspeichers

Die Vorteile der Mikroprogrammierung liegen in der sehr flexiblen Steuerung der Hardwarekomponenten auf unterster Ebene. Dadurch ist es möglich, den Entwurf der Kontrolleinheit weitgehend von dem Entwurf der Kontrollalgorithmen zu entkoppeln. Würden wir zur Steuerung einen endlichen Automaten einsetzen, realisiert durch kombinatorische Logik und Flip-Flops, müssen wir sehr genau wissen, welche Befehle wir realisieren wollen und wie im Detail der Ablauf dieser Befehle auf Register-Transfer-Ebene aussieht. Eine Mikrobefehlssteuerung können wir dagegen entwerfen, ohne den genauen Befehlssatz, den sie realisieren soll, zu kennen. Demzufolge kann auch der detaillierte Ablauf der Befehle später festgelegt und als Mikroprogramm im Mikroprogrammspeicher abgelegt werden. Durch

Änderung des Mikroprogramms können wir völlig verschiedene Befehlssätze auf derselben Register-Transfer-Struktur implementieren. Wir kommen in Abschnitt 5.2.3 noch einmal auf diese Eigenschaft zurück.

Ein weiterer Vorzug der Mikroprogrammierung ist die inhärente Möglichkeit der Kontrolle paralleler Vorgänge in der CPU. Nehmen wir an, es gibt mehrere Verarbeitungseinheiten in der CPU, wie in Abb. 5.11 angedeutet.

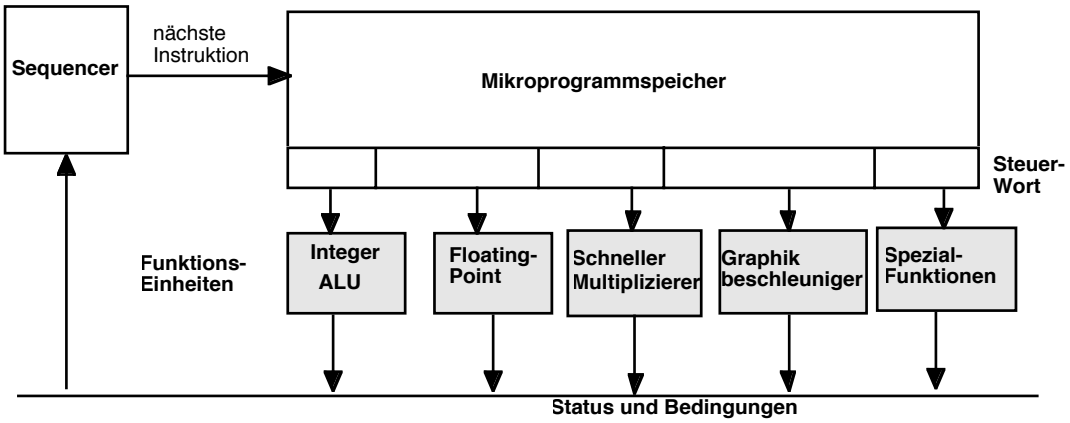


Abb. 5.11 Steuerung paralleler Verarbeitungseinheiten durch ein Mikroprogramm

Durch die Felder im Mikroprogrammsteuerwort kann jede dieser Verarbeitungseinheiten unabhängig gesteuert werden. Dadurch ist es einmal möglich, komplexe Befehle bereitzustellen, die mehrere Operanden gleichzeitig bearbeiten, oder Parallelverarbeitung auf der Ebene der Maschinenbefehle durchzuführen, indem mehrere Maschinenbefehle gleichzeitig ausgeführt werden. Die zentrale Kontrolle dieser Vorgänge wird durch das Mikroprogramm durchgeführt.

Die korrekte Programmierung und zeitliche Synchronisation komplexer Vorgänge mit Hilfe der einfachen direkten Steuerung durch ein Mikroprogrammwort ist allerdings sehr schwierig. Wir wollen uns daher nun mit Verfahren befassen, welche die Realisierung komplexerer Mikroprogramme ermöglichen.

5.2 Optimierungen der Mikroprogrammierung

5.2.1 Vertikale Mikroprogrammierung

Der Ansatz zur Mikroprogrammierung, den wir bisher diskutiert haben, stellt für jedes Steuersignal ein Bit im Mikroinstruktionswort zur Verfügung. Ein solches Format wird als ***horizontales Mikroinstruktionsformat*** bezeichnet. Entsprechend spricht man auch von horizontaler Mikroprogrammierung. Es ist leicht einzusehen, daß man für diesen Ansatz sehr lange Speicherworte und demzufolge auch einen großen Speicher benötigt. Darüberhinaus, wie

auch aus Abb. 5.10 entnommen werden kann, ist der Speicher nur dünn mit "1" besetzt. Nehmen wir z.B. die Steuerleitungen der ALU. Es kann jeweils nur eine Steuerleitung aktiviert sein. Wir stellen aber 6 Bit des Speichers zu deren individueller Auswahl zur Verfügung. Würden wir hier eine Codierung einsetzen, reichten 3 Bit aus. Ähnlich verhält es sich mit der Ansteuerung der Register. Wir müssen nicht unbedingt die Flexibilität haben, alle Register individuell zu steuern. Besonders bei einer größeren Anzahl von Registern ist die Auswahl der Register über eine geeignete Registeradressierung sinnvoll.

Bei der sogenannten ***vertikalen Mikroprogrammierung*** werden diese Überlegungen berücksichtigt, indem einzelne, sich ausschließende Steuersignale codiert und in Feldern zusammengefaßt sind. Eine anschließende Dekodierungsstufe erzeugt die notwendigen individuellen Kontrollsignale. Abb. 5.12 zeigt eine mikroprogrammierte Steuereinheit, die dieses Konzept umsetzt.

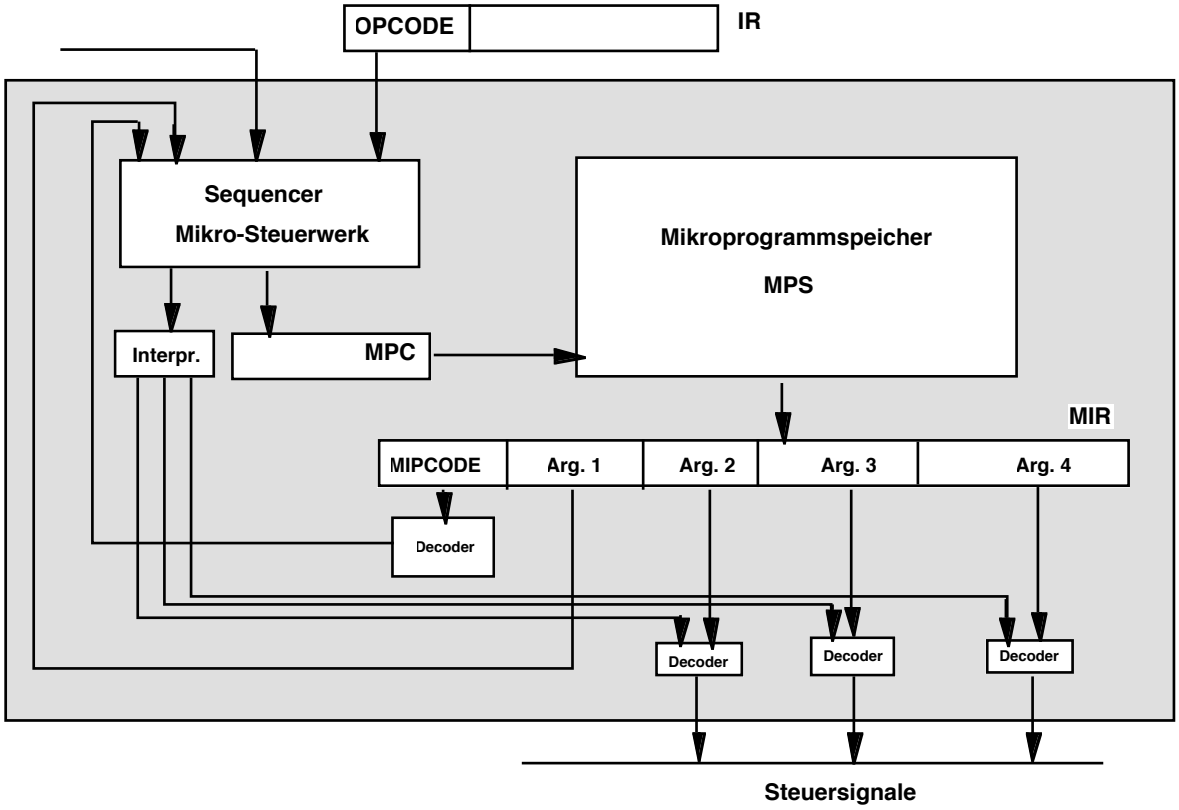


Abb. 5.12 Mikroprogrammierung mit einstufiger Dekodierung (vertikale MP)

Vertikale Mikroprogrammierung ist gekennzeichnet durch:

- **Umfangreiche und komplexe Mikroinstruktionen.** Diese sind notwendig, um eine (bedingte) Auswahl der Argumentfelder zu treffen und die Adreßfortschaltung zu kontrollieren.
- **Komplexere Adreßfortschaltung.** Es wird nicht bei jeder MIP eine vollständige Folgeadresse angegeben, sondern es wird ein Programmzähler eingeführt, der

standardmäßig inkrementiert wird. Dadurch müssen bedingte und unbedingte Sprünge realisiert werden. Der Vorteil liegt in dem geringeren Speicherbedarf, da nicht bei jeder Instruktion eine vollständige Folgeadresse angegeben werden muß.

- **Codierung der Steuerinformation.** Das Steuerwort wird in Felder eingeteilt, die jeweils eine bestimmte Rechnerkomponente oder Gruppen von Komponenten steuern. Beispiele für die Felder sind:
  - ALU-Felder. Ein Feld enthält die codierte ALU-Funktion. Um mehrere ALUs zu steuern, können mehrere dieser Felder existieren.
  - Quellregister-Feld enthält eine Registeradresse
  - Zielregisterfeld 1,..., n. Dies können mehrere Felder sein, die Registeradressen enthalten, so daß der Inhalt eines Quellregisters gleichzeitig in mehrere Zielregister übertragen werden kann.

Generell gilt für vertikale Mikroprogrammierung eine immer stärkere Annäherung an die Programmierung in Maschinensprache, d.h. es wird immer mehr eine "CPU in der CPU". Die Merkmale sind:

- Kompaktere Codierung der Befehle führt zu Einsparung von Speicher.
- Komplexität des "Sequenzers" wird dadurch erhöht.
- Mikroinstruktionen erreichen die Komplexität von Maschinenbefehlen.
- Erstellung von Mikrocode ist einfacher und weniger fehleranfällig.

5.2.2 Nanoprogrammierung

Die vertikale Mikroprogrammierung "vergibt" allerdings einen Vorteil, den die horizontale Mikroprogrammierung hatte: die Möglichkeit der unabhängigen Erzeugung beliebiger Kontrollsignalsequenzen und dadurch die Entkopplung des Entwurfs der Kontrolleinheit und der Festlegung der Mikroinstruktionen. Die feste Aufteilung der Mikroinsruktion in Felder und die feste Dekodierung dieser Felder sind die Ursache dafür. Dafür hat man den Vorteil eines kompakteren Mikrocodes und der einfacheren Erstellung der Mikroprogramme. Ein Ansatz, der versucht die Vorteile horizontaler und vertikaler Mikroprogrammierung miteinander zu verbinden, ist die ***Nanoprogrammierung***. Abb. 5.13 zeigt ein einfaches Verfahren dazu. Anstelle fester Steuerfelder und eines festen Dekoders für das Mikroinstruktionswort wird ein zusätzlicher, sogenannter ***Nanospeicher*** eingesetzt. In diesem Speicher ist ein Steuerwort abgelegt, das analog zum horizontalen Mikroprogrammsteuerwort die Erzeugung beliebiger Kontrollsignalkombinationen ermöglicht. Die vertikale Mikroinstruktion generiert einen Index in den Nanospeicher. Es ist zu beachten, daß in diesem Ansatz das Nanoprogramm selbst

keinerlei Adreßfortschaltungsmechanismen besitzt. Es stellt lediglich einen programmierbaren Dekodierer dar und es besteht eine Eins-zu-Eins-Zuordnung zwischen vertikalem Mikrobefehl und dem entsprechenden Steuerwort im Nanospeicher.

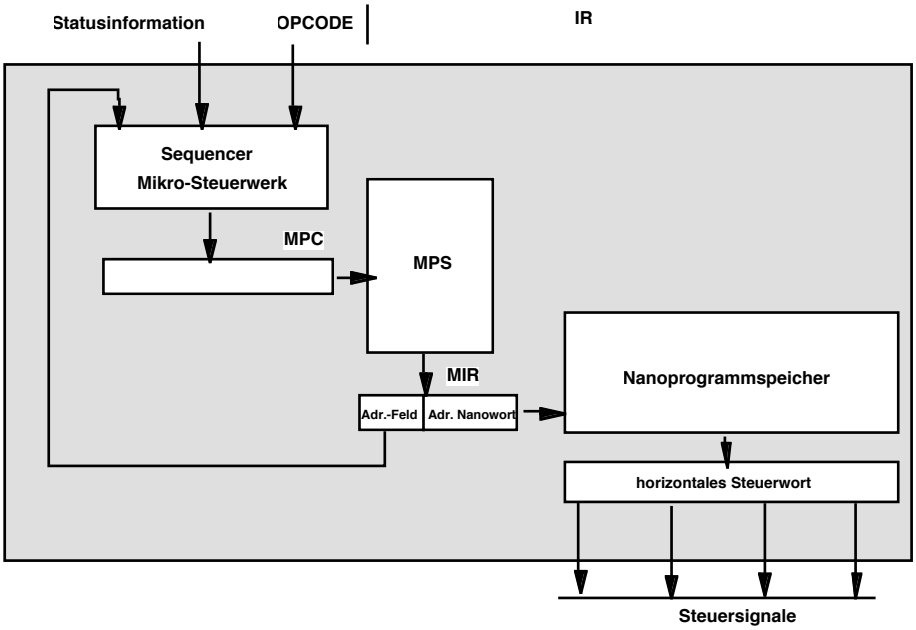


Abb. 5.13 Erzeugung der Steuersignale mit Nanoprogrammierungspeicher

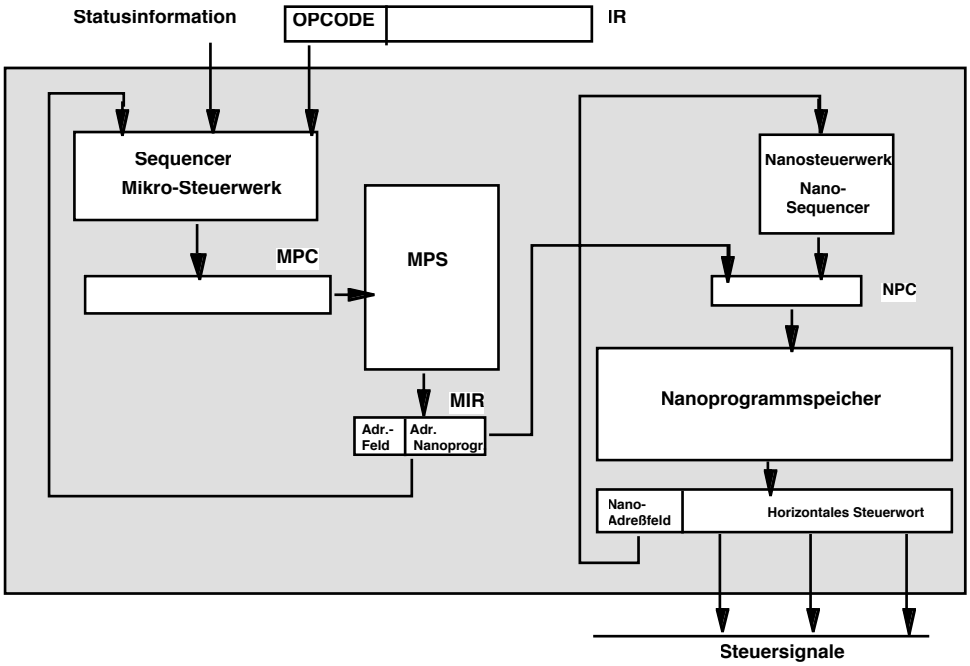


Abb. 5.14 Echte zweistufige Kontrollhierarchie durch Nanosteuerwerk

Eine echte zweistufige Kontrolle ist in Abb. 5.14 gezeigt. Hier generiert die vertikale Mikroinstruktion die Startadresse eines Nanoprogramms, das unter der Kontrolle des

Nanosequenzers abläuft. Der Vorteil hierbei ist, daß bestimmte Kontrollsequenzen, die für mehrere vertikale Mikroinstruktionen gleich sind, zusammengefaßt werden können.

Zusammenfassend gelten folgende Punkte für die Nanoprogrammierung:

- Nanoprogrammierung ist eine Kombination von vertikaler und horizontaler Mikroprogrammierung.
- In der Nanoprogrammierung wird die Dekodierungshardware des vertikalen Ansatzes durch eine weitere Stufe der Mikroprogrammierung ersetzt.

5.2.3 Hierarchie von Maschinen und vertikale Verlagerung

Wir wollen kurz einige grundsätzliche Überlegungen zur Verwendung der Mikro- und Nanoprogrammierung anstellen. Dazu soll Abb. 5.15 herangezogen werden. Sie skizziert grob den Zusammenhang zwischen einem Programm, das in einer Sprache *L* geschrieben ist und einer Maschine, die diese Sprache interpretiert.

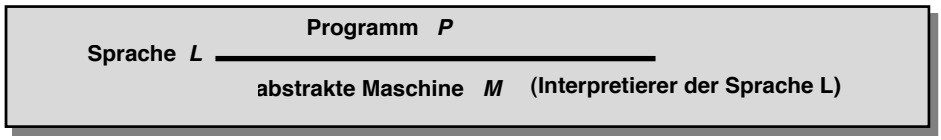


Abb. 5.15 Zusammenhang zwischen Programm, Sprache und Interpreter

In einem Rechner ist meist eine Hierarchie solcher abstrakter Maschinen realisiert. Abstrakt deshalb, weil die Realisierung einer Maschine keine Rolle spielt, sondern nur die von ihr zur Verfügung gestellte Sprache *L* , definiert durch die Instruktionen, die sie interpretieren kann.

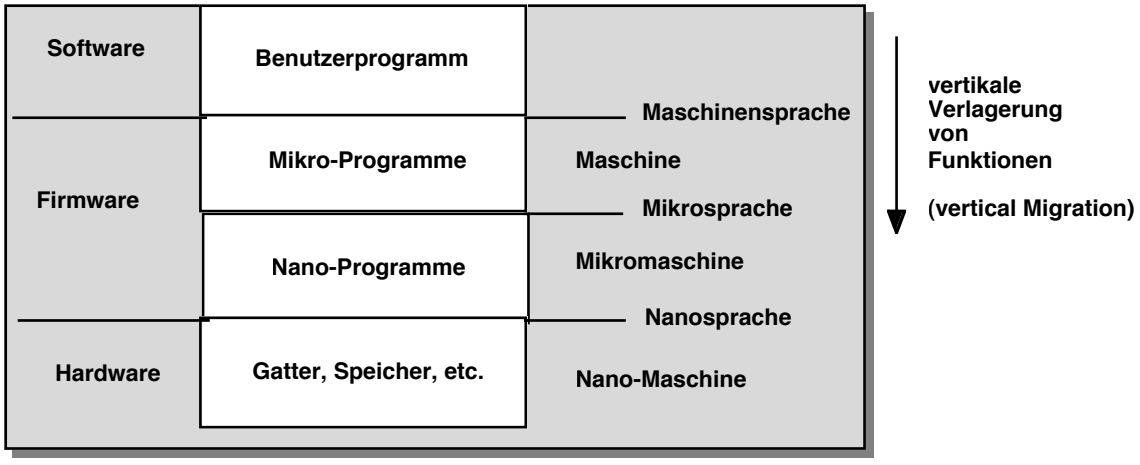


Abb. 5.16 Hierarchie der Interpreter in einem System

Einige dieser Ebenen, die wir bereits kennengelernt haben, sind in Abb. 5.16 dargestellt. Nur auf der untersten Ebene ist die Maschine tatsächlich durch Hardware realisiert. Die Maschinen der oberen Ebene sind durch Programme in der jeweiligen Sprache formuliert, die von der

Maschine der jeweiligen darunterliegenden Ebene interpretiert wird. Durch diese Form der Schichtung kann man sehr komplexe Befehlssätze auf der Ebene der Maschinensprache durch eine Hierarchie einfacherer Maschinen zur Verfügung stellen. Es liegt auf der Hand, daß der Entwurf der Rechnerarchitektur, gegeben durch den Maschinenbefehlssatz, und der Entwurf der Hardware weitestgehend voneinander getrennt sind.

Komplexe Funktionen können anstelle einer Sequenz von Maschinenbefehlen als einzelner Befehl zur Vefügung gestellt werden, indem die darunterliegende Ebene diesen Befehl durch ein Mikroprogramm realisiert. Die Verlagerung von Funktionen aus einer höheren Ebene in der Architektur der unterliegenden Maschine wird als ***vertikale Verlagerung*** bezeichnet. Die Hauptmotivation für die vertikale Verlagerung ist die Annahme, daß die Effizienz der Realisierung auf tieferen Ebenen höher ist.

5.3 Diskussion

Heute sind viele der Gründe, die zur Einführung der Mikroprogrammierung führten, nicht mehr gültig. Insbesondere das Verständnis, daß mächtige Maschinenbefehle die Programmierung erleichtern, hat sich geändert. Die Mikroprogrammierung geht von folgenden Prämissen aus:

1.) Mächtige Maschinenbefehle unterstützen den Programmierer.
2.) Leichte Änderbarkeit der Maschinenbefehle ist notwendig, um die Fortentwicklung des Befehlssatzes und Spezialanwendungen zu unterstützen.
3.) Mikroprogramme können leichter entworfen werden als Logik.
4.) Mikroprogramme können komplexe Operationen effizienter bereitstellen als eine Sequenz von Maschineninstruktionen, da sie die Parallelität der Hardware ausnutzen können.
5.) Mikroprogramme können komplexe Operationen effizienter bereitstellen als eine Sequenz von Maschineninstruktionen, da Mikroprogrammspeicher schneller ist als Speicher für Maschinenprogramme.

Demgegenüber steht:

1.) a.) Rechner werden heute (fast) nur noch in Hochsprachen programmiert. Für Compiler haben sich mächtige Maschinenoperationen als problematisch herausgestellt.
b.) Mächtige Maschinenoperationen benötigen große Mikroprogrammspeicher, bzw. mehrstufige Verfahren der Nanoprogrammierung. Dies führt nicht nur zu komplexen und damit teuren Steuerwerken, sondern auch dazu, daß sich die Ausführungszeit einer Mikroinstruktion verlängert durch mehrfache

Speicherzugriffe und Verzögerungen in den Dekodern. Bei komplexen Mikrooperationen ist dieser Aufwand vielleicht gerechtfertigt. Einfache Operationen können auch mit einfacheren und damit schnelleren Kontrolleinheiten realisiert werden. Ein kompliziertes Steuerwerk wirkt sich negativ auf die Leistung einfacher Maschinenoperationen aus.

2.) a.) Die Kontrolleinheit ist in einem Microprocessor "on-chip" realisiert. Dies hat zur Folge, daß Mikroprogramme nicht mehr einfach verändert werden können (z.B. durch Austauch des ROM, in dem sie gespeichert sind)

b.) Weitere Entwicklungen, durch die die Änderbarkeit des Mikroprogramms an Bedeutung verliert:

- Ein neuer Befehlssatz bedeutet, daß die (Betriebssystem- und Anwendungs-) Software angepaßt werden muß. Diese Anpassung von verfügbarer Software an einen neuen Befehlssatz ist ein erheblicher Kostenfaktor.
- Die Maschinenbefehlssätze werden immer ähnlicher und einfacher. Komplexe Spezialbefehle sind im Maschinenbefehlssatz der CPU nicht mehr vorhanden.
- Spezialbefehle werden durch Einheiten mit eigener Kontrolle ausgeführt (z.B. Coprozessoren).

3.) Programmierbare Logik weist eine ähnlich reguläre Struktur auf wie ein ROM. Sie ist oft effizienter in der Implementierung von Kontrollstrukturen als ein Mikroprogramm. Der Entwurf und die Änderung programmierbarer Logik zur Realisierung einer Kontrolleinheit ist, bedingt durch geeignete Entwurfswerkzeuge, nicht schwieriger oder fehleranfälliger als das Schreiben eines Mikroprogramms.

4.) Es ist richtig, daß die Mikroprogrammierung die inhärente Parallelität von Hardware optimal nutzen kann. Allerdings sollten wir bedenken, daß insbesondere in komplexen (nanoprogrammierten) Steuerwerken die erste Ebene der Mikroprogrammierung ebenfalls stark sequentialisiert ist. Andererseits können CPUs so strukturiert werden, daß mehrere einfache Maschinenbefehle parallel ausgeführt werden können. Dadurch relativiert sich der Vorteil der Parallelitätsausnutzung auf der Mikroprogrammebene. Die entsprechenden Techniken der parallelen Bearbeitung auf der Maschinenbefehlsebene werden wir bei der Diskussion der RISC-Prozessoren näher betrachten.

5.) Wir müssen heute davon ausgehen, daß eine CPU vollständig auf einem VLSI-Chip realisiert ist. Für die Geschwindigkeit von Speicher auf dem Chip gilt, daß Mikroprogrammspeicher nicht schneller ist als Instruktions- und Datenspeicher für

Maschinenbefehle, wenn sich dieser ebenfalls auf dem Chip befindet. Sogenannte Caching Techniken, die wir noch genauer bei der Behandlung von Speicher untersuchen werden, erlauben den Zugriff auf Maschinenbefehle mit der gleichen Geschwindigkeit, in der ein Zugriff auf Mikrobefehle möglich wäre.

Obwohl die Mikroprogrammierung bei modernen RISC-Prozessoren ihre Bedeutung verloren hat, wird sie weiter im Bereich der Großrechner intensiv genutzt. Darüberhinaus bleibt sie ein notwendiges Gedankenexperiment zum Verständnis der Abläufe in der CPU als eine Alternative im Entwurfsprozeß einer Kontrolleinheit.

6 Assembler

Abb 6.1 zeigt den Maschinencode für das Beispielprogramm "Rotate Right" aus Kapitel 4 in Binärdarstellung, Hexadezimaldarstellung und in Oktaldarstellung. In der linken Spalte sind die Speicheradressen angegeben, in der rechten Spalte steht der Maschinencode. Die hexadezimale und die oktale Darstellung sind lediglich verkürzte Notationen der Binärdarstellung, die aber die Lesbarkeit des Maschinencodes nicht verbessern. Das Beispiel soll als natürliche Motivation zur Einführung eines Assemblers dienen.

Sp.- Adr	OPC OP-Adr.	HEX	Octal
00010000	1001 10000001	10 9 81	020 4601
00010001	1000 00000000	11 8 00	021 4000
00010010	1011 10000010	12 B 82	022 5602
00010011	1011 10000100	13 B 84	023 5604
00010100	1010 10000011	14 A 83	024 5203
00010101	1001 10000000	15 9 80	025 4600
00010110	0101 00000000	16 5 00	026 2400
00010111	1010 10000000	17 A 80	027 5200
00011000	1001 10000010	18 9 82	030 4602
00011001	1000 00000000	19 8 00	031 4000
00011010	1011 10000010	1A B 82	032 5602
00011011	1011 10000011	1B B 83	033 5603
00011100	0001 00011110	1C 1 1E	034 0436
00011101	0010 00010101	1D 2 15	035 1025
00011110	0000 00000000	1E 0 00	036 0000

Abb. 6.1 Maschinencode für das Beispielprogramm "Rotate Right" (vgl. Abb. 4.6)

Ein Assembler ermöglicht (im strengen Sinne "erleichtert") die Programmierung eines Rechners auf der Ebene seiner Maschinenbefehle im Wesentlichen durch:

- Verwendung symbolischer Namen anstelle von Binärzahlen
- Durchführung von Adreßberechnungen

Wir haben schon in Kapitel 4 zur Darstellung der Beispielprogramme eine Assemblernotation der Maschinenprogramme eingeführt. Die Elemente der Assemblernotation sind:

1. Mnemotechnische Bezeichnungen für Maschinenbefehle
2. Symbolische Namen für:
   - Speicherplätze
   - Konstanten

- Speicherfelder (aufeinanderfolgende Speicherplätze) z.B. für Arrays oder Listen
- Textstrings
- Sprungmarken

Ein Assembler ist ein Übersetzer, der diese Notation in Maschinencode übersetzt. Abb. 6.2 skizziert den Übersetzungsvorgang vom Quelltext bis zum ausführbaren Maschinencode im Speicher.

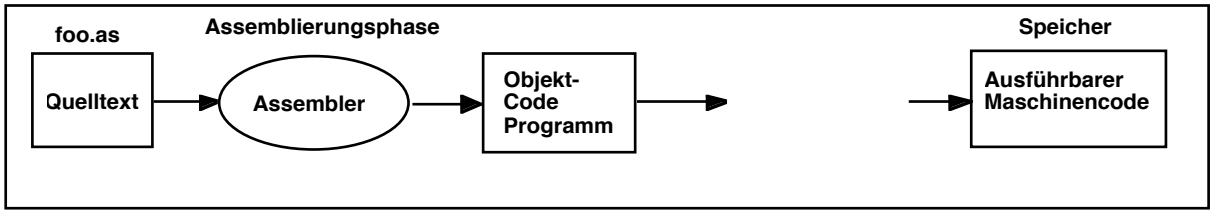


Abb. 6.2a Ablauf der Übersetzung eines Assemblerprogramms

Der Assembler übersetzt den Quelltext und erzeugt ein den Objektcode eines Programms, der den Maschinencode enthält, aber noch nicht an bestimmte Speicheradressen gebunden ist. Der Objektcode enthält noch Referenzen, z.B.Referenzen relativ zum Anfang des Programms, denen keine festen Speicheradressen zugeordnet sind. Erst wenn das Programm in den Speicher geladen wird, werden alle Referenzen aufgelöst und festen Speicheradressen zugeordnet[1]. Diese Aufgabe wird durch den Lader durchgeführt. Der Lader wird auch als "Relocation Loader" bezeichnet, weil er ein Objektprogramm an verschiedenen Stellen im Arbeitspeicher ablegen kann. Die Trennung der Assemblierung von der Auflösung der Referenzen eines Progarmms hat aber noch einen weiteren Vorteil. Es wird dadurch möglich, externe Bibliotheksprogramme, die bereits assembliert sind, vor der Lade-Phase zu einem neuen Programm dazuzubinden. Dies wird durch den Binder realisiert. Der Binder muß dazu auch sogenannte externe Referenzen, die auf Variablen oder Einsprungmarken in anderen Programmen zeigen, in relative Referenzen umwandeln, die anschließend vom Lader festen Speicheradressen zugeordnet werden. Ein erweitertes Bild eines Assemblers, in dem mehrere Quellprogramme zusammen assembliert werden können, ist in Abb. 6.3 dargestellt.

Bei vielen einfachen Assemblern sind Assembler und Lader nicht getrennt, sondern es wird aus dem Quelltext direkt ein ausführbares Maschinenprogramm erzeugt. Damit entfällt auch der Binder und damit die Möglichkeit, Programmbibliotheken zu nutzen.

[1]In vielen kleineren Systemen sind Assembler und Binder/Lader in einem Programm zusammengefaßt.

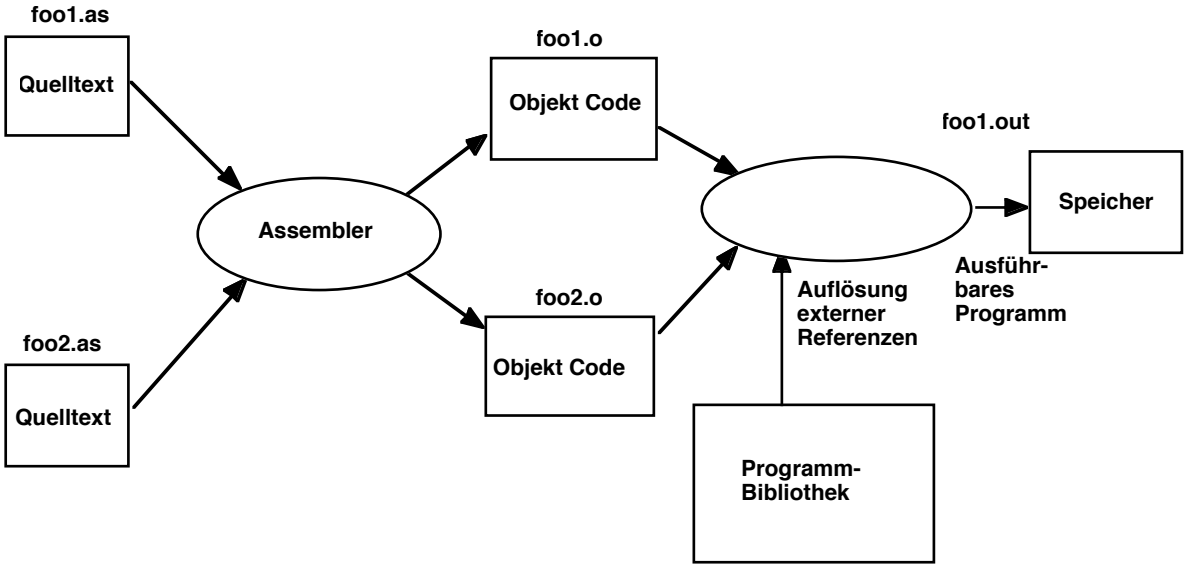


Abb. 6.3 Erweiterter Assembler

Wie bei einem Compiler, der Hochsprachen übersetzt, müssen bei einem Assembler bestimmte Konventionen eingehalten werden. Allerdings sind die Assemblernotationen und die einzuhaltenden Konventionen bei jeder Maschine unterschiedlich. Üblich ist sogar, daß von verschiedenen Assemblern für eine Maschine jeder eine unterschiedliche Notation und andere Konventionen unterstützt.

Abb. 6.4 zeigt die generellen Felder eines Assemblerprogramms. Der rechte Teil von Abb. 6.4, der Markenfeld, Opcodefeld, Operandenfeld und Kommentarfeld enthält, stellt das Quellprogramm dar und wird vom Programmierer spezifiziert. Der linke Teil enthält die Repräsentation des übersetzten Programms, dem bereits Speicheradressen zugeordnet wurden.

Felder im Assembler Programm:

Zeile Nr.	Sp. Adr.	Opc.	Operand	Markenfeld (Labels)	Opcode-Feld	Operanden Feld	Kommentarfeld
1	1000	9	1100	Start	LDA	VAL1	Lade ersten Summanden
2	1001	B	1101		ADD	VAL2	Addiere zweiten Summanden
3	1003	A	1102		STA	SUM	Speichere die Summe
4	1004	3	xxxx		RTS		Zurückspringen
	.				.		
	.				.		
	1100			VAL1	RMB	1	Reserve (1) Memory Byte
	1101			VAL2	RMB	1	Reserve (1) Memory Byte
	1102			SUM	RMB	1	Reserve (1) Memory Byte

Abb. 6.4 Felder eines Assemblers

- Das Markenfeld enthält Sprungmarken oder Marken für zur Assemblierungszeit festgelegte Speicherplätze für Variable und Konstanten (vergl. Abschnitt 6.2).

- Das OPCODE-Feld enthält den mnemotechnischen Code der Maschinenoperation.
- Das Operandenfeld enthält die Operandenspezifikation, z.B. die Adresse des Operanden.
- Das Kommentarfeld ist optional und enthält Information, um die Lesbarkeit von Programmen zu verbessern.

Die Felder sind durch Leerzeichen getrennt.

Ein Assembler unterstützt die Programmierung in folgenden Punkten:
- Berechnung von Speicheradressen bzw. Distanzadressen für
  - Sprungziele
  - Variablen und Konstanten
- Berechnung aller absoluten Referenzen bezüglich einer Programm-Startadresse
- Verwendung mnemotechnischer Bezeichnungen für Befehle
- Umwandlung verschiedener Zahlenbasen
- Reservierung von Speicherplatz

Die Konvention zur Kennzeichnung der Zahlenbasen ist in Abb. 6.5 angegeben. Die Zahlen werden vom Assembler in die entsprechende Binärdarstellung im ausführbaren Programm umgewandelt. Weitere Notationen, insbesondere zur Kennzeichnung verschiedener Adressierungsarten werden in den folgenden Kapiteln eingeführt.

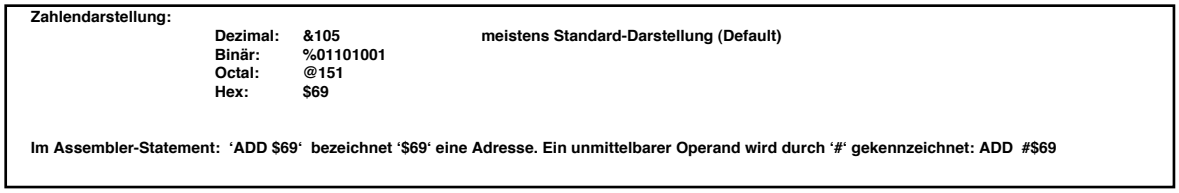


Abb. 6.5 Zahlendarstellung im Assembler Quelltext

Der Assembler führt auch die Berechnung durch für:
- Sprungziele, die durch Marken gekennzeichnet sind, und
- Variablen oder Konstanten, die im Markenfeld durch symbolische Namen gekennzeichnet sind. Im Beispiel von Abb. 6.4 wird für die Variablen Val1, Val2 und Sum eine absolute Adresse eingesetzt.

6.1 Assemblerdirektiven

Assemblerdirektiven sind Statements, die nicht direkt ausführbarem Code entsprechen, sondern zur Steuerung der Übersetzung dienen. Assemblerdirektiven sind sehr stark vom jeweiligen

Assembler abhängig. Hier werden die gängigen Direktiven des Motorola Assemblers für den MC6809 aufgeführt[2]. Allgemein dienen Assemblerdirektiven der:

1.) Zuordnung von Werten zu symbolischen Namen
2.) Definition der Speicheradresse, an die das Programm geladen werden soll
3.) Reservierung von Speicherplatz

1. Zuordnung von Werten zu symbolischen Namen:

EQUATE oder **DEFINE**	name	EQU	number
	name	EQU	name

Beisp.:	VAL	EQU	$A000
	UNIVERSE	EQU	42
	ALPHA	EQU	FIRSTCHAR

2. Definition der Adresse, an die z.B. ein Programm geladen werden soll. Das ORG-Statement ist immer notwendig, um ein Programm oder einen Programmteil in den Speicher zu laden. Alle Adressen eines Programms werden relativ zu der im ORG-Statement angegebenen Adresse berechnet.

ORIGIN	ORG	name
	ORG	number

Beisp.:	ORG	START
	ORG	$C010

3. Reservierung von Speicherplatz :

RESERVE	RMB	number

Die RMB Direktive reserviert eine Anzahl von Bytes.

Beisp.:	BUF	RMB	256

[2] Der MC6809 wird uns im Kapitel 7 als Beispiel für einen 8-Bit Mikroprozessor dienen.

```
DATA        FCB   number
            FDB   number
            FCC   string (ASCII)
```

Die FCB Direktive belegt ein Byte mit einem bestimmten Wert (Konstante).
Beisp.: MAXVAL FCB 100

Die FDB Direktive belegt ein 16-Bit-Wort (Doppel-Byte) mit einem bestimmten Wert.
Beisp.: CHECKPAT FDB %10101010 10101010

Die FCC Direktive belegt ein oder mehrere aufeinanderfolgende Bytes mit ASCII-Zeichen. es ist zu beachten, daß die Zeichen in Hochkomma eingeschlossen sind.
Beisp.: TEXT FCC 'Wo soll das alles enden?'

6.2 Weitere Eigenschaften von Assemblern

1.) Macroassembler

Ein Macro besteht aus einem Namen und einer Folge von Assembleranweisungen. Findet der Assembler im Quelltext den Namen eines Macros, ersetzt er ihn durch die zugehörige Folge von Anweisungen.

2.) Bedingte Assemblierung

Bedingte Assemblierung erlaubt es, verschiedene Versionen eines Programms, abhängig von bestimmten Bedingungen zu erstellen. Beispielsweise soll ein Programm für die Testphase mit zusätzlichen Ausgaben, Überprüfungen, etc. erzeugt werden. Die entsprechenden Assemblerfragmente werden im Quelltext von den: IF CONDITION .... ENDIF Statements eingeklammert. Wenn die Bedingung zutrifft, werden die Assembleranweisungen zwischen IF und ENDIF mitübersetzt, trifft die Bedingung nicht zu, werden sie bei der Assemblierung ignoriert.

```
IF CONDITION = TRUE
•
• Code wird zwischen IF und ENDIF in das
•  Maschinenprogramm eingefügt
•
ENDIF
```

3.) Assemblertypen:

Residenter Assembler:
Ein residenter Assembler wird auf der Maschine ausgeführt, für die er auch den Maschinencode erzeugt, d.h. die Wirtsmaschine des Assemblers und die Zielmaschine, auf der der erzeugte Code ausgeführt wird, sind gleich.

Cross-Assembler
Ein Cross-Assembler läuft auf einer (meist leistungsfähigen) Wirtsmaschine und produziert Code für eine andere Zielmaschine. Der Grund ist, daß die Zielmaschine oft nicht über eine komfortable Programmierumgebung verfügt, wie z.B. Textverarbeitung/Filesystem/Simulator etc.. Insbesondere, wenn die Zielmaschine eine sehr einfache CPU für kleine Kontrollaufgaben darstellt, kann sie eine leistungsfähige Programmierumgebung nicht zur Verfügung stellen.

6.3 Abschließende Bemerkung

Ein Assembler ist ein sehr maschinennahes Programmierwerkzeug. Seine generellen Vor- und Nachteile sind stichpunktartig im Folgenden aufgelistet.

Vorteile des Assemblers:

- er reflektiert die Architektur des Rechners,
- er gibt genau an, was im Rechner geschieht, d.h. 1:1 Abbildung von Programmstatement und Maschinenoperation,
- Vorhersagbarkeit (Predictability) für zeitkritische Anwendungen, z.B. in Steuerungssystemen,
- Kompakter Code.

Aussagen über Struktur und Leistungsfähigkeit eines Prozessors werden durch das Programmiermodell eines Rechners ermöglicht. Es erlaubt die Beurteilung einer Rechnerarchitektur vom Standpunkt eines Compiler- oder Betriebssystementwicklers. Der Assembler ermöglicht den Umgang mit dem Rechner auf der Ebene des Programmiermodells.

Nachteile von Assembler gegenüber einer höheren Programmiersprache:
- Portierbarkeit ist nicht gegeben, da der Assembler auf den Befehlssatz einer CPU zugeschnitten ist.
- Lesbarkeit ist i.a. schlechter als für Hochsprachen.
- Fehlermöglichkeiten sind vielfältiger durch die niedrige Ebene der Programmierung.
- Wartbarkeit ist i.a. schwieriger als für höhere Programmiersprachen.

- Assemblerprogrammierung wird z.B bei RISC-Prozessoren immer schwieriger (delayed branching, umsortieren v. Instr. bei Datenabhängigkeiten).

Für die effiziente Umsetzung einer Hochsprache auf eine Zielmaschine ist die genaue Kenntnis des Assemblers dieser Zielmaschine unerläßlich.

7 Ein 8-Bit Mikroprozessor

Unser Modellrechner, der in Kapitel 4 eingeführt wurde, hatte eine Reihe von Unzulänglichkeiten, die sich aus den Restriktionen bezüglich der Anzahl der Befehle und des gewählten Kompromisses bezüglich der Wortbreite des Speichers ergaben. Wir wollen nun einen 8-Bit Mikroprozessor betrachten, der eine wesentliche Erweiterung des Befehlssatzes und der Adressierungsmöglichkeiten bietet. Als Beispiel soll der MC 6809 von Motorola dienen, der sich durch einen überschaubaren, leistungsfähigen Befehlssatz und die wesentlichen Adressierungsmodi auszeichnet. Darüber hinaus besitzt die Architektur einen hohen Grad an Orthogonalität. ***Orthogonalität*** einer Architektur ist ein Maß dafür, inwieweit Komponenten oder Eigenschaften der CPU wie Befehlssatz, Registersatz, Datentypen und Adressierungsmodi unabhängig voneinander sind und sich deshalb beliebig kombinieren lassen. So z.B., daß Befehle auf beliebige Register angewandt werden können oder daß für einen Befehl eine beliebige Adressierungsart gewählt werden kann. Orthogonalität ist ein wesentliches Entwurfsziel und führt durch die Kombination zu einem sehr großen Spektrum möglicher Operationen. Ein bekanntes Beispiel ist die DEC VAX, ein ehemals weit verbreiteter Rechner im Bereich Prozeßsteuerung und ein Vorläufer der Workstations. Die VAX hat einige hundert Befehle, die, wenn man sie mit den verfügbaren Datentypen, den Adressierungsarten (22) und der Spezifikation der Anzahl von Operanden kombiniert, einige hunderttausend verschiedene Operationen ergibt. Die 6809 CPU, die uns als Beispiel dienen soll, hat 59 mnemonische Codes, 10 (Haupt-) Adressierungsarten, 24 Unterarten (Indiziert) woraus sich durch Kombination 1464 unterschiedliche Operationen bilden lassen. Auch für moderne RISC-Architekturen ist Orthogonalität ein angestrebtes Entwurfsziel. Allerdings gibt es im wesentlichen bei den Adressierungsarten sehr starke Einschränkungen (viel signifikanter als in der Anzahl der Befehle). So hat zum Vergleich der RISC-Prozessor MIPS insgesamt nur 4 Adressierungsarten, der SPARC hat 4 Adressierungsarten für Daten und jeweils 1 Adressierungsart für Sprünge und den Unterprogrammaufruf.

7.1 Betrachtungsebene

Bisher haben wir an Beispielen den detaillierten Aufbau der CPU untersucht, d.h. uns mit Fragen der Rechnerorganisation befaßt (vergl. Kapitel 1). Wir werden nun weitgehend die Hardware/Software-Schnittstelle der CPU untersuchen, uns also auf eine für die Programmierung relevante Abstraktionsebene, die der Rechnerarchitektur begeben. Die CPU wird auf dieser Ebene durch das Programmiermodell beschrieben. Die Aspekte des Programmiermodells umfassen:

- Registersatz
- Befehlssatz

- Befehlsformat
- vom Rechner unterstützte Datentypen
- Speicherorganisation
- Auswahl der Operanden (Adressierung)
- Ein/Ausgabe
- Unterbrechungs- und Ausnahmebehandlung

7.2 Registersatz

Der MC 6809 ist ein Mikroprozessor, der intern eine 16 Bit Wortbreite für arithmetisch-logische Befehle und für die Adreßmanipulation besitzt. Seine Speicherschnittstelle nach außen ist durch einen 8-Bit Datenbus und einen 16-Bit Adreßbus gegeben. Abb. 7.1 zeigt die Hauptkomponenten des Mikroprozessors. Dies sind:

- der Registersatz
- das Bussystem bestehend aus: 16-Bit Adreßbus und 8-Bit Datenbus
- die Kontrolleinheit mit speziellen Komponenten zur Steuerung der Unterbrechungsverarbeitung, des Bussystems und des Timings.

Wir wollen den Mikroprozessor zunächst nur auf der Basis des Programmiermodells betrachten, so daß im Moment nur der Registersatz von Interesse ist. Die Steuerung des Busses und die Unterbrechungsverarbeitung werden später behandelt. Der Registersatz ist in Abb. 7.2 noch einmal gesondert dargestellt. Es sind 7 Register zur Unterstützung der Adressierung und 3 Register für die Datenmanipulation vorhanden. Adreßregister sind:

- die Indexregister X und Y (16 Bit)
- die Stackpointer S und U (16 Bit)
- der Programmzähler PC (16 Bit)
- das Direct-Page-Register DP (8 Bit)

Die Datenregister sind:

- die Akkumulatoren A und B (8 Bit),
  A und B können konkateniert werden zu dem Doppelakkumulator D (16 Bit)
- das Condition Code Register CCR (8 Bit)

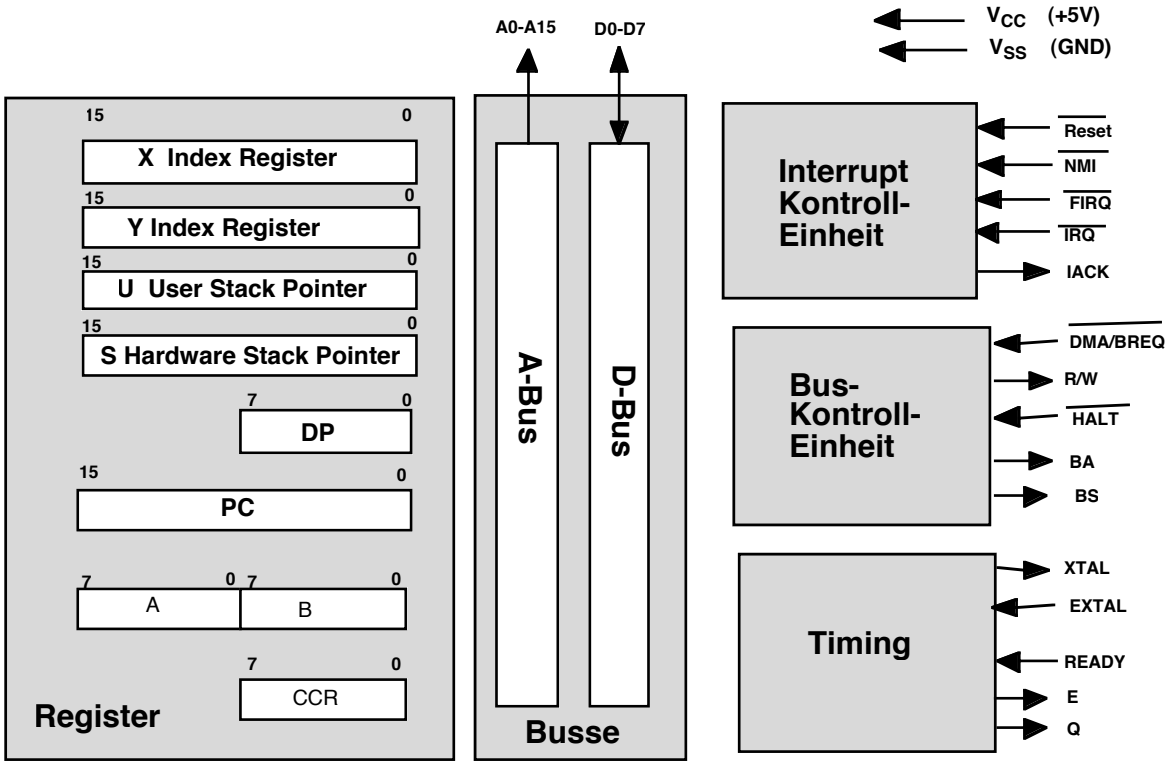


Abb. 7.1 Hauptkomponenten des MC 6809 Prozessors

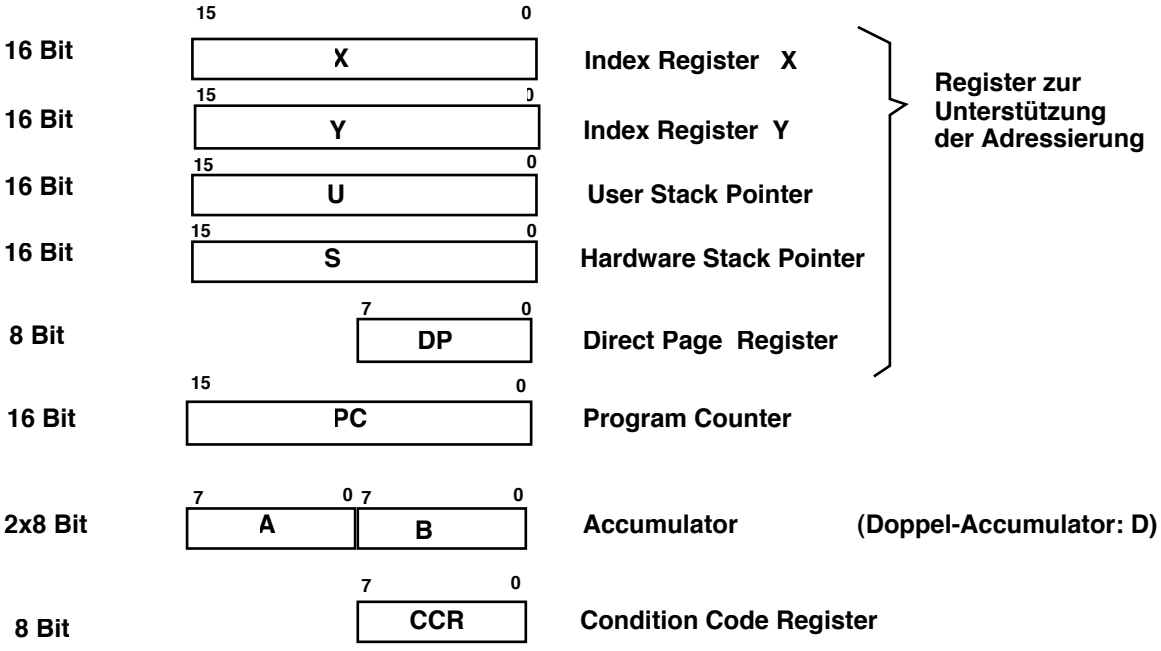


Abb. 7.2 Der Registersatz des MC 6809

Der Programmzähler und der Akkumulator haben die uns schon aus dem Modellrechner bekannten Funktionen. Das Condition Code Register CCR speichert die verschiedenen Flags, welche als Folge einer Datenoperation auftreten können. Abb. 7.3 zeigt die Belegung der

einzelnen Bits des CCR. Die in Abb. 7.3 nicht beschriebenen Bitpositionen dienen der Unterbrechungsverarbeitung und werden später behandelt.

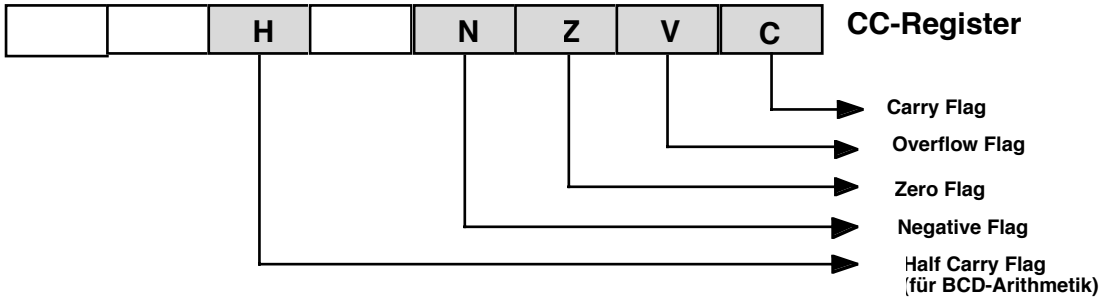


Abb. 7.3 Condition Codes des 6809

- Das ***Carry-Flag*** wird gesetzt, wenn eine Datenoperation einen Übertrag erzeugt.
- Das ***Overflow-Flag*** wird gesetzt, wenn eine Datenoperation einen Überlauf erzeugt.
- Das ***Zero-Flag*** wird gesetzt, wenn das Resultat einer Datenoperation Null ist.
- Das ***Negative-Flag*** wird gesetzt, wenn das Resultat einer Datenoperation negativ ist.
- Das ***Half-Carry-Flag*** wird gesetzt, wenn bei einer Addition das untere Halbbyte >15 ist. Dies wird im Zusammenhang mit der BCD-Addition in Abschnitt 7.3.2 behandelt.

Die Register zur Adressierung werden im Abschnitt 7.4 im Zusammenhang mit den Adressierungsarten des 6809 näher betrachtet.

7.3 Der Befehlssatz

Einen Überblick über die Befehlsklassen des MC6809 gibt Abb. 7.4 .

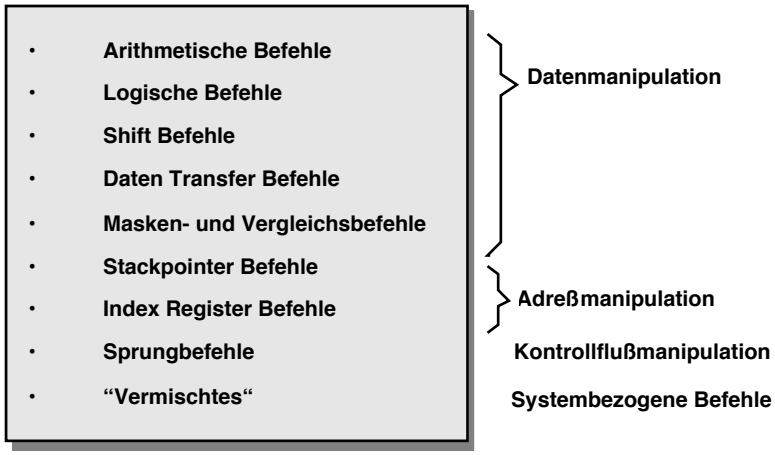


Abb. 7.4 Befehlsklassen des MC6809

Die Befehle sollen hier nur kurz in tabellarischer Form dargestellt werden. Lediglich auf Besonderheiten wird im weiteren Text eingegangen.

7.3.1 Arithmetische Befehle

Arithmetische Befehle:

Addition:	**8**	**ADCA, ADCB**	**Add memory to Accumulator (A,B) with Carry**
		ADDA, ADDB	**Add memory to Accumulator (A,B)**
	16	**ADDD**	**Add memory to Double Accumulator (D)**
BCD-Arithm.:	**8**	**DAA**	**Decimal Adjust Accumulator (A)**
Increment:	**8**	**INC, INCA, INCB**	**Increment memory location bzw. Acc. (A,B)**
Decrement:	**8**	**DEC, DECA, DECB**	**Decrement memory location bzw. Acc. (A,B)**
Multiplikation:	**8**	**MUL**	**unsigned multiply : D ← A x B**
Vorz. Erw.:	**16**	**SEX**	**Sign Extend B Acc. into A Acc.**
Subtraktion:	**8**	**SBCA, SBCB**	**Subtract memory from Acc. (A,B) with Borrow**
		SBA	**Subtract Acc.B from Acc A**
	8/16	**SUBA, SUBB, SUBD**	**Subtract memory from Acc. A,B,D (without Borrow)**

Abb. 7.5 Arithmetisch/logische Befehle

- Addition und Subtraktion liegen in drei verschiedenen Formen vor:
  1.) 8-Bit-Addition/Subtraktion (ADCA, ADCB / SBCA, SBCB) mit Carry/Borrow bedeutet, daß das Carry-Flag aus dem CCR zusätzlich zum Ergebnis addiert, bzw. subtrahiert wird. Dies wird insbesondere bei Multibyte Additionen/Subtraktionen benötigt. Dabei ist zu beachten, daß bei einer Subtraktion das Carry-Bit komplementiert wird, da es ein Borrow darstellt.
  2.) 8-Bit-Addition / Subtraktion ohne Carry /Borrow (ADDA, ADDB / SBA, SBB)
  3.) 16-Bit-Addition / Subtraktion ohne Carry/Borrow (ADDD, SUBD)

- Vorzeichenerweiterung (SEX)

  SEX führt eine Vorzeichenerweiterung des Wertes in Akkumulator B aus, indem es den Akkumulator A mit dem Vorzeichen, d.h. dem Wert von Bit 7 von B, auffüllt. Dies wird genutzt, wenn eine 16-Bit Addition oder Subtraktion mit vorzeichenbehafteten Operanden ausgeführt werden soll und dabei einer der Operanden, der in Accumulator B stehen muß, nur ein 8-Bit Wert ist.

  Beisp.:

  1001 1111

  wird überführt in

  1111 1111 1110 1001

Die Funktion der Operation DAA dient der Realisierung von BCD-Operationen und wird im nächsten Abschnitt genauer behandelt.

7.3.2 BCD-Arithmetik

Heutige Mikroprozessoren haben in der Regel binäre arithmetische Einheiten. BCD-Arithmetik kann damit realisiert werden, indem man zunächst eine Operation wie auf binären Operanden ausführt und im Anschluß daran das Ergebnis entsprechend korrigiert. Der MC 6809 hat dazu zwei Hilfsmittel:
1.) Das Half-Carry-Flag
2.) Die Operation DAA (Decimal Adjust Accumulator)

Abb. 7.6 gibt die Repräsentation von BCD-Zahlen in einem Byte an. Das Half-Carry-Flag im CC-Register wird gesetzt, wenn bei einer binären Addition ein Übertrag aus dem unteren Halbbyte erzeugt wird, d.h. wenn die Summe der unteren Halbbytes > 15 (!) ist. Es ist zu beachten, daß nicht etwa ein Half-Carry erzeugt wird, wenn die Summe der unteren Halbbytes > 9 ist, da die ALU des Mikroprozessors nur Binärzahlen "kennt".

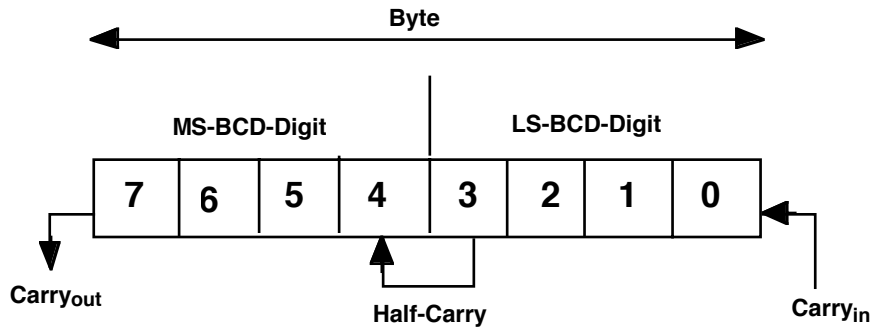


Abb. 7.6 Repräsentation von BCD-Zahlen in einem Mikroprozessor (MC 6809)

Das Half-Carry und der Binärwert des jeweiligen Halbbytes werden von der Instruktion DAA ausgewertet, um nach einer binären Addition eine korrekte BCD-Zahl zu erzeugen. Dazu addiert die Instruktion DAA unter bestimmten Bedingungen einen Korrekturfaktor. Seien a und b zwei 4-Bit BCD-Ziffern ($a \leq 9$, $b \leq 9$). Zunächst einmal halten wir fest, daß die binäre Addition eines Halbbytes mod 16, d.h. Hexadezimal erfolgt, die einer BCD-Zahl mod 10. Im binären Fall wird ein Übertrag in die nächste Stelle erzeugt, wenn a+b >15, im BCD-Fall muß ein Übertrag in die nächste Stelle erfolgen, wenn a+b > 9. Betrachten wir nun die möglichen Fälle bei der BCD-Addition.

1.) Für die Summe gilt: $\mathbf{0 < a+b \leq 9}$. In diesem Fall ist keinerlei Korrektur erforderlich.
2.) Für die Summe gilt: $\mathbf{9 < a+b \leq 15}$. Es ist kein Half-Carry erzeugt worden.. Die Zahlen 10_{10} bis 16_{10} werden durch Addition des Korrekturwertes +6 auf 10_{16} bis 15_{16} abgebildet, so daß das entsprechende Halbbyte nach dieser Korrektur die richtige BCD-Ziffer enthält. Zusätzlich wird durch die Addition des Korrekturwertes ein Übertrag in die nächste Stelle erzeugt, so daß diese der BCD-Rechnung entsprechend inkrementiert wird.

Beisp.:

5	0101	
+7	+ 0111	
12	1100	= 10 + 2
+6	0110	Addition des Korrekturfaktors
1	0010	= 16 + 2, Erzeugung des (binären) Übertrags in die nächste Stelle

2.) Betrachten wir nun den Fall, daß **15 < a+b ≤ 18** (größer als 18 kann die Summe zweier BCD-Ziffern natürlich nie werden). In diesem Fall wurde das Half-Carry gesetzt, da bei der binären Addition ein Überlauf erzeugt wurde. In diesem Fall müssen wir ebenfalls den Korrekturwert addieren, da $16_{16} + x = 10_{10} + x + 6_{10}$. Da wir a+b > 15 angenommen haben, wurde der Überlauf bereits korrekt erzeugt.

Beisp.:

9	1001	
+8	+ 1000	
17	**1** 0001	
+6	0110	Addition des Korrekturfaktors
1	0111	= 23

Die Bedingungen für die Anwendung des Korrekturfaktors +6 lassen sich zusammenfassen: (H = 1) + (sum > 9).

In einem Byte sind normalerweise im sogenannten "gepackten BCD"-Format 2 BCD-Ziffern enthalten. Die oben angegebenen Bedingungen gelten für das LSD (Least Significant Digit). Für das MSD (Most Significant Digit) gelten folgende leicht nachvollziehbare Bedingungen: $(C = 1) + (sum_{MSD} > 9) + (sum_{MSD} > 8 \bullet sum_{LSD} > 9)$.

7.3.3 Logische Befehle

Logische Befehle:

8	**ANDA, ANDB**	**And memory to Accumulator (A,B)**
8	**COM, COMA, COMB**	**Complement memory, Accumulator (A,B)**
8	**EORA, EORB**	**Exclusiv OR memory with Accumulator (A,B)**
8	**NEG, NEGA, NEGB**	**Negate memory or Acc. (A,B) (2's complement)**
8	**ORA, ORB**	**OR memory with Accumulator (A,B)**

A

bb. 7.7 Logische Befehle des 6809

Die logischen Befehle weisen keine Besonderheiten auf und können aus Abb. 7.7 entnommen werden. Es ist lediglich zu beachten, daß sie nur für 8-Bit-Werte vorhanden sind.

7.3.4 Shift Befehle

Die Shiftbefehle sind in Abb. 7.8 aufgelistet. Sie sind für die Register A und B sowie für Speicheroperanden definiert.

Shift Befehle:

8	ASL, ASLA, ASLB	Arithmetic Shift Left memory or Accumulator (A,B)
8	ASR, ASRA. ASRB	Arithmetic Shift Right memory or Accumulator (A,B)
8	LSL, LSLA,LSLB	Logical Shift Left memory or Accumulator (A,B)
8	LSR, LSRA, LSRB	Logical Shift Right memory or Accumulator (A,B)
8	ROL, ROLA, ROLB	Rotate Left memory or Accumulator (A,B)
8	ROR, RORA, RORB	Rotate Right memory or Accumulator (A,B)

Abb. 7.8 Die Shift-Befehle des MC 6809

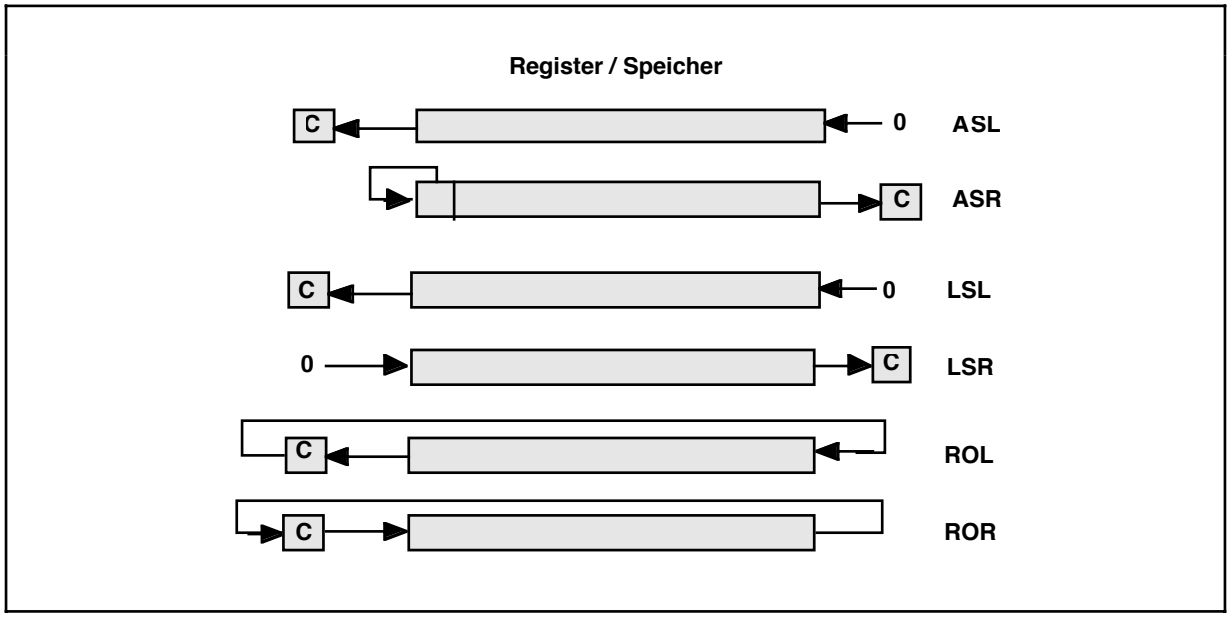


Abb. 7.9 Graphische Darstellung der Shift Befehle des 6809

- ASL: Arithmetischer Linksshift. Das höchstwertige Bit (MSB) wird in das Carry-Flag im CCR geschoben, das niederwertigste Bit (LSB) mit 0 aufgefüllt.
- ASR: Arithmetischer Rechtsshift. Das MSB des entsprechenden Registers wird dupliziert. Das LSB wird in das Carry Flag geschoben.
- LSL: Logischer Linksshift. Entspricht exakt dem arithmetischen Linksshift ASL.
- LSR: Logischer Rechtsshift. Das MSB des entsprechenden Registers wird mit “0" aufgefüllt. Das LSB wird in das Carry Flag geschoben.
- ROL: Zyklischer Linksshift. Das MSB wird in das Carry-Flag im CCR geschoben. Der ursprüngliche Inhalt des Carry-Flags wird in das LSB des entsprechenden Registers übertragen. ROL ist also ein zyklischer Shift, der das Carry-Flag als zusätzliches Bit enthält.

- ROR: Zyklischer Rechtsshift. Das LSB wird in das Carry-Flag im CCR geschoben. Der ursprüngliche Inhalt des Carry-Flags wird in das MSB des entsprechenden Registers übertragen. ROR ist wie ROL ein zyklischer Shift, der das Carry-Flag als zusätzliches Bit enthält.

Abb. 7.9 faßt die Shift-Operationen in einer graphischen Darstellung zusammen.

7.3.5 Maskenoperationen und Vergleichsoperationen

Einen Überblick über die im MC 6809 vorhandenen Masken- und Vergleichsoperationen gibt Abb. 7.10. Eine entscheidende Eigenschaft haben alle diese Operationen gemeinsam: Sie **verändern** den oder die **Operanden** auf die sie angewandt werden **nicht**, sondern setzen lediglich die Flags im CC-Register entsprechend.

8	**BITA, BITB**	**Bit test memory with Accumulator (A,B)**
8	**CMPA, CMPB**	**Compare memory with Accumulator (A,B)**
8	**TST, TSTA, TSTB**	**Test memory location or Acc. (A,B)**
16	**CMPx x=D, S, U, X, Y,**	**Compare memory with Register x**

Abb. 7.10 Masken- und Vergleichsoperationen im 6809

Sie werden nun kurz vorgestellt:

1.) BITA, BITB
Unter einer Maskenoperation versteht man eine Instruktion, die aus einem Byte oder Wort bestimmte Stellen ausblenden und testen kann. Die Operation BITA (BITB ist äquivalent für Register B) führt ein logische UND des Inhalts von A mit einem Speicherwort durch und setzt die Flags N und Z im CC-Register entsprechend. Das folgende kurze Programm zeigt die Verwendung von BITA: Nehmen wir an, in einem Hotel gibt es 8 Zimmer, die mit einer Klingel ausgestattet sind. In regelmäßigen, kurzen Abständen soll überprüft werden, ob und wer geklingelt hat. Die 8 Zimmer sind als je ein Bit in einem Byte repräsentiert. Eine gedrückte Klingel wird als "1" repräsentiert. Mit "LDA bellboard" können wir das Byte lesen.

```
loop    LDA   bellboard
        BITA   #$FF      die Maske "1111 1111" testet, ob irgend eine Klingel gedrückt wurde
        BEQ   loop       falls Z=1, d.h. im "bellboard" befindet sich keine "1", dann zurück nach loop
        LDB   #1         Lade "1"
        STB   mask       Setze den Speicherinhalt an der Adresse mask auf "000 0001"
scan    BITA  mask       logisches UND von mask und Reg. A
        BNE   out        Sprung zur Ausgabe, wenn der Test eine 0 ergab.
        LSL   mask       Shift Left um 1 Stelle
        BRA   scan
out     JSR   ……         Sprung zur Ausgabe, der Inhalt von "mask" enthält die die
                         1-aus-n Repräsentation der Zimmernummer
```

Mit dem Befehl BITx können wir auch sehr einfach überprüfen, ob ein Operand negativ ist, indem wir ihn mit der Maske "1000 0000" testen. Allerdings haben wir dazu noch die Spezialfunktion TST, die genau für diesen Zweck vorgesehen ist.

2.) TST

Mit TST werden das N-Flag und das Z-Flag gemäß dem Wert des getesteten Operanden gesetzt, so daß festgestellt werden kann, ob dieser Operand 0 oder negativ ist.

3.) CMPx

Die Compare-Operation ist die komplexeste der Vergleichsoperationen. Sie führt eine 8-Bit Subtraktion durch. CMPA "mem" subtrahiert den Inhalt der in "mem" spezifizierten Adresse vom Register A und setzt die Flags entsprechend. Weder das A-Register noch der Speicher werden dabei verändert. Es ist zu beachten, daß der Wert des Carry-Flags bei dieser Operation komplementiert wird, da es sich um ein "Borrow" handelt.

Beispiel: CMPA mem

1.)

```
A    =  37 (0010 0101)
mem =  19 (0001 0011)

       0010 0101
       1110 1101          2er-Komplement von 19
   1   0001 0010          Das Carry-Flag wird komplementiert

Folgende Belegung des CC-Registers wird erzeugt:
      N Z V C
      0 0 0 0
```

2.)

A = 39 (0010 0111)
mem = 67 (0100 0011)

```
      0010 0111
      1011 1101        2er-Komplement von 67
   0  1110 0100        Das Carry-Flag wird komplementiert
```

Folgende Belegung des CC-Registers wird erzeugt:

```
N Z V C
1 0 0 1
```

Masken- und Vergleichsoperationen werden meist vor bedingten Sprüngen ausgeführt. Die Sprungbedingungen werden dann aus der Belegung der Flags im CC-Register abgeleitet.

7.3.6 Relative Sprünge

Die Liste der relativen Sprünge ist in Abb. 7.11 angegeben. Bei einem relativen Sprung wird das Sprungziel nicht als absolute Adresse wie in unserem Modellrechner, sondern als Distanz von der gerade aktuellen Position im Programm angegeben. Wir werden auf die Adressierung des Sprungziels im Abschnitt 7.4 näher eingehen.

Branch-Befehle:

BCC, LBCC	**Branch if Carry Clear**	
BCS, LBCS	**Branch if Carry Set**	
BEQ, LBEQ	**Branch if equal**	
BNE, LBNE	**Branch if not equal**	
BMI, LBMI	**Branch if minus**	
BPL, LBPL	**Branch if plus**	
BRA, LBRA	**Branch always**	**unbedingter Sprung**
BRN, LBRN	**Branch never**	
BVC, LBVC	**Branch if overflow clear**	
BVS, LBVS	**Branch if overflow set**	
BSR, LBSR	**Branch to Subroutine**	**Unterprogrammsprung**
Bedingte Sprünge, die bei der Auswertung der Sprungbedingung das Vorzeichen interpretieren (signed):		
BGE, LBGE	**Branch if greater than or equal**	
BLE, LBLE	**Branch if less than or equal**	
BGT, LBGT	**Branch if greater**	
BLT, LBLT	**Branch if less than**	
Bedingte Sprünge, die bei der Auswertung der Sprungbedingung das Vorzeichen nicht interpretieren (unsigned):		
BHS, LBHS	**Branch if higher or same**	
BLS, LBLS	**Branch if lower or same**	
BHI, LBHI	**Branch if higher**	
BLO, LBLO	**Branch if lower**	

Abb. 7.11 Relative Sprünge (Branch-Befehle) des 6809

Alle Branch-Befehle liegen in einer Kurzform (Bxx) und einer Langform (LBxx) vor, abhängig davon, wie weit das Sprungziel von der augenblicklichen Position entfernt ist. Branch-Befehle werden zunächst in unbedingte und bedingte Branch-Befehle eingeteilt. Unbedingte Branch-Befehle sind:
BRA, LBRA und der Unterprogrammsprung BSR, LBSR. Der Befehl BRN, LBRN führt keinerlei Funktion aus, sondern erhöht lediglich den Programmzähler um eine definierten Wert. Dieser Befehl entspricht daher einem NOP[1].

Unter den bedingten Sprüngen gibt es solche, deren Sprungbedingung explizit auf einer Belegung der entsprechenden Flags im CCR beruht.

BCC, BCS,
BEQ, BNE
BMI, BPL
BVC, BVS jeweils mit ihren entsprechenden Langformen.

Die zweite Kategorie der bedingten Sprünge berücksichtigt als Sprungbedingung die Beziehung zwischen zwei Operanden. Diese Befehle mit den entsprechenden Gleichungen/ Ungleichungen für die Relation der Operanden werden in Abb. 7.12 gezeigt.

Bedingung	mit Vorz.	ohne Vorz.	Branch Mnemonic Bxx	Flags
$a \neq b$	x	x	BNE	Z = 0
$a = b$	x	x	BEQ	Z = 1
$a \leq b$	x		BLE	$Z + (N \oplus V) = 1$
$a < b$	x		BLT	$N \oplus V = 1$
$a \geq b$	x		BGE	$N \oplus V = 0$
$a > b$	x		BGT	$Z + (N \oplus V) = 0$
$a \leq b$		x	BLS	Z + C = 1
$a < b$		x	BLO	C = 1
$a > b$		x	BHS	C = 0
$a \geq b$		x	BHI	Z + C = 0
			BCC	C = 0
			BCS	C = 1
			BEQ	Z = 1
			BNE	Z = 0
			BMI	N = 1
			BPL	N = 0
			BVC	V = 0
			BVS	V = 1

Abb. 7.12 Bedingte Sprünge und zugeordnete Sprungbedingungen

[1] Da die Befehle BRN und LBRN verschiedenen Länge haben können, entsprechen sie einer variabel langen NOP Operation.

Die Sprungbedingungen werden aus der Belegung des CCR abgeleitet. Es ist zu beachten, daß Sprungbedingungen existieren, die das Ergebnis einer Vergleichsoperation von Absolutwerten ohne Vorzeichen berücksichtigen und solche, die vorzeichenbehaftete Werte annehmen. Dies ist wichtig, wie folgendes Beispiel zeigen soll:
a = 1001 0101
b = 0010 0111
Als vorzeichenbehaftete Zahlen interpretiert gilt a<b, da a negativ ist.
Als Absolutwerte interpretiert gilt a>b.
Daher muß man bei der Programmierung auf dieser Ebene die bedingten Sprünge sorgfältig auswählen.

Eine typische Programmsequenz für bedingte Sprünge ist eine Masken/Vergleichsoperation gefolgt von einem bedingten Sprung[2]. Die Masken-/Vergleichsoperation setzt die Flags im CCR und die darauf folgende Sprungbedingung wertet sie nach Abb. 7.12 aus.

Zwei Beispiele sollen die Nutzung der Flags im CCR erläutern:
Es gilt:

A = 1010 0011 $A3 (hex)
mem = 0110 0100 $64 (hex)

CMPA mem

```
      1010 0011
    + 1001 1100        2-Komplement von $64
     1 0011 1111
```

Folgende Belegung des CC-Registers wird erzeugt:

```
N Z V C
0 0 1 0
```

1.) CMPA mem
BGT dest

Die Sprungbedingung für BGT lautet: $Z + (N \oplus V) = 0$. Der Sprung wird nicht ausgeführt, da V=1. Das ist auch korrekt, da BGT ein Sprung ist, der mit vorzeichenbehafteten Zahlen arbeitet. In diesem Fall wird der Inhalt von A als negative Zahl interpretiert. Deshalb gilt:

A < [mem].

[2] Natürlich sind bedingte Sprünge auch nach arithmetisch/logischen oder Shift Operationen sinnvoll.

2.) CMPA mem
BHI dest

Die Sprungbedingung für BHI lautet: Z + C = 0. Es gilt Z=0. Da C komplementiert wird (Borrow), ist auch das C-Flag im CC-Register 0. Daher gilt die Bedingung und der Sprung wird ausgeführt. Für die Absolutwerte gilt: |A| > |[mem]|.

Es soll noch einmal darauf hingewiesen werden, daß die CMPx Operation einfach Absolutwerte subtrahiert und keinerlei Vorzeichen berücksichtigt. Erst durch die Auswertung der Flags des CCR werden die erfolgte Subtraktion als vorzeichenbehaftetet oder nicht vorzeichenbehaftet interpretiert und die entsprechenden Sprünge ausgeführt.

7.3.7 Datentransfer Befehle

Diese Befehle ermöglichen den expliziten Operandentransfer zwischen verschiedenen Speicherplätzen, zwischen Speicherplätzen und Registern der CPU und zwischen zwei Registern . Ihre Funktion bedarf keiner weiteren Erläuterung. Die Spezifikation der Quelle und des Ziels eines solchen Transfers wird im Abschnitt über die Adressierung näher betrachtet.

Daten Transfer:

8	EXG R1, R2	Exchange R1 with R2 (R1, R2 : A,B, CC, DP)
16	EXG R1, R2	Exchange R1 with R2 (R1, R2 :D, X, Y, U, S, PC)
8	LDA, LDB	Load Accumulator (A,B) from memory
16	LDx (x: D, X, Y, U, S)	Load Register from memory
8	STA, STB	Store Accumulator (A,B) to memory
16	STx (x: D, X, Y, U, S)	Store Register to memory
8	TFR, R1,R2	Transfer R1 to R2 (R1, R2 : A,B, CC, DP)
16	TFR, D,R	Transfer D to X, Y, S, U, PC
16	TFR, R, D	Transfer X, Y, S, U, PC to D

Abb. 7.13 Befehle zum expliziten Datentransfer

7.3.8 Sonstige Befehle

Der MC 6809 enthält noch eine Reihe weiterer Befehle. Abb. 7.14 und 7.15 geben die Befehle für absolute Sprünge (zur Beachtung: es gibt nur unbedingte absolute Sprünge), den allgemeinen RTS und die Befehle zum expliziten Löschen an.

Sprung- und Unterprogrammsprung- bezogene Befehle:

JMP	Jump	im Gegensatz zu Branch ist JMP ein absoluter Sprung
JSR	Jump to Subroutine	
RTS	Return from Subroutine	

Abb. 7.14 Absolute Sprünge des 6809 und RTS

Löschbefehl:

8	**CLR, CLRA, CLRB**	**Clear memory bzw. Accumulator (A,B)**

Abb. 7.15 Explizite Löschbefehle des 6809

Weitere wichtige Befehlsklassen sind:

- Befehle zur Manipulation von Adreßregistern
- Befehle für den Stack
- Befehle zur Unterbrechungsbearbeitung

Diese Befehle werden im Zusammenhang mit der Adressierung und der Unterbrechungsbehandlung genauer betrachtet.

7.4 Adressierungsarten

Von unserem Modellrechner kennen wir nur eine einzige Adressierungsart, nämlich die Angabe der vollständigen Speicheradresse im Adreßfeld des Operanden. Dies führt bei der Adressierung großer Speicher zu einem sehr langen Adreßfeld und demzufolge zu einem erhöhten Speicherbedarf für Programme. Daher ist eines der Ziele beim Entwurf eines Adressierungsmechanismus, die Adressierbarkeit eines möglichst großen Adreßraums mit einem möglichst kleinen Adreßfeld in der Instruktion zu realisieren. Dies sind zunächst gegensätzliche Forderungen, die unvereinbar erscheinen. Eine Möglichkeit zur Lösung des Zielkonflikts besteht jedoch in der Berücksichtigung der Lokalität von Berechnungen, so daß nicht in jeder Instruktion der gesamte Adreßraum direkt zugreifbar sein muß, sondern nur ein gewisser Teil, den das Programm gerade benötigt. Außerdem muß nicht, wie in unserem Modellrechner, eine feste Länge unterschiedslos für jede Instruktion vorgegeben sein, sondern kann variieren, je nachdem, ob zur Operandenspezifikation eine Adresse überhaupt notwendig ist oder ob sich der Operand z.B. in einem (von wenigen) Registern befindet, das mit einer kurzen Adresse spezifiziert werden kann.

Ein weiteres wichtiges Ziel beim Entwurf des Adressierungsmechanismus ist die Vereinfachung der Adressierung von Datenstrukturen wie Arrays oder Listen für den Programmierer. Mit dem einfachen Adressierungsmechanismus unseres Modellrechners war dies nur durch selbstmodifizierenden Code möglich, was umständlich und fehleranfällig ist.

Im folgenden werden wir Adressierungsarten am Beispiel des 6809 kennenlernen, der die wesentlichen Adressierungsvarianten besitzt.

7.4.1 Speicherorganisation

Für unseren Modellrechner haben wir eine Speicherorganisation gewählt, in der Worte fester Länge adressiert werden können. In jedem Speicherzyklus wird ein vollständiges Wort geholt und vom Prozessor verarbeitet. Dadurch besteht eine 1-zu-1-Zuordnung zwischen dem Wort, das im Prozessor verarbeitet werden kann, und der kleinsten addressierbaren Einheit im Speicher. Der Modellrechner wird deshalb als ***wortorientierte CPU***, der Speicher als ***wortadressierbarer Speicher*** bezeichnet. Abb. 7.16 zeigt eine wortadressierte Speicherorganisation. Wir haben aber bereits in unserem Modellrechner gesehen, daß bei bestimmte Befehlen, z.B. RAL, NOT oder NOP nicht das gesamte Speicherwort benötigt wird, sondern nur der OPCODE, so daß Speicher verschwendet wird. Umgekehrt muß in ein Speicherwort jeweils ein OPCODE und eine Adresse passen, was den Adreßbereich stark einschränkt. Um ein hohes Maß an Flexibilität bei bestmöglicher Ausnutzung des Speichers zu erzielen, ist es deshalb günstig, die Orientierung der CPU und die Addressierung des Speichers orthogonal zu behandeln. Die übliche Einheit der Adressierung eines Speichers ist heute ein Byte. Die Grundform der Byteadressierung ist in Abb. 7.17 angegeben. Für die Kombination eines wortorientierten Prozessors und eines byteaddressierten Speichers muß allerdings die Frage geklärt werden, wie die Bytes, die in aufsteigenden Adressen des Speichers vorliegen, im Prozessor zur Verarbeitung geordnet werden. Abb. 7.18 beschreibt eine mögliche Ordnung der Bytes für einen Prozessor mit einer Wortgröße von 32 Bit.

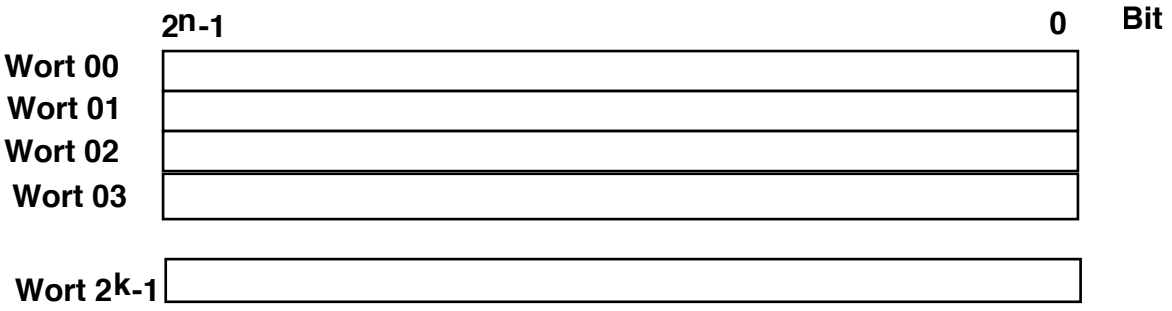


Abb. 7.16 Wortadressierter Speicher

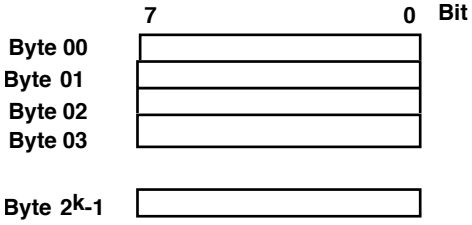
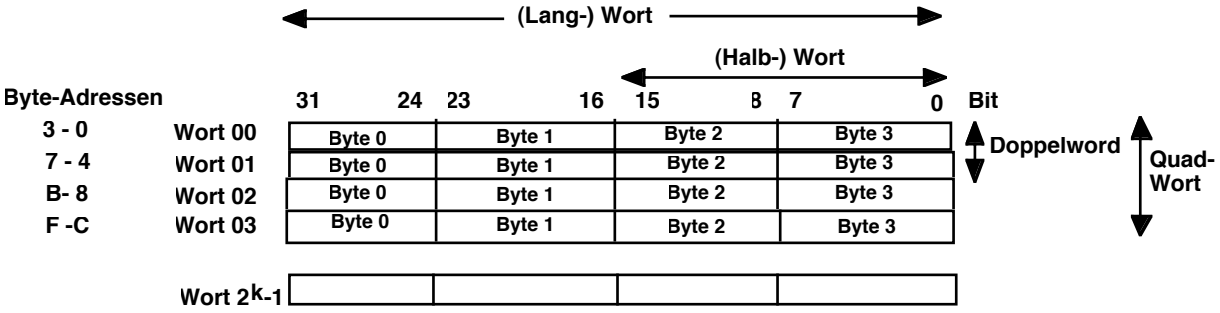


Abb. 7.17 Byteaddressierter Speicher

Bytes können je nach Bedarf zusammengefaßt werden zu:

- 16-Bit Halbworten,
- 32-Bit Worten
- 64-Bit Doppelworten
- 128-Bit Vierfach(Quad)-Worten.



7.18 Wortorganisation in einem byteadressierten Speicher

In der dargestellten Ordnung werden die Bits eines Wortes aufsteigend von rechts nach links numeriert, die Bytes eines Wortes von links nach rechts. Das höchstwertige Byte liegt auf der niedrigsten Speicheradresse. Diese Darstellung des Wortes wird als ***inkonsistente Big-Endian-Darstellung*** bezeichnet[3]. Der Prozessor 6809 ist ein Byte-orientierter Prozessor. Er basiert auf einem Byte-adressierten Speichermodell mit variabler Wortlänge. Dabei sind die Worte, die länger sind als ein Byte, in der oben dargestellten inkonsistenten Big-Endian Reihenfolge geordnet.

7.4.2 Befehlsformat

Das Befehlsformat legt fest, wie der Rechner einen Befehl interpretiert. Dies wurde bereits in Abschnitt 4.2 kurz diskutiert. Das grundlegende Befehlsformat des MC6809 ist in Abb.7.19 angegeben. Da der MC6809 eine Akkumulatormaschine ist, ist auch das grundlegende Befehlsformat äquivalent zu dem unseres Modellrechners.

Abb. 7.19 Basisformat eines Maschinenbefehls des MC6809

Die wesentliche Annahme bei diesem Modell ist, daß ein Operand immer implizit im Akkumulator vorhanden ist. Man muß bei einstelligen Operationen wie Shift-Befehlen oder Komplementbildung keine weitere Operandenspezifikation hinzufügen. Bei zweistelligen Operationen muß nur der zweite Operand weiter spezifiziert werden.

---

[3] Eine Diskussion der unterschiedlichen Byte-Ordnungen ist in dem Teil "Speicher und Periphere Geräte" zu finden.

Im Gegensatz zu unserem Modellrechner hat der MC6809 ein byteadressiertes Speichermodell. Dabei ist es schwer möglich, den gesamten Befehl, d.h. OPCODE und Adreßfeld in einem einzigen Byte zu kodieren, so daß der MC6809 ein Verfahren mit verschieden langen Befehlen realisiert. Die unterschiedlichen Befehlsformate reichen von einem einzelnen Byte bis zu Befehlen, die 5 Bytes umfassen. Sie sind in Abb. 7.20 dargestellt.

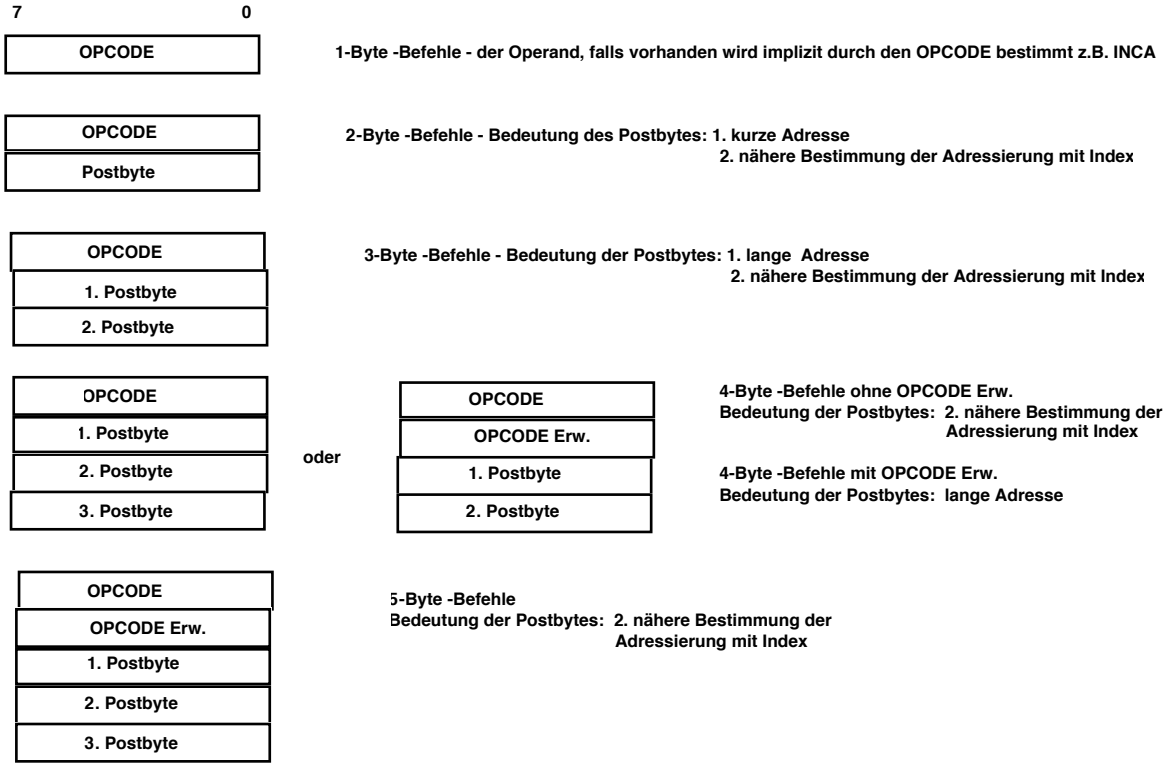


Abb. 7.20 Variationen des Basisbefehlsformats beim MC6809

Im ersten Byte ist immer der OPCODE angegeben. Der MC 6809 besitzt einen sehr orthogonalen Befehlssatz in der Hinsicht, daß jeder Befehl mit jeder Adressierungsart benutzt werden kann, sofern dies sinnvoll ist. Diese Kombinationsmöglichkeit ergibt bei einer begrenzten Menge von Basisbefehlen und Adressierungsarten mehr Kombinationen, als sich in einem Byte kodieren lassen. Die Designer des MC6809 haben deshalb die am meisten benötigten Befehle mit den gängigsten Adressierungsarten in einem einzigen Byte OPCODE kodiert. Die weniger gebräuchlichen oder die mit sehr komplizierten Adressierungsarten dagegen benötigen einen Zwei-Byte-OPCODE. Aus dem ersten Byte kann die Maschine immer schließen, ob es sich um einen 1-Byte- oder einen 2-Byte-Befehl handelt.

Grundsätzlich können in einem Byte solche Befehle kodiert werden, die keine nähere Spezifikation des Operanden benötigen. Z.B. Befehle, die keinen Operanden haben wie NOP oder HLT, oder auch Komplementbildung und Shift-Befehle, bei denen der Operand implizit im Akkumulator angenommen wird. Solche Befehle können auch als ***0-Adreß-Befehle*** angesehen werden. Die Spezifikation des Operanden ist implizit und im OPCODE enthalten (z.B. RAL : Rotate **Accumulator** Left).

Für Befehle, die einen zweiten Operanden erfordern, wird zusätzliche Information benötigt, die in den auf den OPCODE folgenden Bytes abgelegt wird. Dabei gibt es zwei unterschiedliche Möglichkeiten:

1.) Der benötigte Operand steht direkt in den auf den OPCODE folgenden Bytes. In diesem Fall spricht man von einem Direktoperanden, auch Literal genannt. Die Adressierungsart wird als ***unmittelbare Adressierung*** bezeichnet. Abb. 7.21 zeigt die verschiedenen Formate für Direktoperanden im 6809.

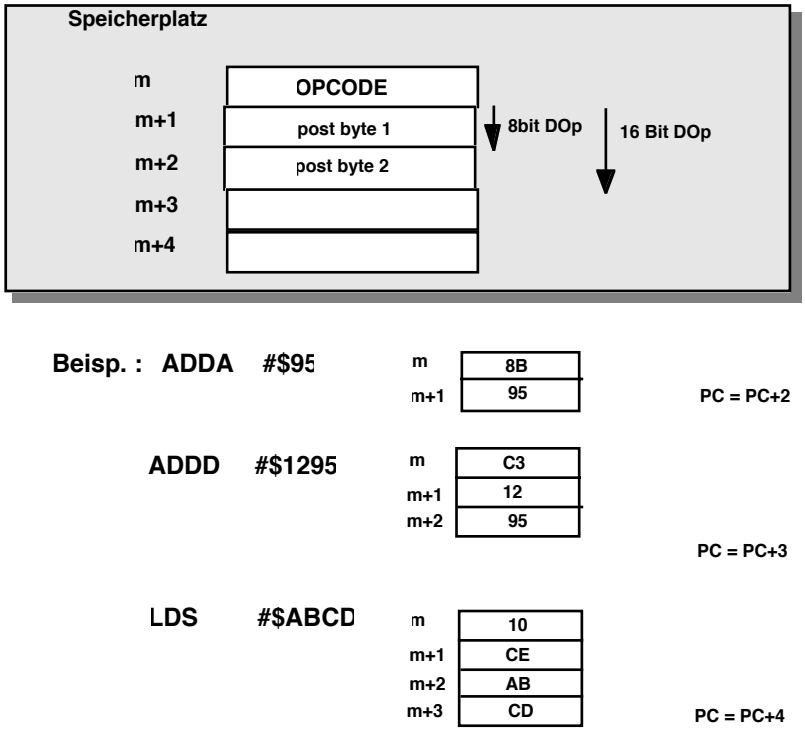


Abb. 7.21 Direktoperanden im MC6809

2.) Die Adresse des Operanden ist in den, auf den OPCODE folgenden Bytes spezifiziert. Dabei gibt es sehr viele verschiedene Möglichkeiten, die wir im folgenden diskutieren werden.

7.4.3 Registeradressierung

Registeradressierung bezieht sich auf die Spezifikation von CPU-Registern, in denen der Quell- oder Zieloperand einer Instruktion enthalten ist. Der MC6809 ist im wesentlichen eine Accumulatormaschine und hat keine allgemeinen Register, so daß es für die arithmetisch-logischen Instruktionen keine Registeradressierung gibt. Ausnahmen sind solche Instruktionen, in denen Quell -und Zielregister implizit sind, z.B. ABX, MUL, etc. Lediglich Transferoperationen zwischen Registern und Stackoperationen nutzen Registeradressierung.

Der MC6809 hat 10 Register, so daß ein Register eindeutig durch eine 4-Bit-Adresse identifiziert werden kann. Da mehr Adressen als Register zur Verfügung stehen, sind ungültige Adressen möglich (vergl. Abb. 7.22).

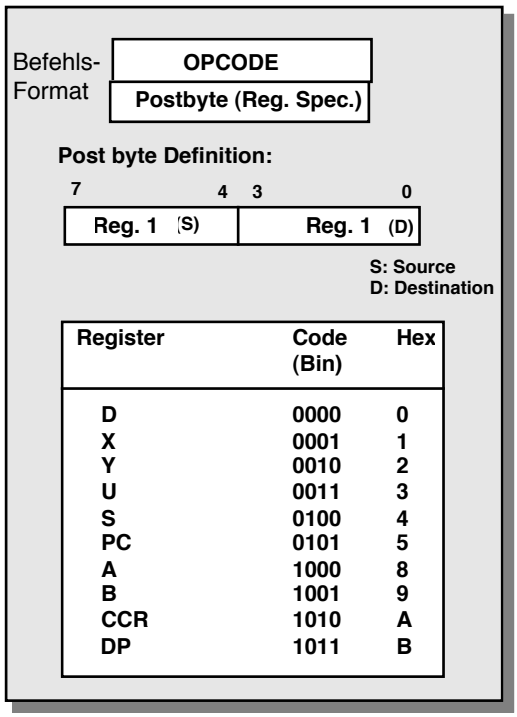


Abb. 7.22 Format der allgemeinen Registeradressierung bei TFR und EXG

Die Datentransferbefehle sind in Abb. 7.13 angegeben. Spezielle Befehle, die sich auf Register beziehen, sind dabei: EXG, TFR, PSHS, PSHU, PULS, PULU. TFR und EXG sind Befehle, die Registerinhalte zwischen Registern transferieren bzw. austauschen. Bei TFR a,b wird der Registerinhalt von Register a nach b transferiert, wobei der alte Registerinhalt von b überschrieben wird. EXG vertauscht die Registerinhalte von a und b. EXG ist symmetrisch, d.h. EXG a,b entspricht EXG b,a . TFR und EXG führen zu Fehlern, wenn:

1. Register unterschiedlicher Länge als Quelle und Ziel angegeben werden.
2. Eine ungültige Registeradresse verwendet wird (siehe unten).

TFR und EXG haben als Quell- und als Zieloperand jeweils ein Register. Das Quell- und das Zielregister werden in diesen Instruktionen im Byte spezifiziert, das unmittelbar auf den OPCODE folgt, dem sogenannten ***Postbyte***.

TFR und EXG sind Operationen, die im Gegensatz zu dem normalen Ein-Adress-Befehl der Accumulatormaschine zwei Adressen in einem Befehl spezifizieren. Durch die kurzen 4-Bit Registeradressen können diese in einem einzigen Byte untergebracht werden.

Eine noch kompaktere Spezifikation von Adressen ist in den Stackbefehlen verwirklicht.

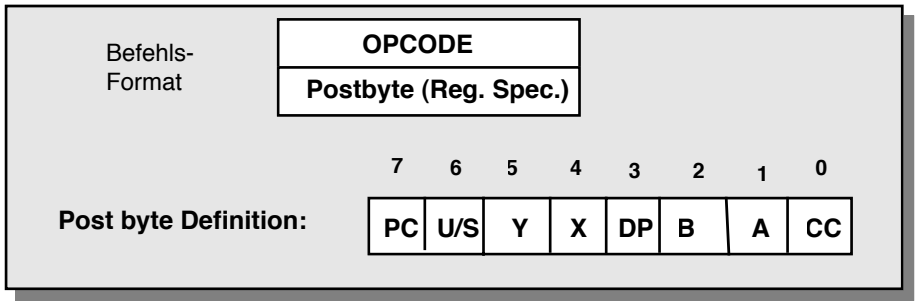


Abb. 7. 23 Registermaske bei den Stackoperationen: PSHS, PSHU, PULS, PULU

Bei Stackoperationen wird eine kodierte Liste der Register angegeben, die auf den Stack geschrieben oder vom Stack gelesen werden sollen. Abb. 7.23 zeigt das Befehlsformat und das Format des Postbytes für Stackoperationen. Jede Bitposition markiert ein bestimmtes Register. Ist das Bit an dieser Stelle auf "1" gesetzt, wird das entsprechende Register bei einem Stackbefehl berücksichtigt, ist das Bit "0", wird das Register ignoriert. Die so kodierte Liste wird auch als ***(Bit-) Maske*** bezeichnet. Entsprechend der Postbyte Spezifikation ist die Reihenfolge der Register auf dem Stack wie in Abb. 7.24 gezeigt.

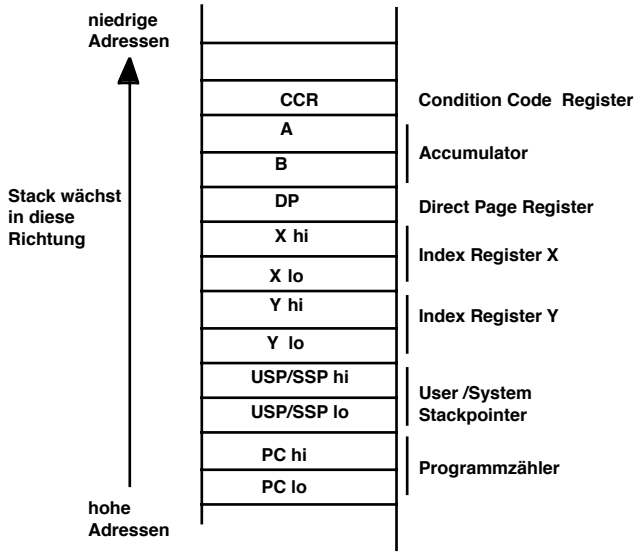


Abb. 7.24 Reihenfolge der Register auf dem Stack nach einer Stackoperation

Der Stack wächst in die Richtung der niedrigen Adressen. Bei einer Push-Operation wird der Stackpointer zunächst dekrementiert (Predecrement), dann wird das Datum an die neue Position abgespeichert. Bei der Pull-Operationen wird das Datum an der aktuellen Position des Stackpointers gelesen, dann wird der Stackpointer inkrementiert (Postincrement). 16-Bit Register belegen 2 aufeinanderfolgende Stackadressen in Big-Endian Ordnung (hi: höherwertiges Byte; lo: niederwertiges Byte). Die Verwendung des Stacks wird in Kapitel 8 weiter diskutiert.

7.4.3 Direkte (absolute) Adressierung

Die Speicheradresse, unter der ein Operand im Speicher gefunden und gelesen bzw. beschrieben werden kann, wird häufig erzeugt aus Informationen, die z.B. im OPCODE, in einem Register oder an einem anderen Speicherplatz stehen. Jede dieser Adreßinformationen für sich alleine genommen stellt noch keine endgültige Speicheradresse dar. Zur Unterscheidung wird der Begriff der Effektiven Adresse eingeführt. Die ***Effektive Adresse (EA)*** bezeichnet im folgenden die endgültige Speicheradresse eines Operanden im Speicher.

Die direkte oder absolute Adressierungsart ist die, die wir schon bei unserem Modellrechner kennengelernt haben. Die Adresse des Operanden wird direkt im Adreßteil des Befehls spezifiziert. Abb. 7.22 zeigt die Langform dieser Adressierungsart im 6809. Der OPCODE wird von einer 16-bit-Adresse gefolgt. Damit kann jede Adresse im gesamten Adreßraum des 6809 direkt erreicht werden.

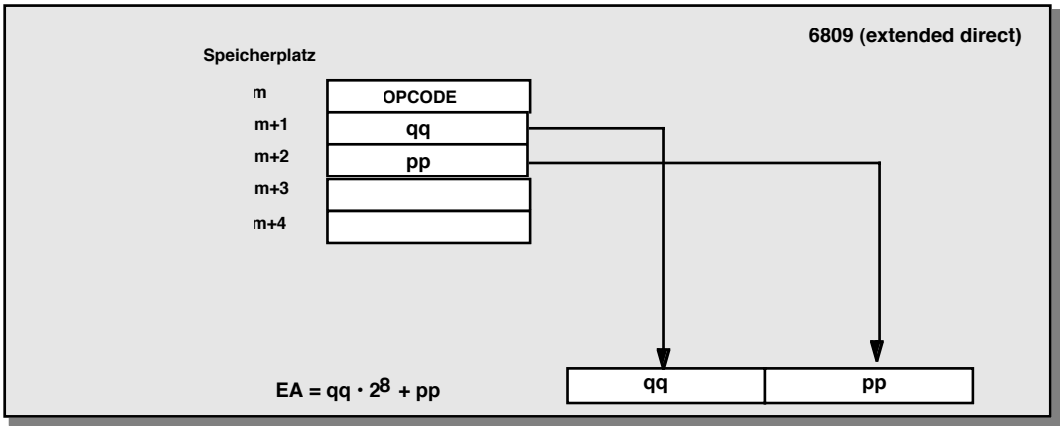


Abb. 7.22 Direkte Adressierung (Langform (engl. extended))

Die effektive Adresse EA wird durch Konkatenation der beiden Post Bytes nach dem OPCODE gebildet. Das erste Post Byte (qq) gibt das höherwertige Byte, das zweite Postbyte (pp) das niederwertige Byte der EA an. Diese Adressierungsart wird im 6809 Context als ***Extended Addressing*** bezeichnet.

Eine Variante der direkten Adressierung gibt Abb. 7.23 an.

Die effektive Adresse EA wird durch Konkatenation einer kurzen 8-Bit-Adresse nach dem OPCODE mit dem Inhalt des Direct Page Registers (DP) gebildet. Die kurze Adresse gibt das niederwertige Byte, der Inhalt des DP das höherwertige Byte der EA an. Das DP muß vorher dynamisch durch einen Transferbefehle z.B. TFR A, DP geladen werden.

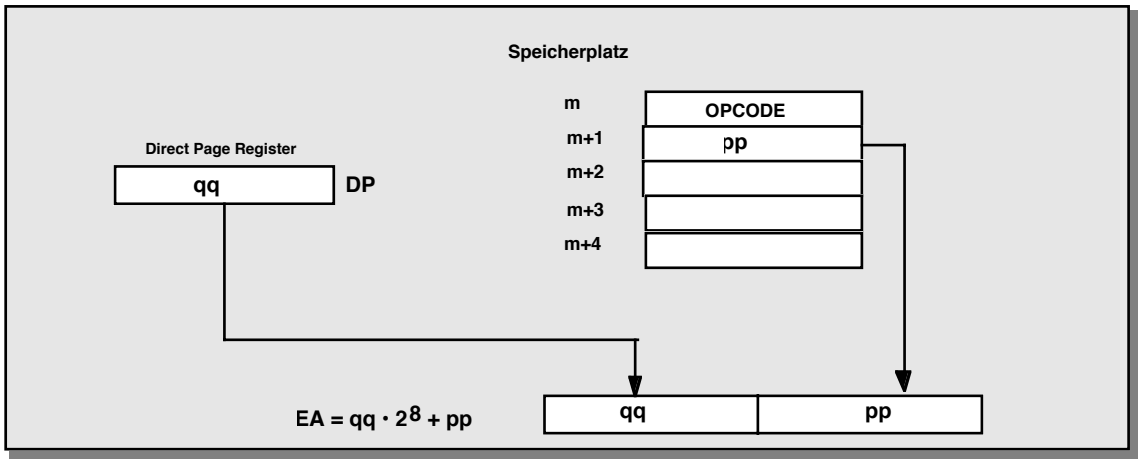


Abb. 7.23 Direkte Adressierung bezogen auf eine Basis-Seite (Base Page Direct)

Diese Adressierungsart bietet gegenüber dem "Extended Addressing" zwei Vorteile:

1.) Man muß nur ein Byte für die Adresse angeben, so daß ein kompakterer Programmcode entsteht.

2.) Dasselbe Programm kann verschiedene Datensätze durch Änderung des Direct Page Registers adressieren.

7.4.4 Indizierte Adressierung

Die Grundform der indizierten Adressierung ist in Abb 7.24 dargestellt. Die effektive Adresse EA wird gebildet aus einer ***Basisadresse*** und einer ***Distanzadresse***, die im Englischen als Displacement oder Offset bezeichnet wird. Durch den indizierten Adressierungsmodus wird die Adresse nicht direkt im Operationscode spezifiziert, sondern im Operationscode wird lediglich angegeben, wie die Adresse gebildet werden muß und wo die Adreßinformationen dafür gefunden werden können. Insbesondere muß spezifiziert werden:

1. Wo die Basisadresse steht.
2. Wo die Distanzadresse steht.

Bei indizierter Adressierung wird die Adresse meist in einem Register angenommen[4], das als ***Indexregister*** oder ***Basisregister*** bezeichnet wird[5]. Wir werden es hier, der Terminologie des MC6809 folgend, als Indexregister bezeichnen. Im 6809 können die Register X, Y, U, S und PC als Indexregister verwendet werden.

[4] Im 6809 ist es auch möglich, daß die Basisadresse im Speicher steht. In diesem Fall liegt indirekte Adressierung vor. Sie wird in 6.4.5 behandelt.

[5] Die Bezeichnung als Indexregister oder Basisregister wird nicht immer gleich verwendet. So wird das Register, in dem die Basisadresse steht, im 6809 als Indexregister, im 68020 als Basisregister bezeichnet. In der 68020 heißt das Register, das die Distanzadresse enthält, Indexregister.

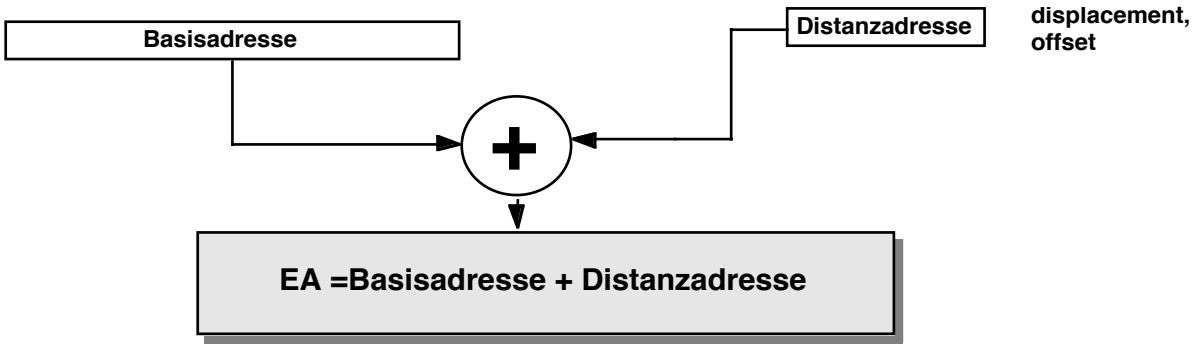


Abb. 7.24

Für die Spezifikation der Distanzadresse gibt es im 6809 folgende Möglichkeiten:

1. Sie steht im Speicher als Teil der Instruktion, d.h. als Literal.
2. Sie steht in einem Register, A, B, D.
3. Sie hat einen festen Wert, +1, +2, -1, - 2, dh. Autoincrement der Basisadresse um 1 oder 2 Byte oder Autodecrement um 1 oder 2 Byte.

Die Kombinationsmöglichkeiten von Basis- und Distanzadresse sind in Abb. 7.25 aufgelistet.

Basisadresse	Distanzadresse	Bezeichnung
(Index-) Register X, Y, U, S, PC	Literal 0, 5, 8, 16 Bit 8, 16 Bit	Adressierung mit konstanter Distanzadresse vom Indexregister “Constant Offset Mode “
(Index-) Register X, Y, U, S	Register A, B, D	Adressierung mit Distanzadresse aus dem Accumulator A, B oder D “Accumulator Offset“
(Index-) Register X, Y, U, S	fest ±1, ±2	Autoincrement / Autodecrement POST increment / PRE decrement

Abb. 7.25 Kombinationen von Basisadresse und Distanzadresse

Das Postbyte selektiert eine dieser Möglichkeiten. Die Codierung des Postbytes ist zusammen mit der 6809 Befehlscodierung im Anhang des Scripts zu finden. Abb. 7.26 gibt die Assemblernotation für die jeweilige indizierte Adressierungsform an. Einige der hier gezeigten Adressierungsmodi, z.B. indiziert mit Indirektion und Programmzähler-relative Adressierung werden an späterer Stelle beschrieben.

Notation	Bedeutung
,R	das Register “R“ (X, Y, U, S, PC) ist als Indexregister spezifiziert
offset ,R	offset wird zum Indexregister “R“ addiert
label ,PCR	Programmzähler-relative Adressierung - die Distanz zwischen PC und Label wird in der Assemblierungsphase berechnet und als Konstante Distanz-adresse im Modus: “Konstanter PC-Offset“ eingesetzt
,R+(+)	Autoincrement um 1 (bzw. 2) des Registers “R“ (X, Y, U, S) (Postincrement)
, -(-)R	Autodecrement um 1 (bzw 2) des Registers “R“ (X, Y, U, S) (Predecrement)
[. . . . .]	indizierte Adresse ist eine indirekte Adresse

Abb.7.26 Assemblernotationen für 6809 indizierte Adressierungsmodi

Die Anwendung der indizierten Adressierung wir in den folgenden drei Beispielen gezeigt. Abb. 7.27 gibt ein Assemblerfragment an, das eine Liste von Zeichen nach einem bestimmten Zeichen durchsucht. Jedes Zeichen ist in einem Byte kodiert. Die Adressierungsart "indiziert mit Autoincrement um 1" kann direkt angewandt werden, um die Listenelemente der Reihe nach zu adressieren.

```
        LDX     HEAD_OF_LIST
COMP    LDA     ,X+             Indiziert mit (Post-) Autoincrement
        CMPA    VW
        BNE     COMP
        .
        .
        .
VW      FCC     "w"
```

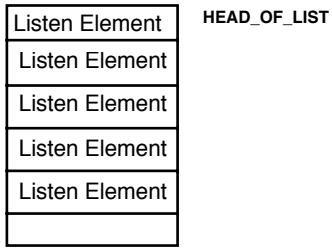


Abb. 7.27 Beispiel für indizierte Adressierung mit Autoincrement

Ein Beispiel der Adressierung einer Liste, in der die Listenelemente größer als ein Byte sind, ist in Abb. 7.28 angegeben. Hier wird die indizierte Adressierung mit einer Distanzadresse aus dem Akkumulator angewandt.

```
        LDX     HEAD_OF_LIST
        LDA     Sem_O
COMP    LDB     A ,X            Indiziert mit Register A Offset
        CMPB    S_zahl          Falls Semesterzahl < 12 :
        BHI     OUTPUT          Ausgabe
        ADDA    RL              Addieren der Record-Länge
        CMPA    #End-List               Ende der Liste erreicht ?
        BHI     Fin
        BRA     COMP
Fin
        .
        .
        .
RL      FCB     32
Sem_O   FCB     31              Offset für Semesterzahl
S_zahl  FCB     12              Vergleichswert für Semesterzahl
```

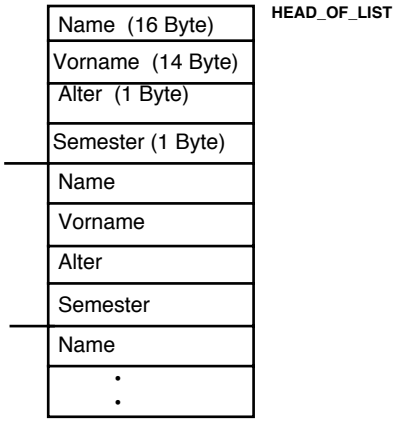


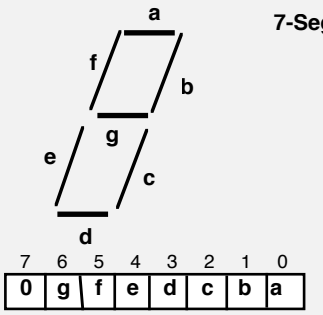
Abb. 7.28 Beispiel für indizierte Adressierung mit Accumulator Offset

Das Beispiel in Abb. 2.29 zeigt ein Unterprogramm zur Indizierung einer Code-Tabelle. Die Ziffern 0-9 werden in den entsprechenden 7-Sement Code zur Steuerung einer Anzeige umgewandelt.

```
SP. Adr  OPC OP          SP.M  MNE.   OP              Kommentar
A000     3F              CLRB                         setzt den Fehlercode : löschen des Displays
A001     96   A0         LDA   $A0                    Wert in den Acc. A holen
A003     81   09         CMPA  #9                     ist der Wert eine Ziffer 0…9 ?
A005     22   05         BHI   DONE                   wenn A > 9 bleibt der Fehlercode gesetzt
A007     8E   AF00       LDX   #SSG                   Lade X  mit dem Beginn der Conversionsliste "SSG"
A00A     E6   86         LDB   A, X                   Lade 7-Segment Muster in Acc A
A00C     D7   ..... DONE STB   MMSEG                  Abspeichern auf ein Ausgabegerät
A010     39              RTS                          Return from Subroutine

AF00              SSG    FCB   $3F, $06, $5B, $4F, $66
AF05                     FCB   $6D, $7D, $07, $7F, $6F

.....             MMSEG  .....  .....                 Adresse des Ausgabegeräts
```



7-Segment Code:

0	3F
1	06
2	5B
3	4F
4	66
5	6D
6	7D
7	07
8	7F
9	6F

Zuordnung von Segmenten zu Bitpositionen im Ausgabewort

Abb. 7.29 Beispiel zur Codeumwandlung mit indizierter Adressierung

7.4.5 Indirekte Adressierung

Die indirekte Adressierung kann als eine Form der indizierten Adressierung aufgefaßt werden, bei der die Basisadresse nicht in einem Indexregister steht, sondern im Speicher. Die dem OPCODE folgende Adresse gibt nicht an, wo sich die Daten für den Befehl befinden, sondern die Adresse, die auf die Daten zeigt.

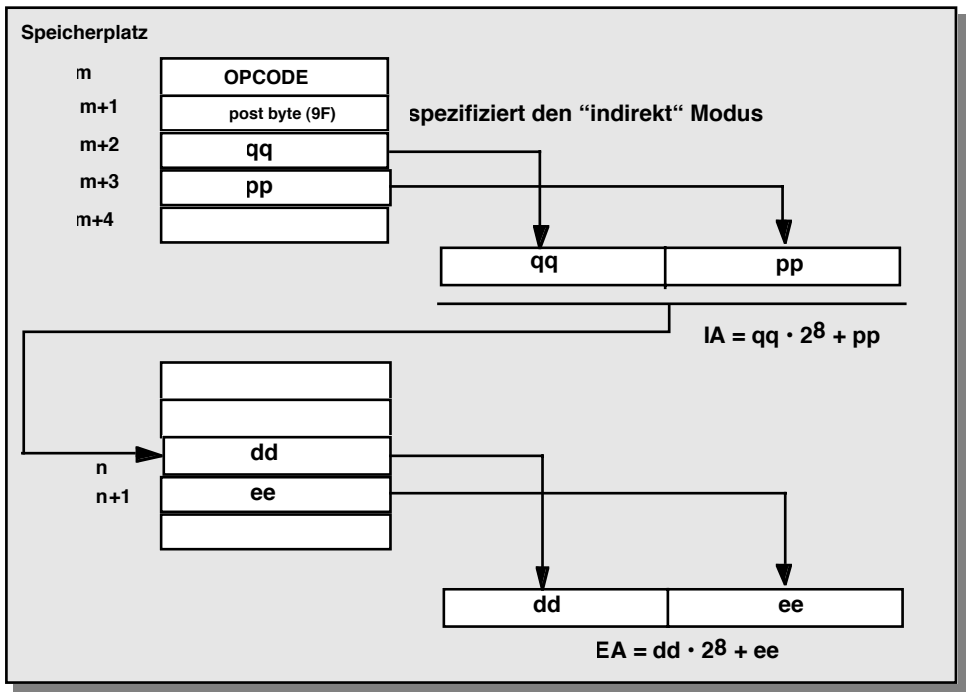


Abb. 7.30 Indirekte Adressierung

Abb. 7.30 zeigt die Grundform der indirekten Adressierung. Zunächst wird der indirekt Modus im Postbyte des Befehls spezifiziert. Die folgenden Bytes enthalten die Speicheradresse, an denen die EA abgespeichert ist. Sie wird schließlich zur Adressierung des Operanden benutzt. Indirekte Adressierung wird beispielsweise angewandt, wenn nicht genug Indexregister zur Verfügung stehen.

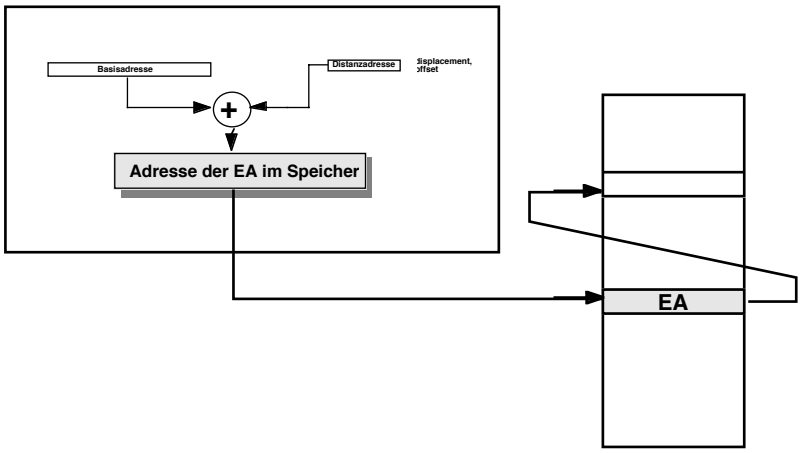


Abb. 7.31 Indiziert indirekte Adressierung

Eine komplexe Form der indirekten Adressierung ist die ***indiziert indirekte Adressierung***, die in Abb. 7.31 dargestellt ist. Die Adressierung des Operanden erfolgt in zwei Stufen. In der ersten wird die Adresse der EA durch indizierte Adressierung ermittelt. In der zweiten Stufe wird mit der EA der Operand selektiert. Ein Beispiel, in dem diese Adressierungsform benutzt wird ist in Abb. 7.32 angegeben.

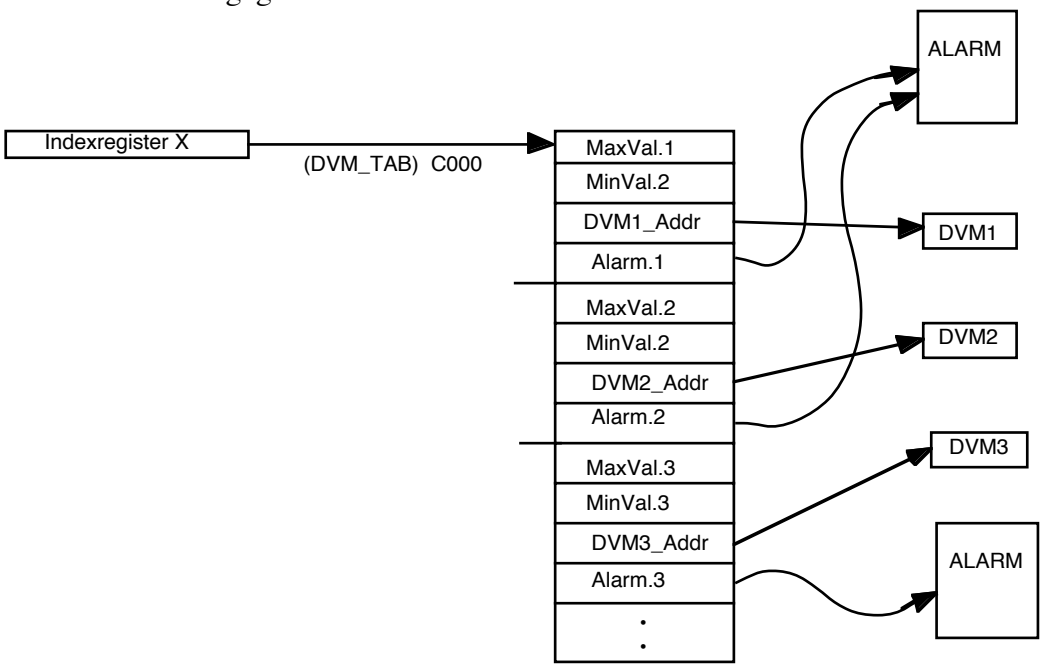


Abb. 7.32 Auslesen eines DVM-Wertes, Vergleich mit dem Maximalwert und Auflösung des Alarms

```
DVM_TAB         EQU   $C000
DVM-1_BASE      EQU   $0
DVM-2_BASE      EQU   $8
DVM-3_BASE      EQU   $10
•
•
MaxVal          EQU   0
MinVal          EQU   2
DVM_INDEX       EQU   4
Alarm           EQU   6


        LDX     #DVM_TAB
        LDB     #DVM-1_Base
        ABX
        LDA     [DVM_INDEX, X]
        CMPA    MaxVal, X
        BLE     OK
        JMP     [Alarm, X]
OK      •
        •
```

Abb. 7.32 Beispiel für den Adressierungsmodus "Indiziert Indirekt"

Mehrere Digitalvoltmeter sollen ausgelesen und mit eine Maximalwert verglichen werden. Bei Überschreitung des Maximalwertes soll ein Alarm ausgelöst werden. Jedes Digitalvoltmeter (DVM) belegt selbst mehrere Adressen. DVM1 und DVM2 benutzen denselben Alarm. Um die Adresse des entsprechenden Gerätes, das den Alarm auslöst, nicht direkt fest in den Programmcode "hineinzuprogrammieren", wird indiziert indirekte Adressierung eingesetzt. So ist es möglich, die Adresse des Gerätes, das den Alarm auslöst, zu ändern, indem man die Tabelle für die DVMs ändert und nicht den Programmcode. Außerdem kann derselbe Code für verschiedene DVMs mit unterschiedlicher Konfiguration genutzt werden. Dies ist nur mit der zusätzlichen Indirektion des indiziert indirekten Adressierungsmodus möglich.

7.4.6 Relative Adressierung

Eine ***relative Adresse*** wird bezogen auf den augenblicklichen Inhalt des PCs definiert. Sie stellt eine Distanzadresse dar. Alle relativen Sprünge nutzen diese Adressierungsart. Relative Adressen werden im 2er-Komplement dargestellt, so daß sowohl positive als auch negative Distanzen zum PC spezifiziert werden können. Relative Adressierung hat den Vorteil der Unabhängigkeit von einer bestimmte Speicheradresse. Ein Programm, das alle notwendigen Informationen relativ spezifiziert, kann im Speicher beliebig verschoben werden, ohne daß Adressen geändert werden müssen. Relative Adressierung ist die Basis für positionsunabhängigen Code. Wir werden in Abschnitt 8.3 darauf noch genauer eingehen.

7.5 Zusammenfassung

Abb. 7.33 gibt einen Überblick über die vorgestellten Adressierungsarten. Die Aufgabe der Adressierung ist die Spezifikation eines oder mehrerer Operanden. Für den Direktoperanden wird keine zusätzliche Adreßspezifikation benötigt, da er direkt im Befehlswort enthalten ist. Implizite Adressierung legt bereits im OPCODE die Adresse(n) des oder der Operanden fest. ABX z.B. addiert den Inhalt von Accumulator A zum Indexregister X. Der Vorteil ist ein besonders kurzer Befehl, der die Registeradressen nicht mehr explizit enthalten muß. Dies ist natürlich nur bei häufig verwendeten Operationen sinnvoll.

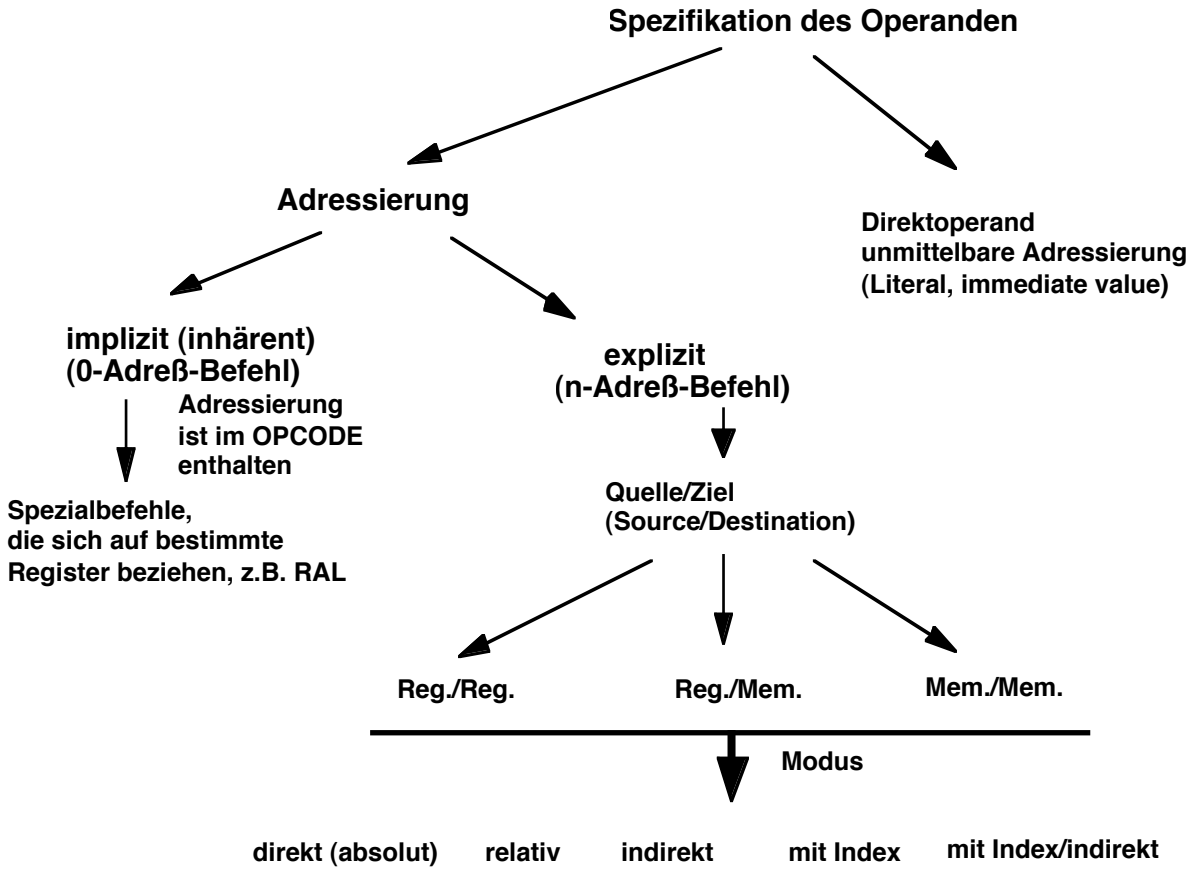


Abb. 7.33 Adressierungsarten

Die anderen Variationen der expliziten Adressierung wurden besprochen. Wir haben dabei die Akkumulatormaschine MC6809 als Beispiel angenommen. Bis auf die Befehle zum Transfer von Registerinhalten, in denen zwei Registeradressen angegeben werden, wird in einem Befehl immer nur eine Adresse spezifiziert, da bei zweistelligen Operationen angenommen wird, daß einer der Operanden im Accumulator steht. Wir werden später CPUs kennenlernen, die in einem Befehl explizit zwei Adressen für Operanden und eine Zieladresse für das Resultat einer Berechnung angeben. In diesem Fall liegt ein Drei-Adreß-Befehl vor.

8 Prozessornahe Programmierung

Dieses Kapitel soll die Programmierung auf Assemblerebene vertiefen. Der eingeführte MC6809 dient dazu als Zielmaschine. Die eingeführten Techniken der prozessornahen Programmierung sollen einmal die Maschinenprogrammierung erleichtern, zum anderen sollen sie zeigen, wie ein Compiler eine Hochsprache in entsprechende Maschinenkonstrukte übersetzen könnte.

8.1 Kontrollkonstrukte

Eine der Fehlerquellen bei der Programmierung auf Assemblerebene stellt die korrekte Programmierung von Kontrollkonstrukten dar. Wir werden daher mit der Diskussion dieser Konstrukte beginnen, wobei wir uns an den üblichen Kontrollkonstrukten in Hochsprachen orientieren.

Eine Assemblerrealisierung des Kontrollkonstrukts IF . . .THEN ist in Abb. 8.1 dargestellt. Wenn die Bedingung zutrifft, wird der Programmteil "Aktion" ausgeführt. Es ist zu beachten, daß der bedingte Sprung dann ausgeführt wird, wenn die Bedingung NICHT zutrifft.

```
IF <Bedingung> THEN <Aktion>

CMP   a, b          a: Register A,B,D,S,U,X,Y    b:Speicheradresse
Bxx   EXIT          Branch on Condition FALSE
        •
        •           Aktion
        •
EXIT  Continue
```

Abb. 8.1 Assemblerumsetzung von IF.....THEN

Bedingung	mit Vorz.	ohne Vorz.	Branch Mnemonic Bxx
a=b	x	x	BNE
a≠b	x	x	BEQ
a>b	x		BLE
a≥b	x		BLT
a<b	x		BGE
a≤b	x		BGT
a>b		x	BLS
a≥b		x	BLO
a<b		x	BHS
a≤b		x	BHI

Abb. 8.2 Auswahl des Sprungbefehls

In Abb. 8.2 werden die unterschiedlichen Bedingungen und die dazugehörigen Branchbefehle aufgelistet. Da der Sprung ausgeführt werden soll, wenn die Bedingung NICHT zutrifft, muß jeweils der zur Bedingung komplementäre Branchbefehl verwendet werden, also z.B. bei der Gleichheitsbedingung der BNE (Branch not equal) Befehl.

In den folgenden vier Abbildungen sind häufig verwendete Kontrollstrukturen und ihre Assemblerumsetzung dargestellt:

IF a<b THEN aktion1 ELSE aktion2

```
          CMP     a,b
          BGE     aktion2
            •
            •               Ausführung
            •               von aktion1
          BRA     EXIT
aktion2     •
            •               Ausführung
            •               von aktion2
EXIT
```

Abb. 8.3 Das IF-THEN-ELSE-Konstrukt

Beim IF-THEN-ELSE-Konstrukt wird durch die Bedingung eine von zwei Alternativen ausgewählt. Die Kontrollstrukturen in Abb. 8.4, 8.5 und 8.6 kontrollieren Programmschleifen. Dabei wird in der FOR-Schleife explizit eine Zählervariable gesetzt, die bei jedem Durchlauf der Schleife inkrementiert wird[1]. In der WHILE-Schleife und der REPEAT-UNTIL-Schleife wird allgemein eine Bedingung abgefragt. Die Abfrage der Bedingung in der WHILE-Schleife wird direkt am Beginn der Schleife durchgeführt, so daß die Schleife möglicherweise kein Mal ausgeführt wird. Die unterscheidet sie von der REPEAT-UNTIL-Schleife, die mindestens einmal durchlaufen, und bei der die Bedingung am Ende der Schleife abgeprüft wird.

FOR K = I TO J DO aktion

```
          LDB     #I
REPEAT
            •
            •               Ausführung
            •               von aktion
          INCB
          CMPB    #(J+1)
          BNE     REPEAT
EXIT
```

Abb. 8.4 Die FOR-WHILE-Schleife

[1]Es ist zu beachten, daß in mehreren Sprachen (z.B. in PASCAL) eine FOR-Schleife evtl. gar nicht durchlaufen wird. Dieser Fall wird vom gezeigten Muster der FOR-Schleife jedoch nicht abgedeckt.

```
WHILE  a>b  DO aktion

REPEAT  CMP    a,b
        BLE    EXIT
        •
        •              Ausführung
        •              von aktion
        BRA    REPEAT
EXIT
```

Abb. 8.5 Die WHILE-DO-Schleife

```
REPEAT aktion UNTIL a=b

REPEAT
        •
        •              Ausführung
        •              von aktion
      CMP    a,b
      BNE    REPEAT
EXIT
```

Abb. 8.6 Die REPEAT-UNTIL-Schleife

Das CASE-Statement in Abb. 8.7 erlaubt die Auswahl einer Programmsequenz aus mehreren Alternativen. Ausgehend von dem Index der Programmsequenz wird ein indirekter Sprung über eine Tabelle von Eintrittspunkten zu ihrem Anfang durchgeführt. Abb. 8.7.a zeigt die Organisation der Tabelle mit Einsprungadressen, wobei "hh" das höherwertige, "ll" das niederwertige Byte der jeweiligen 16-bit-Adresse angibt.

Abb. 8.7.a Organisation der Einsprungtabelle

```
B enthält den CASE Index : 3

            CMPB   #N           gültige Eingabe ?
            BHI    EXCEPTION
            LSLB                Berechng .d. Offsets in die Tabelle (ein Eintrag:2Byte)
            LDX    #aktion0
            ABX                 Addiere Offset in B zu X
            JMP    [ ,X]        Springe indirekt zum Anfang des Progr.-Teils "aktion3"
EXCEPTION
```

Abb. 8.7.b Programmfragment zur Indizierung der Einsprungtabelle

Abb. 8.7.b gibt das Assemblerprogramm zur Indizierung der Tabelle an. Falls der Index größer als eine vorgegebene Obergrenze oder negativ ist, wird die Routine nicht durchgeführt. Es kann dann zu einem geeigneten Programm zur Ausnahmebehandlung gesprungen werden.

8.3 Positionsunabhängiger Objektcode

Werden im Assemblerprogramm feste Adressen spezifiziert, z.B. durch "LDA $BA00", wird unabhängig davon, wo der Objektcode im Speicher steht, durch den Ladebefehl immer auf die Adresse $BA00 zugegriffen. Will man unter diesen Voraussetzungen z.B. eine Objektcodebibliothek erstellen, müßte man für alle Programme dieser Bibliothek im voraus festlegen, welche Adressen im Speicher sie jeweils belegen dürfen, damit es nicht zu Konflikten kommt. Da man die Bibliotheksprogramme aber in der Regel unabhängig voneinander erstellen, oder sogar fremde Programme einbinden möchte, ist eine solche Festlegung nur schwer möglich. Es ist daher wünschenswert, Objektcode frei im Speicher verschieben zu können. Die Voraussetzung dafür ist, daß Programm und Daten nicht mit absoluten Adressen, sondern relativ zu eine dynamisch festlegbaren Position spezifiziert werden können.

Die Mechanismen zur Unterstützung positionsunabhängigen Codes, die der 6809 zur Verfügung stellt, sind:

- Relative Sprünge (Branches)
- Befehle zur Adreßmanipulation (insbesondere LEAx)
- Adressierung von Speicher durch "Konstante Distanz" vom Programmzähler
- Nutzung des Hardware-Stacks als temporären Speicher

8.3.1 Relative Sprünge

Wie wir in Kapitel 7 gesehen haben, werden alle Branchbefehle relativ zum aktuellen Inhalt des Programmzählers durchgeführt. Dabei wird die Sprungdistanz als 2er-Komplement-Zahl

aufgefaßt, so daß vorwärts und rückwärts gesprungen werden kann. Die Branchbefehle erfüllen also die Voraussetzung für positionsunabhängigen Objektcode.

8.3.2 Befehle zur Adreßmanipulation

Der zentrale Befehl zur Adreßmanipulation ist der LEAx-Befehl (Load Effective Address). Der LEAx-Befehl erlaubt arithmetische Operationen auf den Indexregister X, Y, S, U. LEAx bildet eine effektive Adresse. Anstatt sie aber zur Adressierung zu verwenden, wird sie in eines der Indexregister geladen. Im nächsten Befehl kann dann über eine indizierte Adressierung auf die gebildete Adresse zugegriffen werden. Da LEAx auch den Inhalt des Programmzählers in ein Indexregister laden und eine Distanz addieren oder subtrahieren kann, ist eine relative Adressierung möglich. Abb. 8.8.a zeigt einige Beispiele für die Nutzung des Befehls.

```
LEAX, LEAY, LEAS, LEAU

Beisp.:
            •       LEAX        1,X         Increment X
            •       LEAY        -1,Y        Decrement Y
            •       LEAU      $ABCD,U       Addiere $ABCD zum U-SP
            •       LEAX        0,PC        äqu. zu TFR  PC,X
```

a. Beispiele für die Verwendung von LEAx

```
            Addr.  OPCODE      OP-Addr.  Mnemonic

Beisp.:     0100   30    8D     0109      START  LEAX   TABLE,PCR
            0104   A6    80               LOOP   LDA    ,X+
            0106

            020D                          Table  FCC    /table of whatever/
```

b. Positionsunabhängige Indizierung einer Tabelle

Abb. 8.8 Der Befehl LEAx

Ein Beispiel für die positionsunabhängige Adressierung einer Tabelle ist in Abb. 8.8.b angegeben. Hier wird die Adressierungsart PCR (Program Counter Relativ) benutzt. PCR ist eine Besonderheit des 6809 Assemblers zur automatischen Berechnung der Distanz einer Marke (in diesem Fall "Table") vom Programmzähler. Der Anfang der Tabelle “Table“ hat eine Distanz von $10D vom START. Der Assembler berechnet die Distanz $109, da er die Distanz vom um 4 incrementierten PC berücksichtigt. Durch die Assemblerdirektive “PCR“ wird der Befehl “LEAX TABLE,PCR“ vom Assembler in "LEAX offset, PC", d.h. "30 8D offset" übersetzt.

```
B enthält den CASE Index : 3

                CMPB    #N              gültige Eingabe ?
                BHI     EXCEPTION
                LSLB                    Berechng .d. Offsets in die Tabelle (ein Eintrag:2Byte)
                LEAX    aktion0,PCR     Lade Indexregister mit Anfang d. Tabelle relativ zum PC
                JMP     [B ,X]          Springe indirekt zum Anfang des Progr.-Teils "aktion3"
                                        (B wird Acc.Offset  automatisch zu X addiert)
EXCEPTION
```

Abb. 8.9 Programmieralternative zu 8.7.b

Abb. 8.9 stellt eine verbesserte Version des Programms in Abb. 8.7.b dar, die den Befehl LEAx und die Adressierungsart PCR ausnutzt. In 8.7.b wird die Tabelle mit Einsprungadressen fest im Programm durch ein Literal (#action0) spezifiziert. Hier wird die Tabelle relativ zum Programmzähler adressiert. Außerdem wird beim Sprung der automatische Akkumulator Index ausgenutzt. Dadurch kann ein Befehl eingespart werden.

8.3.3 Nutzung des System-Stacks als temporärer Speicher

Bisher haben wir Methoden kennengelernt, wie man Speicher zur Assemblierungszeit reservieren kann, z.B. durch die Assemblerdirektiven RMB, FCC und FCB. Dies nennt man auch ***statische Reservierung(static allocation)***, da man sie zur Laufzeit des Programms nicht mehr ändern kann. Bei der Assemblierung werden feste Adressen erzeugt, die durch die Direktiven festgelegt sind. Der so reservierte Speicher kann während der Programmlaufzeit nicht (ohne Gefahr) anderweitig genutzt werden. Auch wenn das entsprechende Programm nicht aktiv ist, ist es gefährlich, diesen Speicherbereich zu benutzen, da bei Aktivierung des Programms die dort abgelegten Daten anderer Programme überschrieben werden.

Bei der dynamischen Reservierung (dynamic allocation) wird Speicher während der Laufzeit reserviert, genutzt und dann wieder freigegeben. Die Reservierung muß so erfolgen, daß keine Daten anderer Programme beeinflußt werden. Die dafür am besten geeignete Speicherstruktur ist der Stack. Der Stack wird von einem Programm oder einem Programmteil immer nur in eine Richtung, z.B. in absteigender Adreßfolge, benutzt. Die von anderen Programmen belegten Teile des Stacks werden deshalb nicht berührt. Abb. 8.10 zeigt die Reservierung von Speicherplatz auf dem Stack.

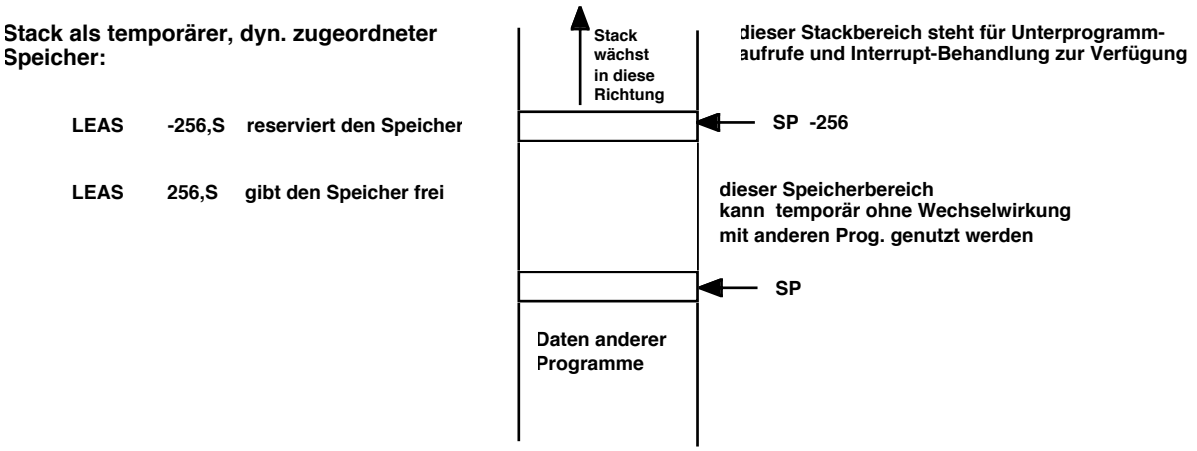


Abb. 8.10 Dynamische Speicherreservierung auf dem Stack

Wir nehmen an, ein Programm benötigt einen Speicherbereich von 256 Byte, der dynamisch reserviert werden soll. Wir benutzen den Befehl "LEAS -256,S", der den Stackpointer um 256 dekrementiert. Da der Stack immer in die Richtung der niedrigen Adressen wächst (z.B. bei Unterprogrammaufrufen) bleibt der reservierte Bereich davon unbeeinflußt. Wir können nun den reservierten Bereich relativ zum Stackpointer adressieren. Wir werden diese Technik im nächsten Abschnitt noch genauer betrachten.

8.3.4 Unterprogrammtechniken

Unterprogramme isolieren häufig im Programm verwendete Funktionen. Zusammengefaßt sind die Vorteile:

- Modulare Strukturierung nach funktionalen Gesichtspunkten,
- Wiederverwendung von Programmen oder Programmstücken,
- Unterprogrammbibliotheken,
- Verbesserung der Testbarkeit,
- Reduzierung des Speicherbedarfs durch Code-Sharing.

Es soll hier nicht weiter auf die Aspekte des Software-Engineerings eingegangen werden, sondern wir wollen die grundsätzlichen Programmiertechniken auf Assemblerebene einführen. Abb. 8.11 zeigt die typische Einbettung eines Unterprogramms. Der Aufruf erfolgt aus dem Hauptprogramm durch einen Unterprogrammsprung. Dabei werden folgende Aktionen notwendig:

1.) Sichern der Rücksprungadresse. Der Inhalt des Programmzählers (der bereits auf die nächste Instruktion zeigt) wird abgespeichert.
2.) Evtl. sichern des Prozessorzustands. Dies ist notwendig, wenn der Prozessorzustand nach Rückkehr aus dem Unterprogramm im Hauptprogramm wieder zur Verfügung

stehen soll. Der Prozessorzustand ist durch die Inhalte der Prozessorregister gegeben. Es muß allerdings nicht immer der gesamte Prozessorzustand gesichert werden, wenn man weiß, welche Register, in denen Information für das Hauptprogramm gespeichert ist, vom Unterprogramm tatsächlich genutzt werden. Außerdem muß sichergestellt werden, daß das Unterprogramm keine anderen Daten des Hauptprogramms im Speicher überschreibt, weil zufällig dieselben Adressen verwendet werden. Hier müssen die in Abschnitt 8.3.3 eingeführten Techniken des positionsunabhängigen Codes eingesetzt werden. Insbesondere der Stack eignet sich für die temporäre Speicherung von Daten, da es, bei korrekter Verwaltung, nicht vorkommen kann, daß sein Inhalt von einem Unterprogramm zerstört wird.

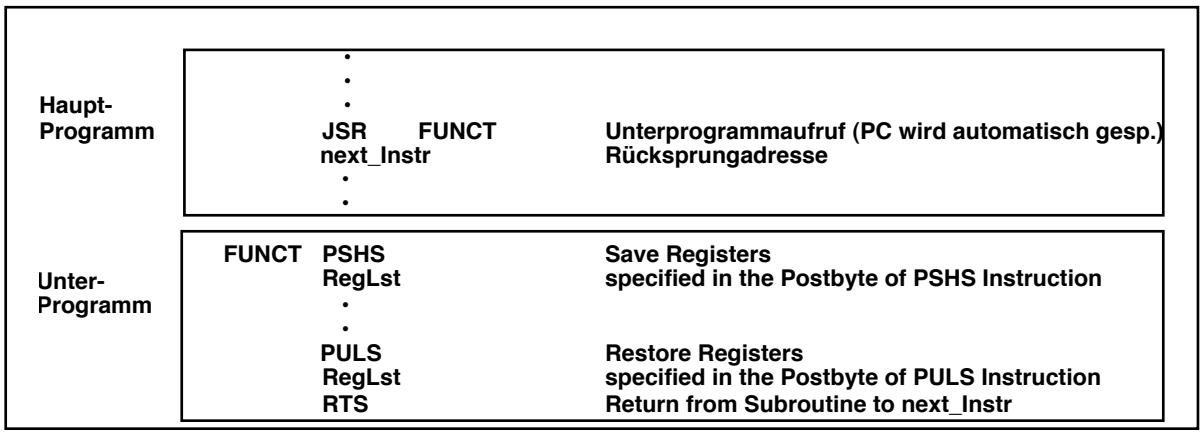


Abb. 8.11 Typische Programmsequenz beim Aufruf eines Unterprogramms

3.) Übergabe der Parameter. Das Unterprogramm benötigt in den meisten Fällen Eingabedaten, die es bearbeiten soll. Diese müssen ihm vom Hauptprogramm zur Verfügung gestellt werden. Für die Übergabe von Parametern gibt es zwei grundsätzliche Verfahren:

- call-by-value: Die Parameter für das Unterprogramm werden beim Aufruf als Wert übergeben, z.B. auf bestimmten, dem Unterprogramm bekannten Stellen auf dem Stack oder an dem Unterprogramm bekannten Stellen im Speicher.
- call-by-reference: Es wird ein Zeiger übergeben, wo die Parameter zu finden sind.

Im ersten Fall müssen die Parameter meist explizit an eine bestimmte Stelle kopiert werden, was zusätzlichen Aufwand erfordert. Die meist verwendeten Speicherstrukturen für die Übertragung von Parametern sind:

- Register
- Stack
- Fester Speicherbereich

Abb. 8.11 zeigt ein Unterprogramm zur Mehrbyte-Addition. Den beiden Summanden und der Summe sind feste Speicherbereiche zugeordnet (Summand 1: 0048-004B, Summand 2: 004C-004F, Summe: 0050-0053). Im Hauptprogramm befindet sich eine Initialisierungssequenz, welche die entsprechenden Anfangsadressen und die Anzahl der Bytes, die im Unterprogramm addiert werden sollen, in die Indexregister bzw. den B-Accumulator des 6809 lädt. Die Parameterübergabe erfolgt also durch die Übergabe von Referenzen.

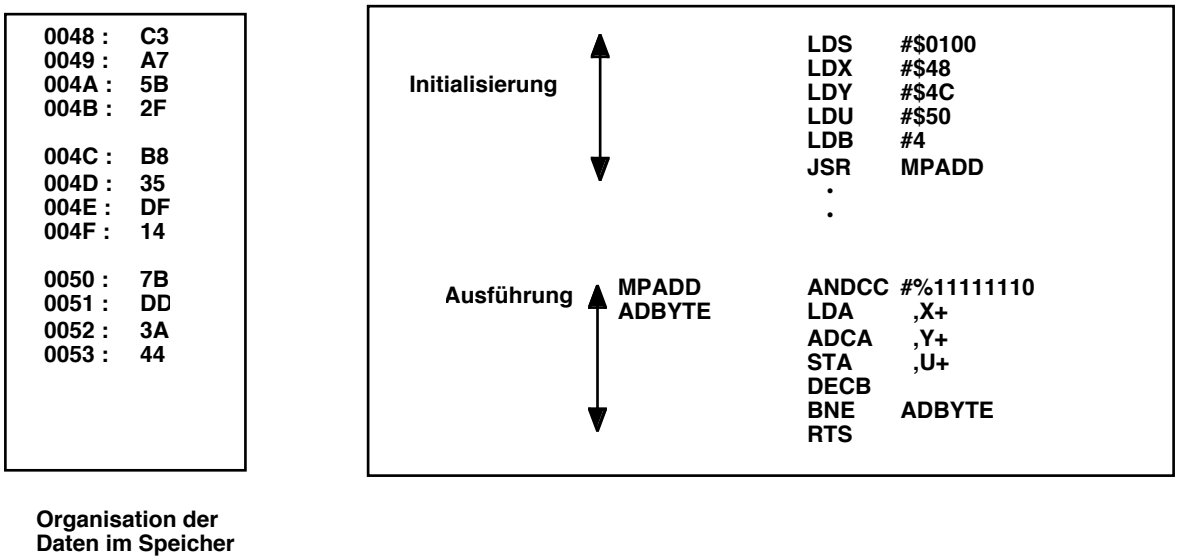


Abb. 8.11 Unterprogramm zur Mehrbyte-Addition

Im Unterprogramm selbst werden explizit keine Register mehr gesichert. Es muß also gewährleistet sein, daß der Accumulator A keinen für das Hauptprogramm relevanten Wert gespeichert hat, da er sonst am Anfang des Unterprogramms z.B durch ein PSHS A gesichert werden müßte. Die erste Zeile des Unterprogramms (ANDCC ...) stellt sicher, daß kein $Carry_{IN}$ bei der Addition der niederwertigsten Bytes berücksichtigt wird.

Im folgenden sind zwei Beispiele für Unterprogramme zur Berechnung der Fakultätsfunktion angegeben. Das erste zeigt, wie Schachtelung, insbesondere Rekursion von Unterprogrammen realisiert werden kann. Zum Vergleich ist noch ein iteratives Programm hinzugefügt. Rekursion erfordert die dynamische Allokation von Speicher für temporäre Variablen. Da das Unterprogramm sich selbst aufruft, kann kein fester Speicherbereich zur Sicherung von Zwischenergebnissen genutzt werden. Das Unterprogramm würde in jedem Durchlauf die dort gespeicherten und noch benötigten Werte überschreiben. Dasselbe gilt für Register. Nur der Stack, auf dem in jedem Durchlauf ein neuer Bereich alloziiert wird, erfüllt die Anforderungen, wie im Beispiel erläutert wird.

Beispiel: Berechnung der Fakultätsfunktion: n ! für n ≤ 5

(Diese Einschränkung wurde gemacht, weil bei größerem n eine aufwendige Mehrbyte-Multiplikation erforderlich wäre, die hier nicht Gegenstand des Problems ist.)

Mathematische Def. : $0! = 1, n! = n \bullet (n-1)!$

Daraus leitet sich die rekursive Definition der Funktion FACT folgendermaßen ab:

$FACT(0) = 1, FACT(N) = N\ FACT(N-1)$ Nebenbedingung $(0 \le N \le 5)$

In der Programmiersprache C kann die Definition relativ einfach umgesetzt werden, wie Abb. 8.12 zeigt.

```
main ()
{
    printf ("die Fakultät von 5 ist: %d\n", fact (5));
}

int fact (int n)
{
    if (n < 1)
      return (1);

    else
     return (n * fact (n-1));
}
```

Abb. 8.12 Rekursives C - Programm zur Berechnung der Fakultät

N: Parameter der Funktion FACT (N) S
F: Funktionswert S-1

```
 1          LEAS    -1,S          Zuordnung eines temporären Speicherplatzes F  auf dem Stack
 2          LEAX      ,S          Abspeichern der Adresse von F in Register X ( äquivalent zu : TFR   S,X )
 3          LDA     #1            Initialisiere F mit 1
 4          STA       ,X
 5          LDA       1,S         Lade den Parameter N vom Stack
 6          INCA                  Erhöhe um 1, um korrekte Abbruchbedingung zu erzeugen
 7          BSR     FACT          Springe zum Unterprogramm FACT
 8          LDA       ,X          "Retten" des Ergebnisses in A
 9          LEAS    1 ,X          Freigeben des temporären Speicherplatzes auf dem Stack
10 END      ...

11 FACT     DECA                  Decrementiere N
12          BLE     RET           IF (N-1)≤0 THEN RETURN
13          PSHS    A             Speichere das aktuelle N des Unterprogramms auf dem Stack
14          BSR     FACT          Rufe FACT zur Berechnung des nächsten N auf
15          PULS    B             Lade ein N vom Stack nach B
16          LDA       ,X          Lade das bisherige Produkt von F nach A
17          MUL                   A·B
18          STB       ,X          Speichere das Produkt (LSByte von Acc D) in F
19 RET      RTS
```

8.13 Rekursives Assembler - Programm zur Berechnung der Fakultät

Es wird angenommen, daß der Funktionsparameter N auf dem Stack liegt (call-by-value). Zu Beginn des Unterprogramms zeigt der Stackpointer auf die Speicherzelle, die N enthält. In Zeile 1 des Unterprogramms wird ein temporärer Speicherplatz reserviert, um den Funktionswert zwischenzuspeichern. Die Adresse dieser Stackposition wird in Zeile 2 in das Indexregister X gespeichert, um immer darauf zugreifen zu können. In Zeile 3und 4 wird der temporäre Bereich mit dem (vorläufigen) Funktionswert "1" initialisiert. Falls der Funktionsparameter N=0 ist, wird durch das Programm der korrekte Funktionswert ausgegeben. Der Funktionsparameter N wird in das Register A geladen (Zeile 5) und inkrementiert (Zeile 6). Damit ist die Initialisierung des Unterprogramms beendet und der Unterprogrammsprung wird ausgeführt. Dabei wird der Programmzähler (PC) auf den Stack gespeichert. Zum Verständnis des Unterprogramms muß man genau verfolgen, was auf dem Stack geschieht. Dazu ist in Abb. 8.14 die Belegung des Stacks vor dem ersten RTS angegeben (für N = 5)

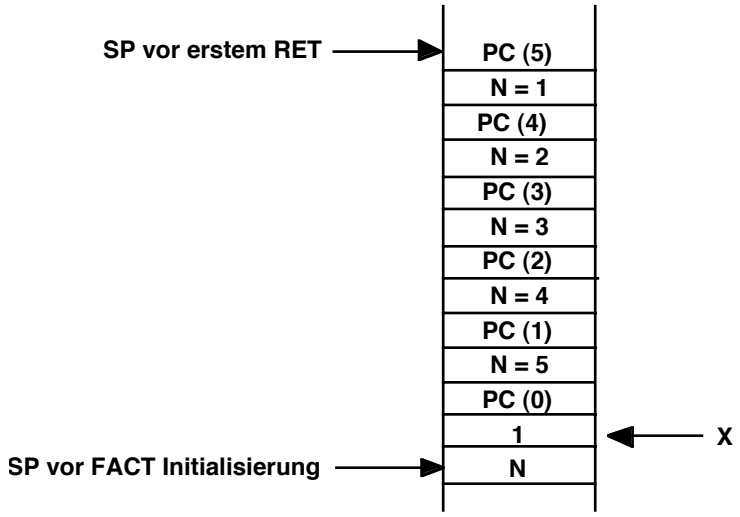


Abb. 8.14 Belegung des Stacks bei der Bearbeitung des Unterprogramms FACT

In Zeile 11 und 12 wird überprüft, ob die Abbruchbedingung (N=0) erreicht wurde. Falls nicht, wird der dekrementierte Wert von N auf den Stack geschrieben und das Unterprogramm ruft sich selbst auf. Dadurch wird wiederum der aktuelle PC auf den Stack gepusht und der Ablauf wiederholt sich so lange, bis N=0 erreicht ist. Der zuletzt auf den Stack geschriebenen Wert von N beträgt dann "1". Abb. 8.14 zeigt die Belegung des Stacks vor dem ersten Return. Die Prüfung der Abbruchbedingung für die Rekursion erfolgt in Zeile 12, von der aus zur Marke RET gesprungen wird. Das jetzt ausgeführte Return nimmt als Rücksprungadresse den aktuellen Wert auf dem Stack, der auf die Zeile 15 zeigt. Was folgt ist die Multiplikation des auf dem Stack gespeicherten Wertes mit dem bisherigen (temporären) Funktionswert. Dies wird für alle Rekursionsstufen durchgeführt. Mit dem letzten Return gelangt man ins Hauptprogramm zurück, wo zunächst das Ergebnis in Register A gesichert wird, bevor der Stack in den Zustand vor der Initialisierung des Programms zurückgesetzt wird.

Rekursive Programme stellen einen eleganten Weg dar, dafür geeignete Probleme zu formulieren. Meist muß jedoch die elegante Implementierung mit einem erhöhten Laufzeitaufwand bezahlt werden, der aus den vielen Unterprogrammaufrufen resultiert. Daher wird zum Vergleich in Abb. 8.15 und 8.16 eine iterative Variante der Fakultätsberechnung in C und Assembler angegeben.

```
main ()
{
    printf ("die Fakultät von 5 ist: %d\n", fact (5));
}

int fact (int n)
{
    f = 1;
    i = 1;
    while (i++ < n)
            f = f * i;
    return (f);
}
```

8.15 Iteratives C - Programm zur Berechnung der Fakultät

```
N: Parameter der Funktion FACT (N)      S
Z: Schleifenzähler                      S - 1
F: Funktionswert                        S - 2

                .
                .
                TFR     S, X        Nutze X als Basisregister zur Adressierung von N
                LEAS    -2, S       Zuordnung von 2 temp. Speicherplätzen für F auf dem Stack
                BSR     FACT
                PULS    A           Funktionswert in Register A retten
                LEAS    2 ,S        Stack "aufräumen"

FACT            LDA     #1          Initialisieren
                STA     -1 , X      des Schleifenzählers Z und
                STA     -2, X       (temporären) Funktionswertes F

REPEAT          LDB     -1, X       Laden des aktuellen Wertes des Schleifenzählers Z
                CMPB    ,X          Vergleich mit dem Funktionsparameter
                BGE     OUT         wenn  Z <  N ist, dann beenden mit Funktionswert = 1
                INCB                Erhöhen des Schleifenzählers
                STB     -1, X       und abspeichern
                LDA     -2 ,X       Laden des (temporären) Funktionswertes
                MUL                 Multiplikation A * B ; erhöhter Schleifenzähler * temp. Funktionswert
                STB     -2 ,X       Abspeichern des neuen (temporären) Funktionswertes
                BRA     REPEAT      Rücksprung zum Anfang der Schleife
OUT             RTS
```

8.16 Iteratives Assemblerprogramm zur Berechnung der Fakultät

Im Gegensatz zum rekursiven Programm hat das iterative Programm explizite Kontrollstrukturen, in diesem Falle eine Schleife, die von 1 bis zum gewünschten Wert läuft. Zwar wird auch hier der Stack zur Speicherung temporärer Variablen genutzt, aber es wäre auch möglich, mit festen Speicheradressen oder Registern zu arbeiten.

9 Eingabe und Ausgabe

Wir haben uns bisher darauf beschränkt, zwei Hauptkomponenten eines Rechners, die CPU und den Speicher, näher zu betrachten. Nun werden wir uns mit der dritten der klassischen Rechnerkomponenten beschäftigen, der Eingabe und Ausgabe. Ohne Ein-/Ausgabe (EA) ist ein Rechner von seiner Außenwelt isoliert. Erst EA macht es möglich, mit einem Rechner zu kommunizieren. Über die Eingabe werden dem Rechner Daten und Programme auf die verschiedensten Arten übermittelt, angefangen von einfachen Schaltern oder einer Tastatur mit Datenraten von wenigen Bit/Sekunde bis hin zu externen Plattenspeichern und Videokanälen mit vielen MBit/Sekunde. Auf der Ausgabeseite reicht die Palette von der einfachen Ausgabe von ASCII-Zeichen auf Drucker über Bildschirme bis wiederum zu schnellen Massenspeichern und Audio- bzw. Videokanälen. An dieser Stelle werden wir uns allerdings nicht mit speziellen Hochleistungs-EA-Schnittstellen befassen, sondern mit der einfachen Steuerung eines peripheren Geräts, wie etwa einem Drucker, oder, im Anwendungsbereich der Überwachungs- und Steueraufgaben, einer Menge von Relais oder Analog/Digitalwandler. Dies wollen wir am Beispiel eines parallelen Schnittstellenadapters für den 8-Bit Mikroprozessor diskutieren.

Bisher haben wir den Bus als Verbindung zwischen Prozessor und Speicher betrachtet. Unser erster Ansatz für die EA soll daher der Anschluß der peripheren Geräte (PG) an den Prozessor/Speicher-Bus sein. Abb. 9.1 zeigt eine solche Konfiguration.

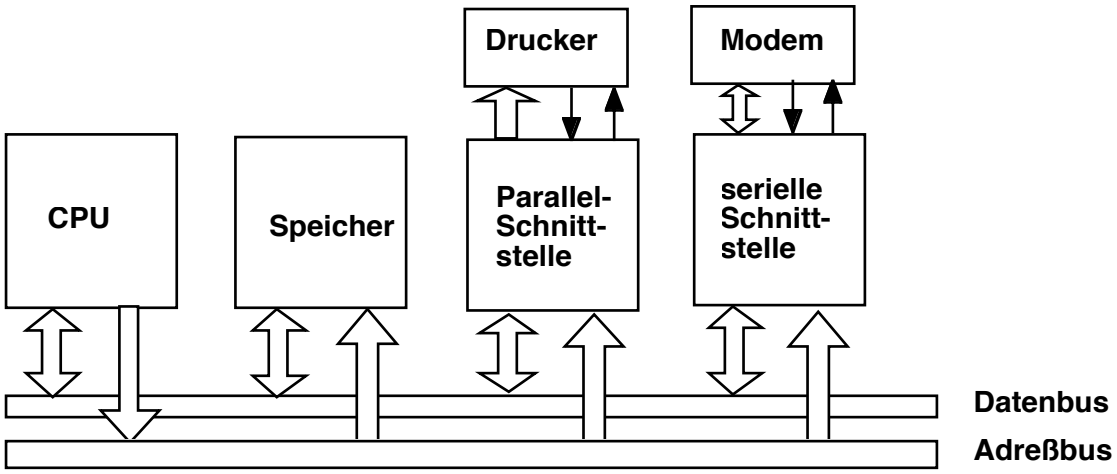


Abb. 9.1 Der Prozessor-Speicher-Bus

Prozessor, Bus und Speicher sind in ihrer Geschwindigkeit meist gut aufeinander abgestimmt. In unseren einfachen CPUs wird die Zykluszeit des Speichers optimalerweise so gewählt, daß man bei jedem Speicherzugriff des Prozessors sicher sein kann, daß der Speicher auch zur Verfügung steht. Können diese zeitlichen Eigenschaften garantiert werden, kann man ein

einfaches Protokoll für die Zusammenarbeit zwischen Prozessor und Speicher entwickeln. Ein ***Protokoll*** ist eine Festlegung der Regeln, mit der Information zwischen mehreren Komponenten ausgetauscht wird. Abb. 9.2 zeigt das Protokoll zum Lesen und Schreiben eines Bytes für den 6809. Takt E wird als Systemtakt bezeichnet und bildet die Basis für die zeitliche Steuerung des Protokolls. Q ist ein um eine halbe Perioden von E verzögertes Taktsignal. Es wird benötigt, um eine feinere Unterteilung des Gesamtzyklus T_{CYC} zu ermöglichen. Insgesamt kann man damit 4 definierte Zeitpunkte unterscheiden:

1. Takt E : negative Flanke - Start des Zyklus
2. Takt Q: steigende Flanke
3. Takt E: steigende Flanke
4. Takt Q: fallende Flanke

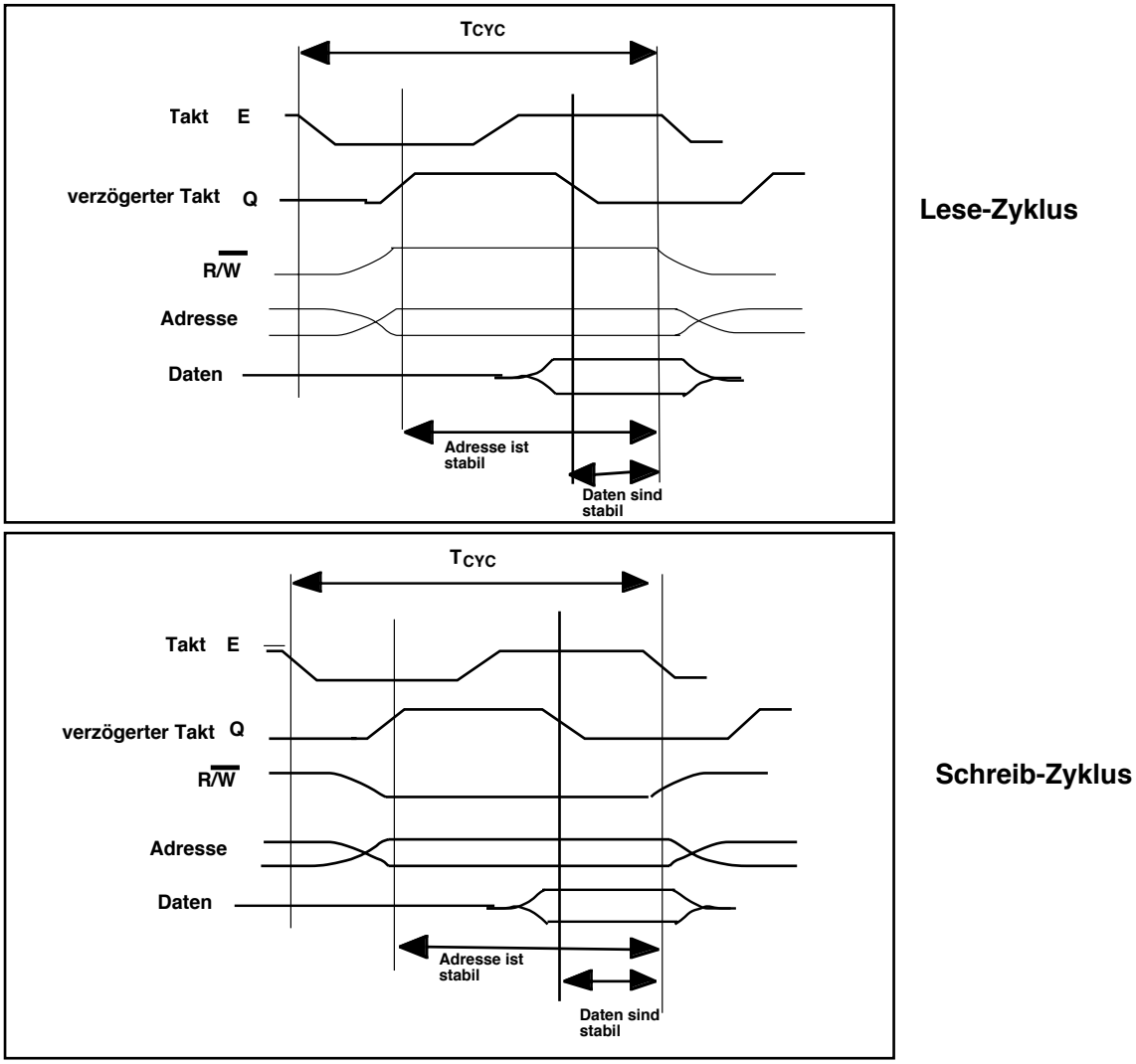


Abb. 9.2 Lese- und Schreibzyklus des Speicherbusses für den 6809

Mit der fallenden Flanke von E wird der Speicherzyklus initiiert und die Adressen auf den Adreßbus gelegt. Diese müssen zum Zeitpunkt 2 stabil sein, um den Datenzugriff im Speicher vorzubereiten. Ab dem Zeitpunkt 3 liegen die Daten stabil auf dem Datenbus und können vom Prozessor übernommen werden. Dieser muß bis zum Start des neuen Buszyklus die Übernahme abgeschlossen haben. Der Schreibzyklus wird analog in umgekehrter Transferrichtung durchgeführt. Das Signal R/W‘ bestimmt, ob es sich um einen Lese- oder einen Schreibzugriff handelt.

Beide Protokolle unterliegen einem festen Zeitraster, das durch den Takt E definiert ist. Der Speicher bzw. die technische Realisierung des Speichers muß so gewählt werden, daß gilt: (Zykluszeit des Speichers) ≤ (Periodendauer des Taktes E). Die Einhaltung der Zeitbedingung von Takt E allein ist entscheidend für die korrekte Funktion des Protokolls. Falls diese Bedingung garantiert werden kann, ist keine weitere Abstimmung zwischen Prozessor und Speicher über die Verfügbarkeit der Information im Lese - bzw. Schreibzyklus notwendig. Diese Form eines Protokolls, daß nur die Zeit für die Koordination des Informationstransfers benötigt, nennt man ein ***synchrones Protokoll***.

Wenden wir uns nun dem Anschluß eines PGs an diesen Bus zu. Nehmen wir an, das PG sei ein Drucker. Es ist klar, daß ein Drucker nicht mit der Geschwindigkeit Zeichen drucken kann, in der ein Prozessor in der Lage ist, Zeichen zu übermitteln. Dies führt zur ersten Schwierigkeit bei der EA: PGs sind meist wesentlich langsamer als ein Speicher, so daß sowohl Lesen als auch Schreiben nicht mit der durch den Systemtakt bestimmten Geschwindigkeit erfolgen kann. Eine völlig unbrauchbare Lösung wäre, den Systemtakt und damit die Datenrate des Busses entsprechend zu verlangsamen, da damit auch der Zugriff zum Speicher um Größenordnungen langsamer würde. Eine zweite Möglichkeit ist die Entkopplung des PGs vom Bus, indem man eine geeignete Hardwarekomponente vorsieht, die sich zum Bus hin wie eine Speicherkomponente verhält und auch das synchrone Busprotokoll erfüllt. Gegenüber dem PG dagegen erlaubt die Komponente eine zeitliche Steuerung, die auf die Möglichkeiten des PGs zugeschnitten ist. Eine solche Komponente wird als ***Steuereinheit für periphere Geräte*** oder kurz als PIA (***Peripheral Interface Adapter***) bezeichnet. Eine solche Einheit muß zwei Probleme lösen:

1. Ein PIA muß selektiert werden.
2. Die Daten zum PG müssen über den PIA mit einer adäquaten Datenrate übetragen werden.

9.1 Selektion der peripheren Geräte

Ein PG bzw. sein PIA wird über die Leitungen des Adreßbusses selektiert. Bereits bei unserem Modellrechner hatten wir den Adreßbus (oder Teile davon) als Auswahlleitungen für PGs

(Device Select Lines, DSL) nach außen geführt. Allerdings sind zwei Punkte zu berücksichtigen:

1. Falls ein PG selektiert wird, darf nicht gleichzeitig der Speicher selektiert werden.
2. Im Vergleich zur Größe des Adreßraum benötigen PGs nur wenige Speicherplätze.

Der erste Punkt besagt, daß wir Maßnahmen treffen müssen, die verhindern, daß bestimmte Adressen als Speicheradresse **und** als Adresse für ein PIA interpretiert werden. Dafür gibt es mehrere Lösungen:

• Spezielle EA-Befehle, die den Speicher deaktivieren.
• Die EA-Geräte liegen in einem Adreßraum, für den physikalisch kein Speicher vorhanden ist.
• Logik, die bei Erkennung einer PIA-Adresse den Speicher deaktiviert.

Die erste Lösung der speziellen EA-Befehle spannt einen vom Speicheradreßraum getrennten Adreßraum nur für PGs auf. Sie ist in Abb. 9.3 skizziert.

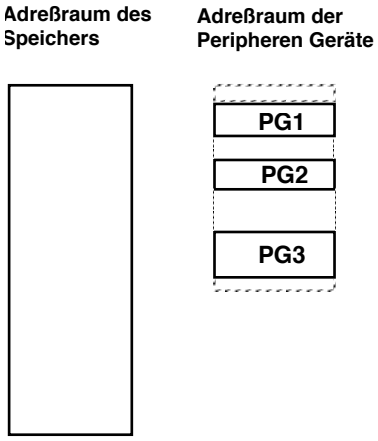


Abb. 9.3 Getrennter Adreßraum durch spezielle EA-Befehle

Wenn der Prozessor einen EA-Befehl ausführt, werden in der Datenphase spezielle EA-Leitungen aktiviert und gleichzeitig die Selektionsleitungen für den Speicher deaktiviert. Der gesamte Adreßraum, d.h. alle Leitungen des Adreßbusses stehen nun zur Adressierung externer Geräte zur Verfügung. Das Problem der gleichzeitigen Aktivierung von Speicher und PGs ist damit durch diesen Ansatz gelöst. Darüber hinaus wird durch die Verfügbarkeit des gesamten Adreßraums und die zusätzliche Bereitstellung von Signalleitungen zur Anzeige eines EA-Befehls der Anschluß von PGs vereinfacht.

Das Prinzip der EA ohne spezielle Befehle ist in Abb 9.4 dargestellt. Hier wird der Adreßraum von Speicher und PGs gemeinsam genutzt. Die PGs werden in den normalen Adreßraum abgebildet. Deshalb nennt man diesen Ansatz auch "Memory mapped I/O" (MMIO).

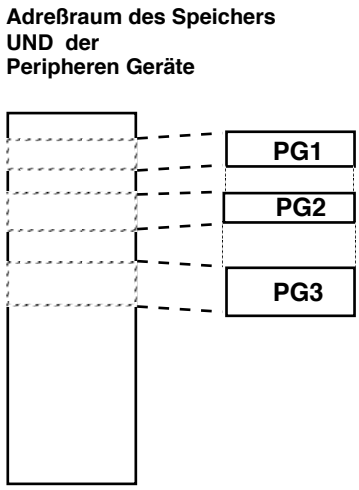


Abb. 9.4 Memory Mapped I/O

Bei MMIO muß man das oben angesprochene Problem der Synchronisation zwischen Speicher und PIAs lösen. Wird der Teil des Adreßraums adressiert, in den ein PIA abgebildet ist, muß der Speicher deaktiviert werden. Der Adreßraum, der für den PIA benötigt wird, ist klein und belegt typischerweise nur wenige (oft nur 4 - 16) Adressen.

Wir benötigen einen Adreßdekoder, der genau dann ein Aktivierungssignal erzeugt, wenn eine Adresse im Adreßraum des PIAs liegt. Abb. 9.5 zeigt einen typischen Adreßdekodierer, der genau dann ein Aktivierungssignal liefert, wenn auf dem 16-Bit Adreßbus die Adresse:
0011 1111 1111 11xx erscheint. Der PIA belegt dann die Adressen: 3FFC, 3FFD, 3FFE, 3FFF. Gewöhnlich ist der Adreßraum der CPU nicht vollständig durch (physischen) Speicher belegt, so daß man den Adreßraum der PIAs in diesen freien Adreßraum legt[1].

Das CS-Signal (Chip Select) aktiviert den PIA, die Adressen A_0 und A_1 werden zur Auswahl interner Register des PIAs genutzt (RS: Register Select). Dadurch lassen sich vier Register anwählen, die Daten und Status des PIAs enthalten können. Die Zuordnung von Adressen und Registern (Memory Map) ist im rechten Teil der Abb.9.5 angegeben.

[1] Im ungewöhnlichen Fall, daß PIA und Speicher auf denselben Adressen liegen, muß der Speicher mit dem CS-Signal des PIA deaktiviert werden.

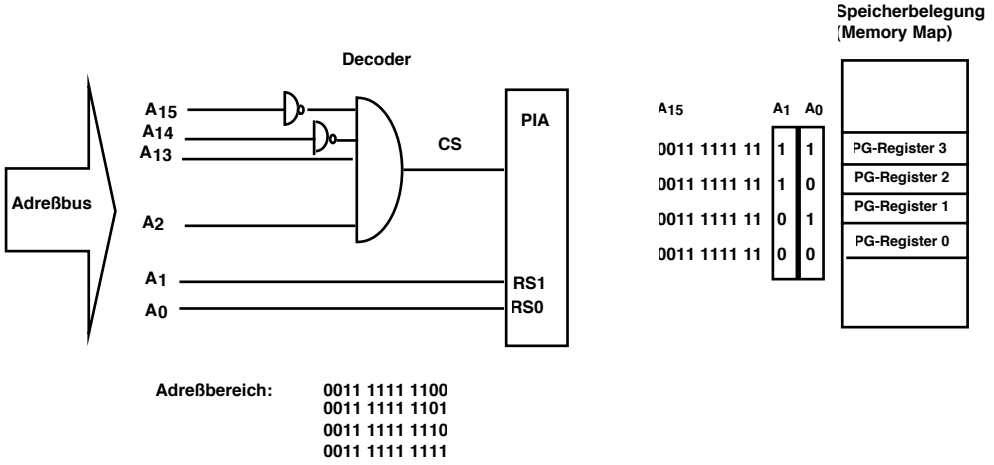


Abb. 9.5 Dekodierung einer Speicheradresse zur Selektion eines PIAs

9.2 Übertragung von Daten vom Prozessor zum peripheren Gerät

Wir wollen nun einen einfachen PIA entwerfen. Betrachten wir zunächst die Ausgabe, z.B. einen Drucker. Es sollen nacheinander Zeichen vom Prozessor zum Drucker übertragen werden.

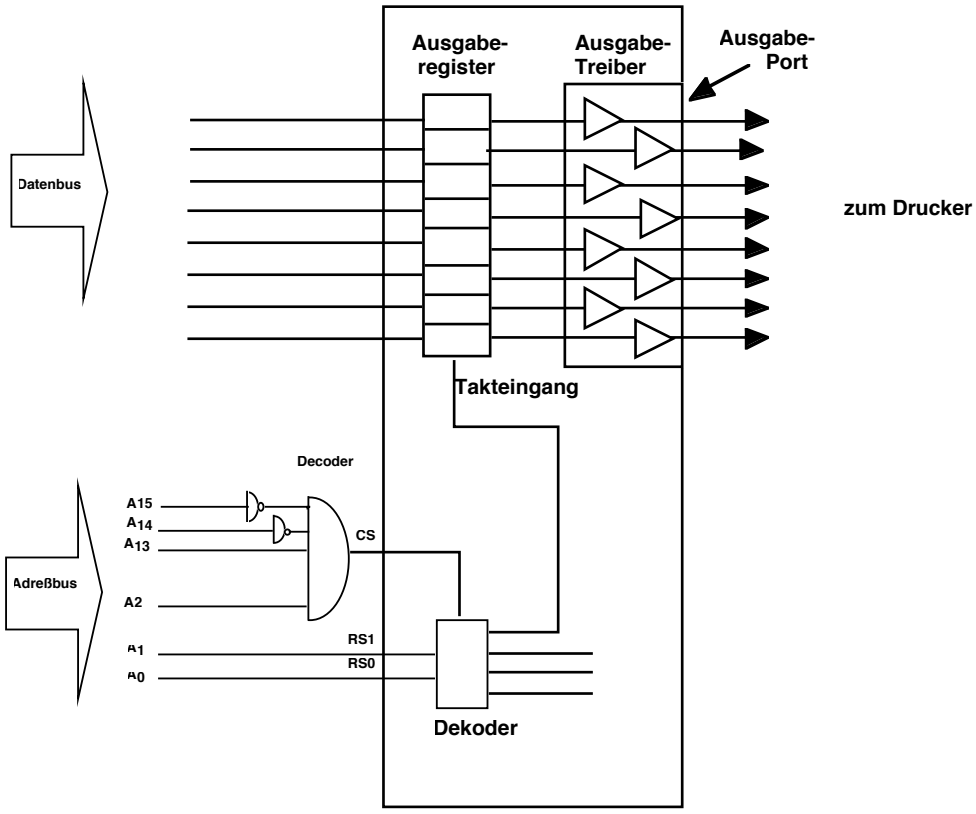


Abb. 9.5 Ein adressierbares Register als Ausgabepuffer

Da der Prozessor mit sehr viel höherer Geschwindigkeit arbeitet als der Drucker, ist es naheliegend, daß ein Zeichen vom Prozessor in einen Puffer geschrieben wird, aus dem der Drucker es entnehmen kann. Kombinieren wir die Selektionslogik aus dem vorigen Abschnitt mit einem Ausgaberegister, erhalten wir eine Komponente wie in Abb. 9.6 gezeigt. Es ist im wesentlichen ein adressierbares Register, in das der Prozessor schreiben kann. Der Drucker ist damit zeitlich entkoppelt . Die Ausgabeleitungen als Schnittstelle zum PG bezeichnet man auch als ***Ausgabe-Port*** des PIA. Das Ausgaberegister wird über die Adresse 3FFF selektiert. In einem Schreibzyklus kann der Prozessor das Register unter dieser Adresse beschreiben wie eine Speicherzelle.

Wenden wir uns nun der Eingabe zu und betrachten als Beispiel den Anschluß einer Tastatur.

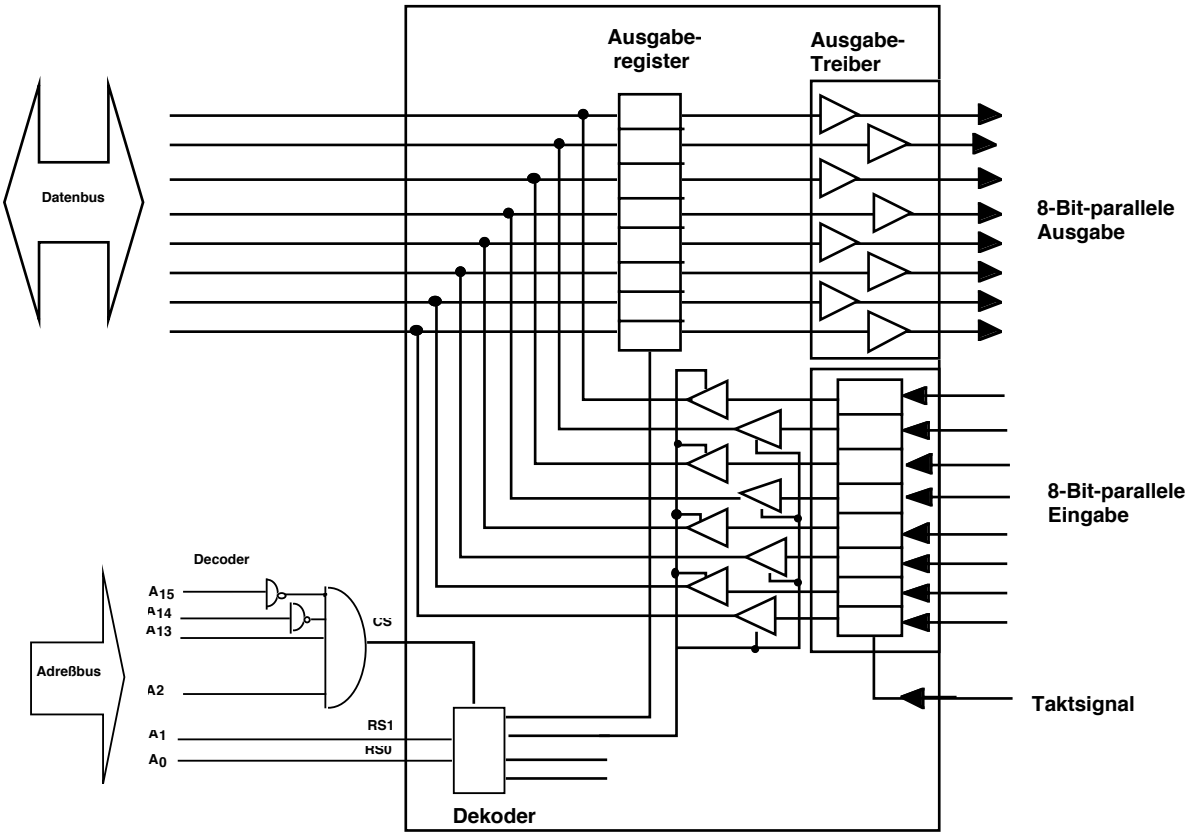


Abb. 9.6 Erweiterung um einen parallelen Eingabe-Port

Wir machen dabei folgende (realistischen) Annahmen:

- Ein Zeichen liegt im 7-Bit ASCII-Code vor.
- Die Übertragung dieses Zeichens erfolgt über acht parallele Leitungen.
- Bei jedem Tastendruck soll das der Taste entsprechende Zeichen übertragen werden.

• Bei jedem Tastendruck wird zusätzlich ein Signal TS erzeugt , das signalisiert, daß eine Taste gedrückt wurde.

Auf der Seite des PIA sehen wir ein Eingangsregister vor. Das Signal TS nutzen wir als Takt, um das ASCII-Zeichen im Eingangsregister zu speichern. Abb. 9.7. zeigt unseren so erweiterten PIA. Wir haben einen sogenannten ***Eingabe-Port*** hinzugefügt. Das Eingaberegister kann vom Prozessor über die Adresse 3FFE ausgelesen werden. Dabei werden die Tri-State-Puffer durch das Selektionssignal aktiviert und der Inhalt des Eingaberegisters wird auf den Bus gelegt.

Soweit haben wir gesehen, wie die Eingabe und Ausgabe durch Register gepuffert werden kann, so daß ein externes Gerät nicht den zeitlichen Bedingungen des Prozessor/Speicher-Busses unterworfen ist. Allerdings müssen wir trotzdem noch die Probleme der Synchronisation zwischen Prozessor und peripherem Gerät lösen. Nehmen wir an, der Prozessor hat ein Zeichen zur Ausgabe auf dem Drucker in das Ausgaberegister geschrieben. Bevor der Drucker dieses Zeichen nicht gedruckt hat, darf der Prozessor kein neues Zeichen in das Ausgaberegister schreiben, da das alte Zeichen sonst vom neuen überschrieben würde und damit verlorenginge. Auch auf der Eingabeseite besteht das Problem. Ohne zusätzliche Information kann das PG nicht entscheiden, ob das Zeichen im Puffer bereits ausgelesen wurde. Der Prozessor hingegen kann nicht wissen, ob im Puffer ein neues oder bereits gelesenes Zeichen liegt.

Eine zeitliche Steuerung ist an dieser Stelle schwierig, da die Zeitbedingungen, die z.B. für die Tastatureingabe gelten, nicht festgelegt werden können. Deshalb muß ein ***ereignis-gesteuertes Protokoll*** entwickelt werden. Um die Wirkungsweise eines ereignisgesteuerten Protokolls zu verdeutlichen, wollen wir das Beispiel der Koordination startender Maschinen auf einem Flughafen betrachten. Die Aufeinanderfolge der Schritte vom ersten Rollen bis zum Start wird durch eine Folge von Statusmeldungen, Anforderungen und Freigaben zwischen dem Piloten der Maschine und dem überwachenden Tower geregelt.

Ereignis/ Status	Pilot	Tower
Maschine rollbereit	LH 695, wir sind bereit zum Rollen und erbitten Rollerlaubnis	
		LH 695, verstanden, erteilen Rollerlaubnis für Runway W23
	LH 695, verstanden	
Maschine ist am Ende des Runways	LH 695 sind startbereit und erbitten Starterlaubnis	
Startbahn ist belegt		LH 695, verstanden, warten Sie bitte
	LH 695, verstanden	
•		
•		
Startbahn ist frei		LH 695 wir erteilen Starterlaubnis auf NW 4
	LH 695 verstanden, Starten auf Bahn NW 4	

Wir können folgende Eigenschaften des Protokolls beobachten:

- Ein Schritt des Protokolls ist die Folge eines Ereignisses, das durch einen bestimmten Status repräsentiert wird, z.B. der Status rollbereit ist die Folge davon, daß alle Passagiere Platz genommen haben, der Pilot einen bestimmten Teil seiner Checkliste erfolgreich abgearbeitet hat, die Türen geschlossen sind usw.
- Jeder Schritt basiert auf dem erfolgreichen Abschluß des vorhergehenden Schrittes. Bevor nicht der erfolgreiche Abschluß einer Aktivität gemeldet wurde, wird keine weitere Aktion unternommen.
- Das Protokoll wird durch explizite Anforderungen und explizite Bestätigung gesteuert.

Betrachten wir nun einen Datentransfer zwischen einem Prozessor und einem PG, der nach diesen Prinzipien gesteuert wird. Das Flußdiagramm des Protokollablaufs ist in Abb. 9.8 angegeben. Sind die Daten, die übertragen werden sollen, verfügbar, d.h. sie wurden vom Prozessor in das Ausgangsregister des PIA geschrieben, wird das Signal "Data available" aktiviert. Dieses Signal fordert die Abnahme der Daten durch das PG an. Hat das PG die Daten übernommen, aktiviert es seinerseits das "Data acknowledge" Signal. Damit bestätigt es den Empfang der Daten. Bevor das Signal "Data acknowledge" nicht empfangen wurde, werden keine neuen Daten in das Ausgangsregister geschrieben. Umgekehrt, bevor nicht das Signal

"Date available" erzeugt wird, "weiß" das PG, daß keine neuen Daten verfügbar sind. Abb. 9.9 zeigt den zeitlichen Ablauf des Protokolls, wobei die Pfeile die kausale Abhängigkeit der Signale andeuten.

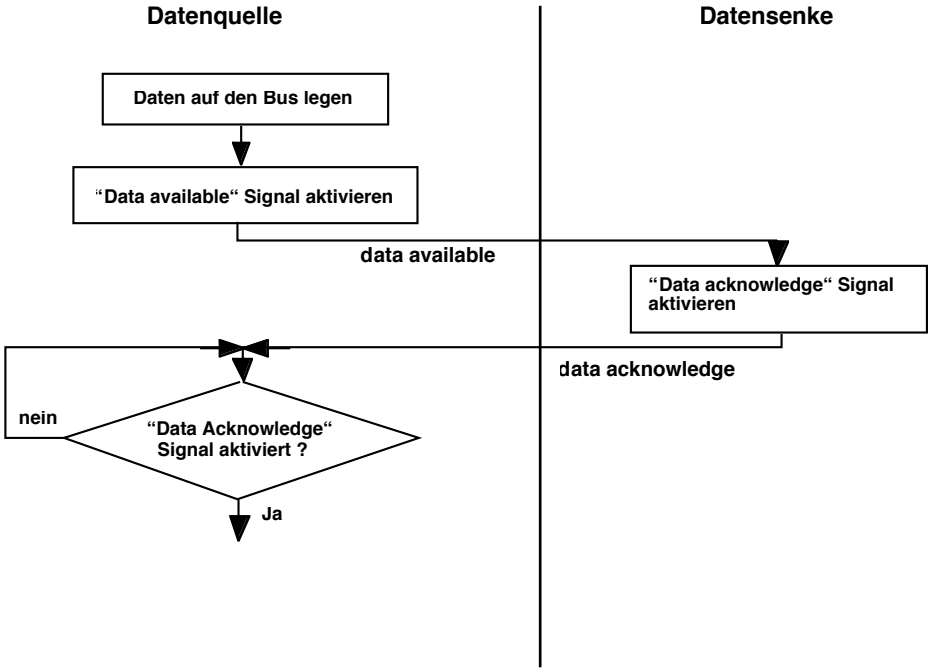


Abb. 9.8 Einfaches Handshake-Protokoll (Zeitdiagramm)

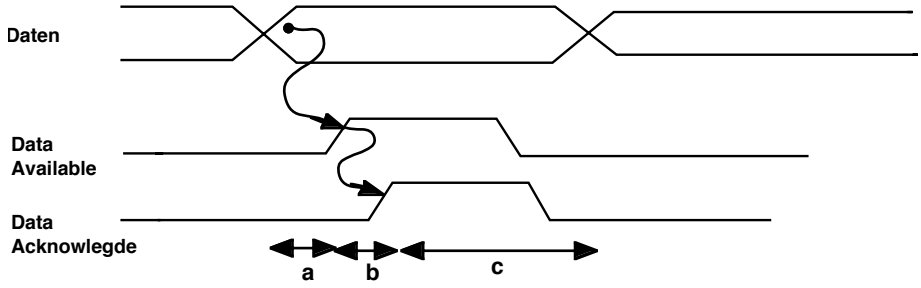


Abb. 9.9 Einfaches Handshake-Protokoll (Flußdiagramm)

Protokolle dieser Art werden auch als ***Handshake-Protokolle*** oder ***asynchrone Protokolle*** bezeichnet. Das bisherige einfache Handshake-Protokoll macht nur Aussagen darüber, wann die Steuersignale aktiviert werden, aber nicht wann sie wieder, abhängig voneinander, deaktiviert werden. Das entsprechende Protokoll wird als ***vollständig verschränktes (fully interlocked) Handshake-Protokoll*** bezeichnet und ist in seinem Ablauf in den Abb. 9.10 und 9.11 dargestellt.

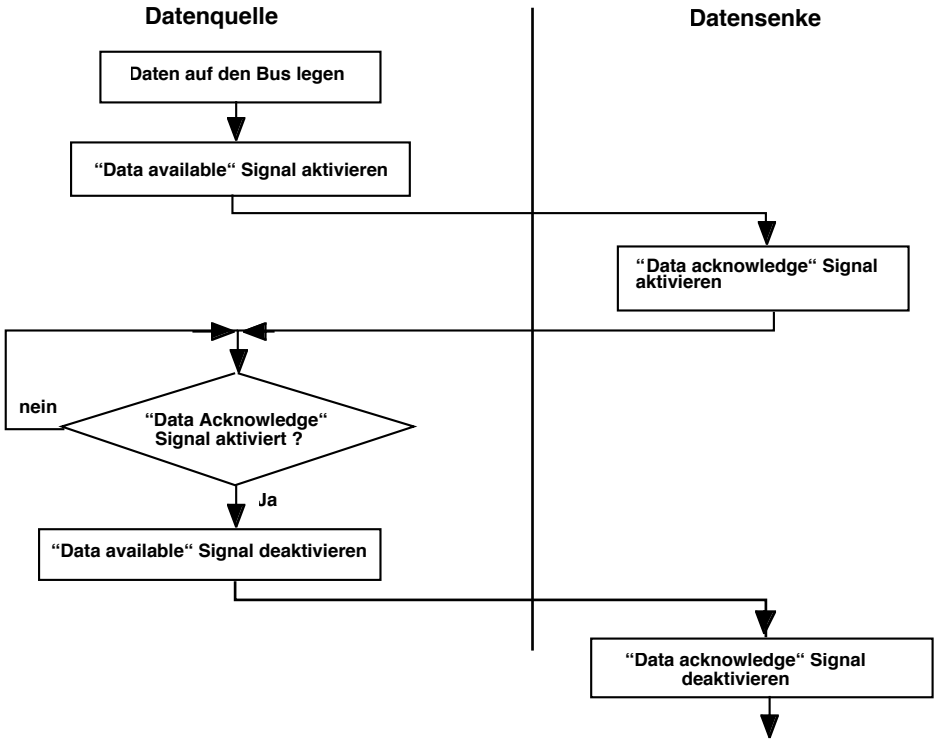


Abb. 9.10 Vollständig verschränktes Handshake-Protokoll (Flußdiagramm)

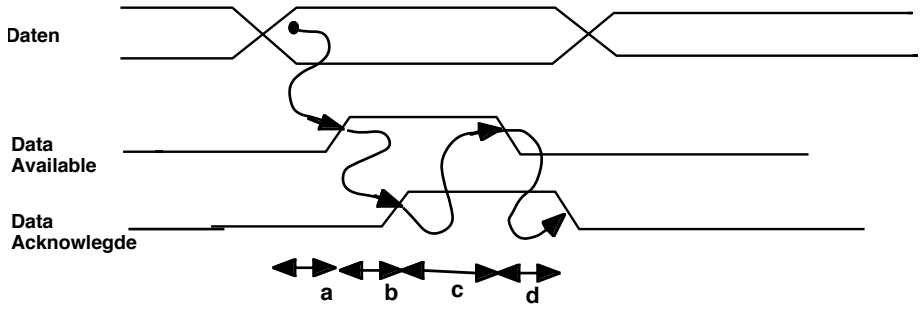


Abb. 9.11 Vollständig verschränktes Handshake-Protokoll (Zeitdiagramm)

Um ein Handshake-Protokoll zu realisieren, müssen wir einen Anforderungs- und Bestätigungsmechanismus entwerfen. Zu diesem Zweck führen wir auf der Eingabe- und Ausgabeseite je ein Status-Flag ein. Abb. 9.12 zeigt den erweiterten PIA. Über SF_{OUT} wird die Ausgabe, über SF_{IN} die Eingabe synchronisiert. Am Beispiel des Druckers soll die Funktionsweise erläutert werden.

Wenn die CPU ein Zeichen für den Drucker in das Ausgaberegister schreibt, setzt sie gleichzeitig das SF_{OUT} auf 0. (vgl. 9.12: der R-Eingang des SF_{OUT} ist mit dem Takteingang

des Ausgaberegisters verbunden.). Die Kontrolleitung CH $_{OUT}$ signalisiert dem Drucker nun die Verfügbarkeit eines neuen Zeichen im Ausgaberegister. Hat der Drucker das Zeichen gelesen, setzt er das SF_{OUT} über die Kontrolleitung CH $_{IN}$ auf 1. Die CPU kann den Status des SF_{OUT} lesen und somit feststellen, daß der Drucker das Zeichen abgeholt hat. Abb. 9.13 zeigt die Verbindung zwischen Drucker und PIA mit dem entsprechenden Protokoll für den Datentransfer. Diese 8-Bit parallele Schnittstelle ist als "Centronics-Schnittstelle" genormt.

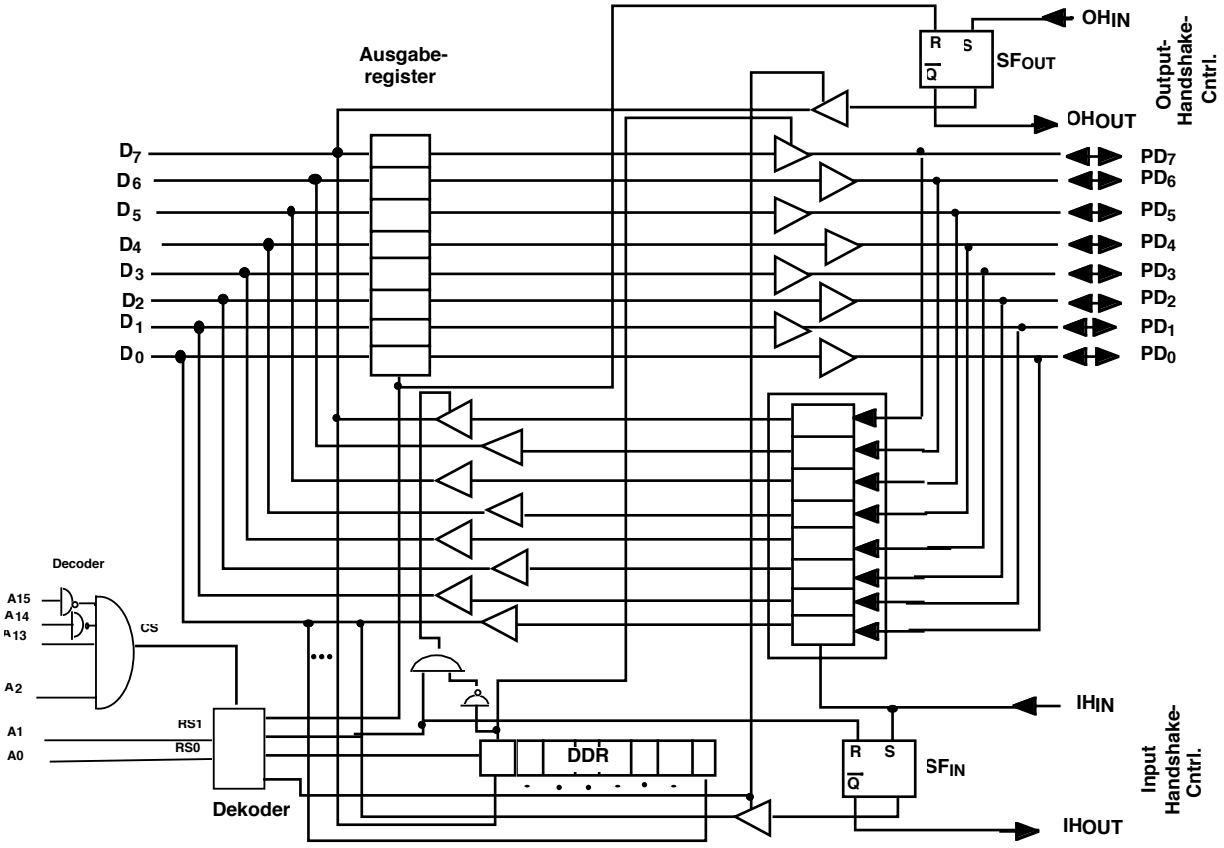


Abb. 9.12 Vollständiger PIA

Der Drucker hat zwei sogenannte Handshake Kontrolleitungen, eine Ausgangsleitung RDY‘, um seine Bereitschaft Daten zu übernehmen anzuzeigen, und eine Eingangsleitung STROBE‘, mit der die Daten in sein internes Latch übernommen werden.

Kann der Drucker ein neues Zeichen übernehmen, aktiviert er seine RDY‘-Leitung (RDY‘ ← 0) und setzt damit SF_{OUT} auf 1. Die CPU erkennt durch explizites Abfragen des Status-FF, daß ein weiteres Zeichen übernommen werden kann. Wie aus dem Zeitdiagramm ersichtlich ist, werden daraufhin Daten in das Ausgangsregister geschrieben und damit die Strobe-Kontrolleitung aktiviert (SF_{OUT} ← 0). Die Daten können nun in den Drucker übernommen werden. T_{min} bezeichnet die minimale Zeit, in welcher die Daten nach der Aktivierung von

STROBE stabil anliegen müssen. Nach der Aktivierung des STROBE-Signals wird die Deaktivierung des RDY'-Signal dazu benutzt, SF_{OUT} wieder auf 0 zurückzusetzen. Das explizite Abfragen des Status erfolgt meist zyklisch in einer Programmschleife und wird als ***Polling*** bezeichnet[2]. Insbesondere, wenn mehrere peripheren Geräte angeschlossen sind, ist Polling eine Methode, um festzustellen, von welchem Gerät eine Anforderung an die CPU vorliegt.

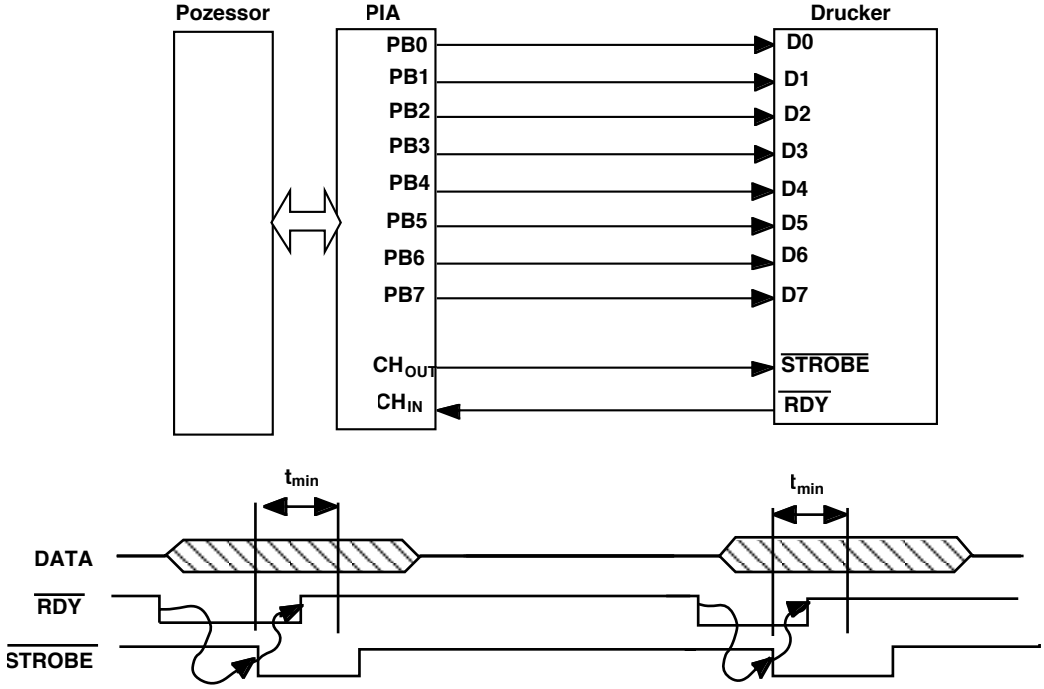


Abb. 9.13 Steuerung einer Centronics Druckerschnittstelle

Zur Zeichenübertragung zum Drucker werden alle 8 Leitungen der E/A-Ports als Ausgangsleitungen benötigt. Im allgemeinen sind jedoch Ein- und Ausgangsleitungen erforderlich.

Nehmen wir als Beispiel eine Windrichtungsanzeige, die 16 verschiedene Himmelsrichtungen unterscheidet. Eine Nockenwelle schließt dabei je nach der Windrichtung einen Schalter. Eine schematische Skizze des Sensors für die Windrichtung ist in Abb. 9.14 gegeben. Wollten wir nun jeden Schalter einzeln über die PIA abfragen, benötigten wir 16 Eingänge, die bei unserer PIA nicht zur Verfügung stehen. Eine Standardmethode, um mit weniger individuellen Eingängen auszukommen, ist die Anordnung der Schalter in einer n x m Matrix. Nacheinander werden die n Zeilen aktiviert und parallel die Belegung der m Spalten abgefragt. Die Schalteranordnung für unser Beispiel ist in Abb. 9.15 angegeben.

[2] vgl. auch Script: Speicher und periphere Geräte.

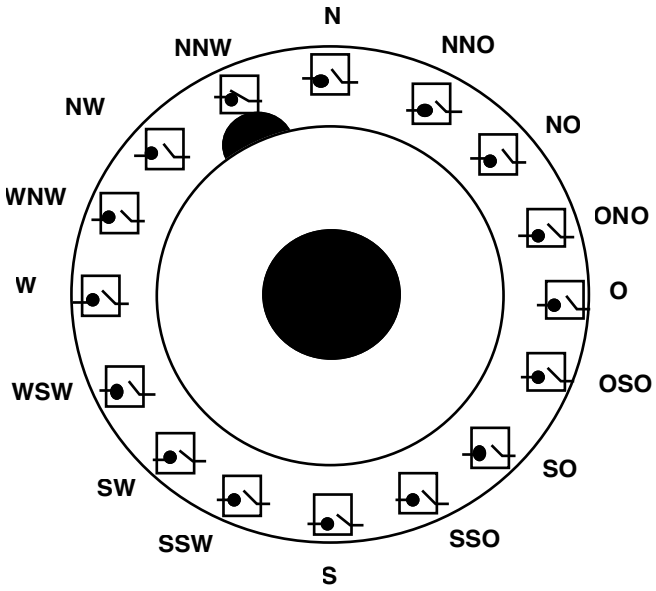


Abb. 9.14 Eine Windrichtungsanzeige

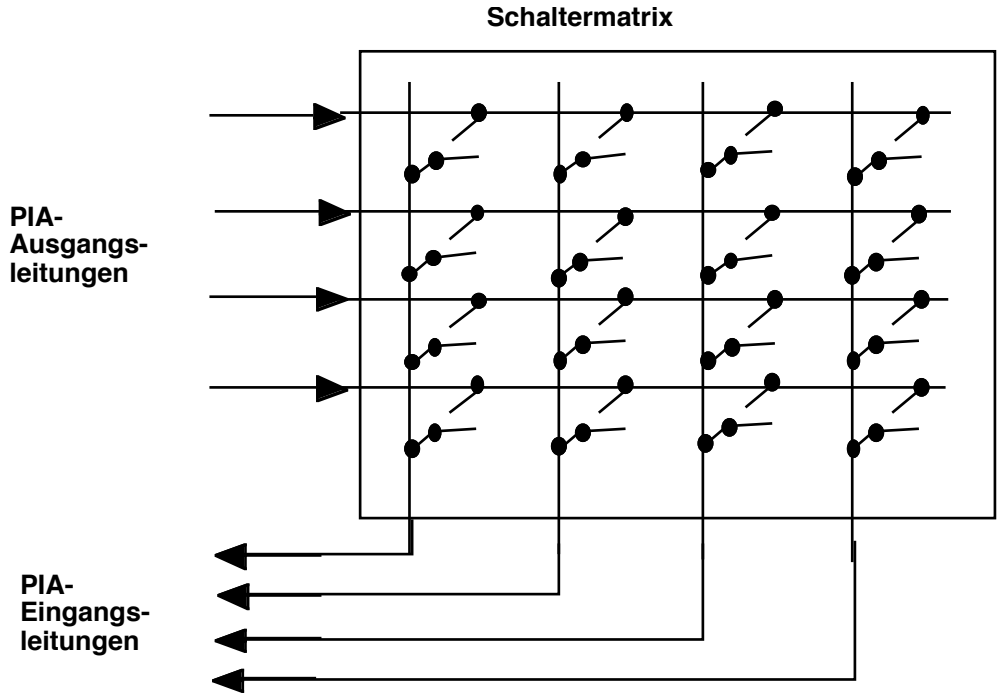


Abb. 9.15 Anordnung der Schalter in einer Matrix

Wir haben die 16 Schalter in einer 4 x 4 Matrix angeordnet[3]. Wir benötigen 4 Ausgangsleitungen, um die einzelnen Zeilen der Matrix nacheinander zu aktivieren. Über 4

[3] Die Anordnung der Schalter in einer quadratischen Matrix ist bezüglich der benötigten Leitungen zur Abfrage der Matrix (Anzahl der Zeilen und Spalten) optimal.

Eingangsleitungen kann dann abgefragt werden, welcher Schalter geschlossen ist. Grundsätzlich können wir auch feststellen, wenn mehrere Schalter geschlossen sind, was in unserem Beispiel eine noch feinere Unterteilung der Windrichtungen ermöglicht.

Unser PIA muß also die Forderung erfüllen, die Leitungen eines Ports sowohl als Ausgangsleitungen wie auch als Eingangsleitungen je nach Anwendung definieren zu können. Dies wird durch die Einführung eines Daten-Richtungs-Registers (Data Direction Register, DDR) realisiert. In Abb. 9.12 ist das DDR bereits gezeigt. Im Detail ist der Übersichtlichkeit wegen nur für ein Bit angegeben, wie das DDR die Konfiguration der Port-Leitungen übernimmt.

Das DDR ist ein Write-Only-Register, d.h. es kann von der CPU unter der Adresse 3FFC lediglich beschrieben werden. Jedes Bit des DDR repräsentiert die Konfiguration einer Port-Leitung folgendermaßen:
(DDR-Bit n) =1 : Leitung n ist als Ausgangsleitung konfiguriert.
(DDR-Bit n) = 0: Leitung n ist als Eingangsleitung konfiguriert.

Über die nachgeschaltete Logik wird bei einer 0 der entsprechende Ausgangstreiber gesperrt und der Eingangstreiber aktiviert, falls zusätzlich das Eingangsregister von der CPU selektiert wird. Bei einer 1 ist es umgekehrt mit dem einzigen Unterschied, daß die Ausgangstreiber unabhängig von der Selektion der CPU immer aktiviert sind, d.h. die Daten im Ausgaberegister ständig auf den Ausgangsleitungen anliegen.

Das Assemblerfragment:

```
PIABSE  EQU   xxxx          Festlegen der PIA-Basisadresse
DDR     EQU   PIABSE+3      DDR wird mit Distanz 3 adressiert (Abb. 9.13)
EAPAT   EQU   %1111 0000    Muster zur Konfiguration des DDR

        LDA   #EAPAT
        STA   DDR
```

konfiguriert die Leitungen 4, ... , 7 des PIA als Ausgänge, die Leitungen 0, ..., 3 als Eingänge, so wie wir es für unsere Schaltermatrix benötigen. Anstelle des Windrichtungs-anzeigers könnten wir mit demselben Prinzip z.B. eine hexadezimale Tatstatur oder einen Joystick mit einer relativ feinen Auflösung realisieren.

Der in diesem Kapitel entwickelte und in Abb. 9.12 dargestellte PIA weist die grundlegenden Funktionen einer solchen Einheit auf. Eine sehr ähnliche Einheit ist der 6820/21 von Motorola, der allerdings 2 getrennte Ein/Ausgabe-Ports hat und über wesentlich mehr

Konfigurationsmöglichkeiten verfügt, wie z.B die softwareseitige Einstellung, ob die Kontrollsignale durch positive oder negative Flanken aktiviert werden, verschiedene Alternativen des Handshake-Protokolls und die Möglichkeit, über die Kontrolleitungen Programmunterbrechungen auszulösen, um auf zur Programmausführung asynchrone Ereignisse, wie sie typisch für die Ein- und Ausgabe sind, zu reagieren. Diesem Thema werden wir uns im nächsten Kapitel zuwenden.

9.3 Serielle Eingabe und Ausgabe

Die bisher betrachtete parallele Ein- und Ausgabe ist besonders geeignet, wenn keine größeren Leitungslängen benötigt werden. Für größere Entfernungen sind parallele Leitungen relativ teuer. Hier wird eine Bit-serielle Übertragung der Daten zwischen den peripheren Geräten und der CPU, bzw. dem E/A-Controller bevorzugt. Da die Übertragung des Bitstroms zwischen dem E/A-Controller und dem peripheren Gerät asynchron erfolgt, wird der entsprechende E/A-Controller oft als ACIA (Asynchronous Communication Interface Adapter) (Abb. 9.16) bezeichnet.

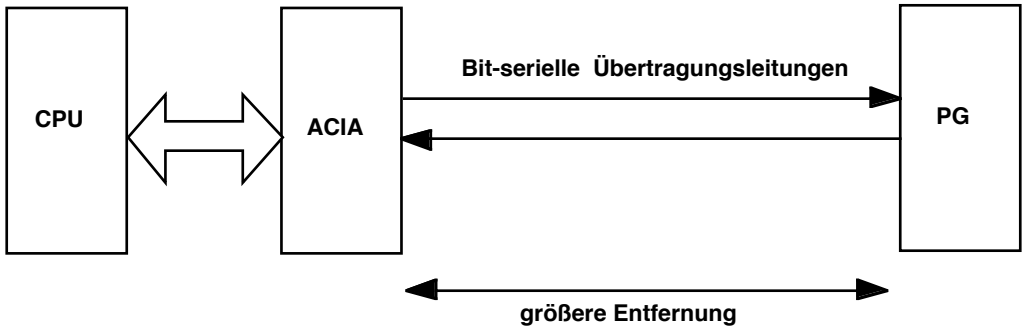


Abb. 9.16 Bit-serielle Datenübertragung

Die grundsätzlichen Methoden der Synchronisation zwischen der CPU und dem ACIA können wir von unserem PIA übernehmen. Wir müssen dann zunächst die Wandlung der parallelen Daten von der CPU in einen seriellen Bitstrom betrachten. Dazu ersetzen wir das Eingabe-Register und das Ausgabe-Register der PIA durch Schieberegister, die wir von der Seite der CPU parallel lesen, bzw. parallel beschreiben können. Auch auf der Seite des peripheren Geräts nehmen wir einen Serien/Parallel Wandler an. Abb. 9.17 zeigt die grundlegende Struktur. Beim Senden wird ein Zeichen parallel in das Sender-Shift-Register geladen und die serielle Übertragung initiiert. Der Inhalt des Sender-Shift-Register wird nun mit dem Sendetakt seriell über den Ausgang TxD ausgelesen.

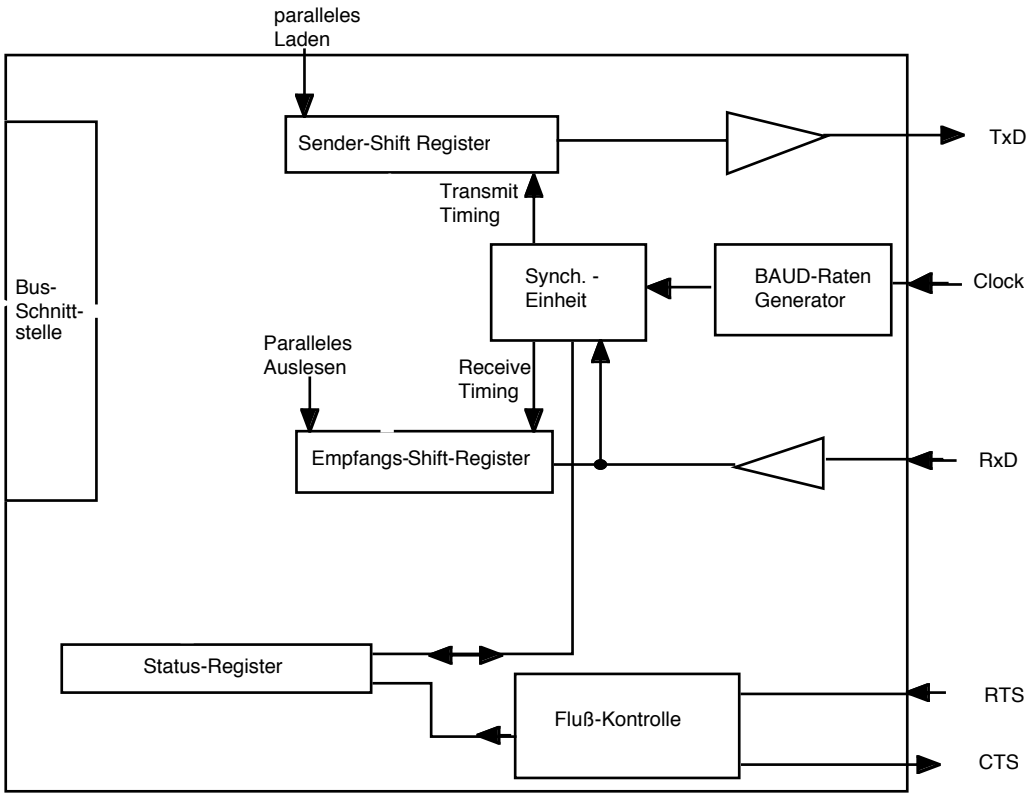


Abb. 9.17 Wesentliche Komponenten eines ACIA

Das Übertragungsformat ist in Abb. 9.18 angegeben. Eine "1" wird durch einen positiven Spannungspegel, eine "0" durch einen negativen Spannungspegel repräsentiert. Sender und Empfänger müssen dieselbe Übertragungsrate annehmen, die durch einen vereinbarten Sende- und Empfangstakt festgelegt ist. Damit der Gleichtakt im Sender und im Empfänger gewährleistet ist, wird bei jeder Übertragung eines Zeichens synchronisiert. Dazu werden den im Sender-Shift-Register gespeicherten Informationsbits noch zusätzliche Synchronisationsbits hinzugefügt. Werden keine Daten übertragen, liegt die Sendeleitung auf "1". Die Synchronisation zwischen Sender und Empfänger erfolgt bei jeder Zeichenübertragung durch ein Startbit (eine "0"). Im Empfänger wird das Startbit durch eine spezielle Einheit, die Synchronisationseinheit erkannt. Die Synchronisationseinheit erzeugt nun den Takt für das Empfangsschieberegister. Bei jedem Takt wird der jeweilige logische Wert auf der Übertragungsleitung in das Empfangsschieberegister übernommen. Ist das Zeichen übertragen, kann optional ein Paritätsbit erzeugt und angehängt werden. Nach der Übertragung eines Zeichens muß die Übertragungsleitung für mindestens die Zeit eines Bits auf "1" liegen. Dies wird als Stopbit bezeichet. Danach kann das nächste Zeichen übertragen werden, was durch ein neues Startbit angezeigt wird.

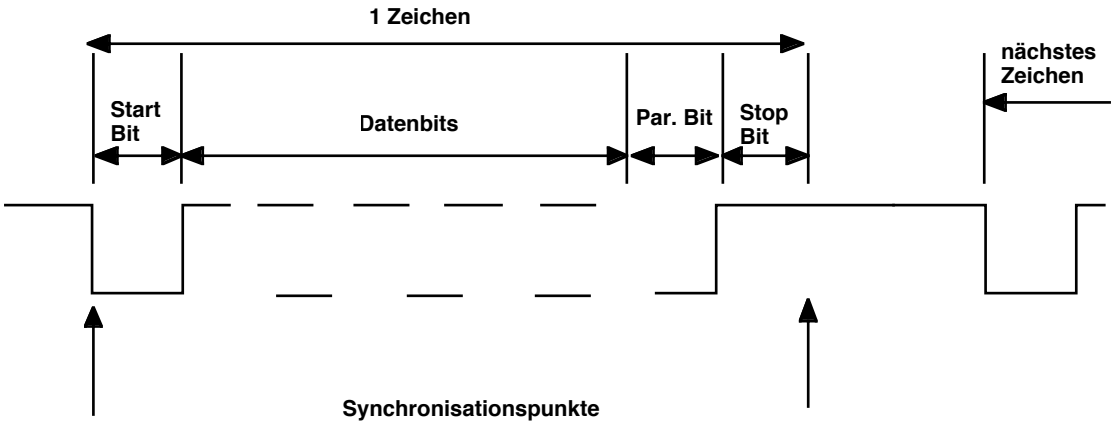


Abb. 9.18 Übertragungsformat eines Zeichens

Abb. 9.19 zeigt als Beispiel die Übertragung eines 7-Bit Zeichens ohne Paritätsbit. Hat die Synchronisationseinheit ein Startbit erkannt, erzeugt sie das Zeitintervall ΔT. Nach Ablauf von ΔT wird der Wert auf der Übertragungsleitung in das Schieberegister geschoben. Nun wird jeweils in der Mitte des Übertragungsintervalls für ein Bit Δt der logische Wert auf der Leitung abgefragt und im Schieberegister gespeichert. Dadurch wird gesichert, daß der Wert auf der Leitung stabil ist. Nach Ablauf von 7 Takten "weiß" die Synchronisationseinheit aufgrund des vereinbarten Übertragungsformats, daß die Übertragung beendet ist und erkennt das Stopbit. Sie geht daraufhin in einen Wartezustand bis sie erneut ein Startbit erkennt und ein neues Zeichen seriell eingelesen wird. Falls kein Stopbit gesendet wird, liegt ein Fehler vor.

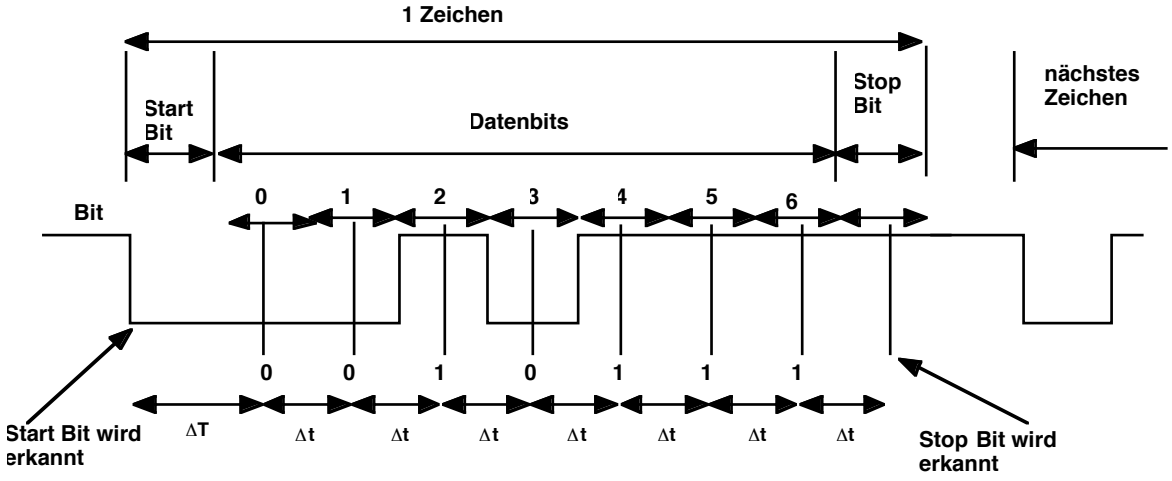


Abb. 9.19 Beisp. Übertragung des Zeichens "0010111_2"

Das Sender-Shift-Register wird parallel vom Prozessor geladen, das Empfangsregister wird parallel vom Prozessor ausgelesen. Um anzuzeigen, daß im Senderregister des ACIA ein Zeichen zur Übertragung bereitliegt, bzw. um zu signalisieren, daß der ACIA bereit ist, ein

neues Zeichen einzulesen, ist ein Handshake-Protokoll vorgesehen, die sogenannte ***Flußkontrolle***. Die Signale zur Flußkontrolle sind CTS (Clear To Send), das anzeigt, daß ein Zeichen im Sender-Shift-Register vorliegt und übertragen werden kann, und RTS (Request To Send), das die Bereitschaft zur Übernahme eines Zeichens auf der Empfängerseite signalisiert. Mit Hilfe der Flußkontrolle kann sichergestellt werden, daß kein Zeichen bei der Übertragung von Prozessor(register) zu Prozessor(register) verlorengeht.

10 Unterbrechungsbehandlung

Bisher wurde die Steuerung der Eingabe und Ausgabe ausschließlich von der CPU übernommen. Der PIA enthält dafür alle Status und Datenregister. Die Bereitschaft des PGs für einen Datentransfer wurde von der CPU durch explizites Abfragen des Status im PIA (Polling) festgestellt. Diese explizite Statusabfrage muß periodisch in ein Programm eingefügt werden. Dabei spielt die Häufigkeit, mit der ein externes Ereignis abgefragt werden muß, eine Rolle. Jede Abfrage erzeugt Zusatzaufwand, so daß man bestrebt ist, die Intervalle zwischen den Abfragen groß zu machen, damit der relative Aufwand sinkt. Damit wird aber auch die Verzögerung mit der das Auftreten eines externen Ereignisses behandelt werden kann entsprechend hoch. Es besteht also ein Zielkonflikt zwischen der Häufigkeit der Abfragen und dem damit verbundenen Aufwand auf der einen, und der Verzögerung, mit der auf ein externes Ereignis reagiert werden kann, auf der anderen Seite.

Eine Möglichkeit, dieses Problem zu umgehen, wäre, wenn ein externes Ereignis an die CPU signalisiert werden könnte, ohne daß dieses explizit abgefragt werden muß. Da ein solches Signal an die CPU den normalen Programmfluß unterbricht und eine zum Programm asynchrone Behandlung erfordert, wird es als Unterbrechung (Interrupt) bezeichnet. Die Steuerung der Ein- und Ausgabe wie wir sie bisher kennengelernt haben ist nur ein Anwendungsbereich. Weitere wichtige Aufgaben liegen vor allem auch in der Steuerung und Regelung externer (physikalischer) Prozesse. Hier gibt es zwei Klassen von Ereignissen, für die Interrupts den adäquaten Mechanismus darstellen:
1. Sehr häufige, periodische Ereignisse, wie z.B. eine Echtzeituhr, Zählungen bei Drehzahlmessungen usw. Hierbei darf kein Ereignis (Uhr-Tick, Zählimpuls) verlorengehen, so daß eine asynchrones Protokoll wie bei anderen Ein-Ausgabegeräten Probleme aufwirft bzw. die explizite Abfrage sehr genau in einem kleinen Zeitfenster erfolgen muß.
2. Sehr seltene Ereignisse, die sporadisch auftreten, deren Behandlung jedoch keine langen Verzögerungen duldet. Dies sind z.B. Ausfall des Netzstroms (Power Fail) oder andere Alarmsituationen.

Wir halten fest: Ein Interrupt (Programmunterbrechung) ist ein zur Programmausführung asynchrones Ereignis, das die sequentielle Programmausführung unterbricht und die Kontrolle an ein spezielles Programm zur Behandlung des Ereignisses übergibt.

10.1 Ein einfaches System zur Unterbrechungsbehandlung

Abb. 10.1 zeigt den prinzipiellen Ablauf eines Interrupts. Nach dem Auftreten des Interrupts wird zunächst die gerade aktuelle Instruktion zu Ende geführt. Damit steht ein definierter

Zustand des unterbrochenen Programms zur Verfügung, an den nach der Abarbeitung des Interrupts zurückgekehrt werden kann. Die darauf folgenden Schritte sind:

1. Rettung des Prozessorstatus
2. Auswahl der Interrupt-Behandlungsroutine
3. Ausführung der Behandlungsroutine
4. Rückkehr vom Interrupt einschließlich der Wiederherstellung des Prozessorstatus

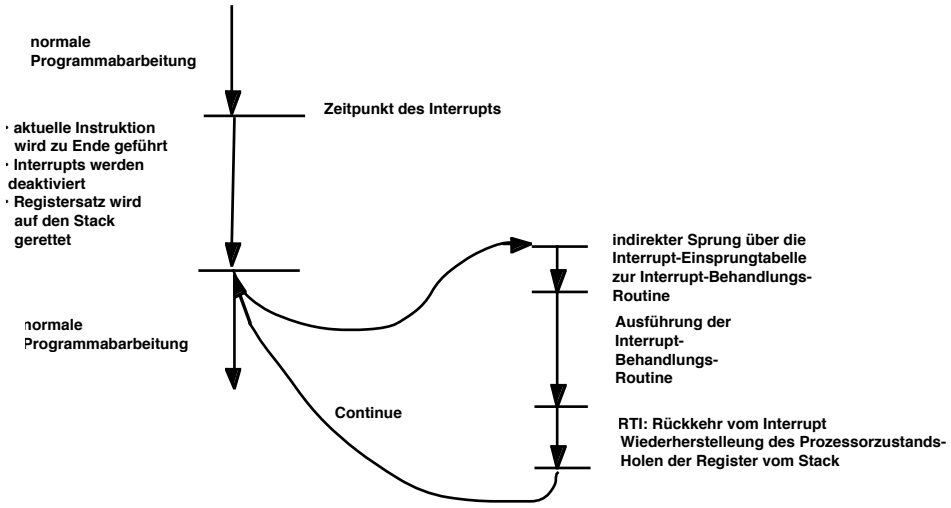


Abb. 10.1 Schritte bei der Interruptbehandlung

Um die jeweilige Bearbeitungsroutine für einen Interrupt zu bestimmen, ist im System eine Tabelle vorgesehen, welche die Einsprungadressen für die entsprechende Bearbeitungsroutinen enthält. Für jeden vom System unterstützten Interrupt ist in dieser Einsprungtabelle ein Eintrag vorgesehen. Beim Start des Systems wird diese Tabelle üblicherweise initialisiert, d.h. es werden die Adressen der Behandlungsroutinen eingetragen. Abb. 10.2 zeigt die vom MC6809 unterstützten Interrupts und die entsprechende Einsprungtabelle.

Interrupt	MSB	LSB
$\overline{\text{RESET}}$	FFFE	FFFF
$\overline{\text{NMI}}$	FFFC	FFFD
SWI	FFFA	FFFB
$\overline{\text{IRQ}}$	FFF8	FFF9
$\overline{\text{FIRQ}}$	FFF6	FFF7
SWI2	FFF4	FFF5
SWI3	FFF2	FFF3
reserviert	FFF0	FFF1

Abb. 10.2. Einsprungtabelle und Beschreibung der vom MC6809 unterstützten Interrupts

Die Interrupts habe folgende Bedeutung und Eigenschaften:

RESET:	Löscht den gesamten Prozessorstatus, Transferiert die Kontrolle an die Routine, deren Adresse in FFFE und FFFF steht.
IRQ:	Interrupt Request. Alle Register werden auf den Systemstack gerettet. Auf externen Leitungen (Bus Available und Bus State) wird angezeigt, daß der Interrupt akzeptiert wurde. Kontrolle wird zu der im entsprechenden Interrupt Vektor angegebenen Routine transferiert.
FIRQ:	Fast Interrupt Request: Nur der PC und das CCR werden gerettet. Sonst wie IRQ.
NMI:	Non-Maskable-Interrupt. Kann nicht ausgeschaltet (d.h. im CCR maskiert) werden. Sonst wie IRQ.
SWI:	Software Interrupt (Trap). Die SWI Instruktion erzeugt genau dieselbe Reaktion wie ein Hardware Interrupt. Der Unterschied liegt lediglich darin, daß er nicht durch ein externes Signal, sondern durch den entsprechenden Befehl ausgelöst wird, d.h. im strengen Sinne kein asynchrones Ereignis darstellt. Wird beim Debugging ausgenutzt, zur Behandlung von Ausnahmebedingungen und zur Emulation nicht vorhandener Hardware (z.B. Coprozessor-Traps).
SWI2,SWI3:	Wie SWI. Einziger Unterschied, I- und F-Bit im CCR werden NICHT modifiziert

Zunächst müssen wir zwischen:

- Hardware-Interrupts: RESET, IRQ, FIRQ und NMI und
- Software-Interrupts: SWI, SWI2, SWI3

unterscheiden.

Hardware-Interrupts sind die oben erwähnten, zum Programmablauf asynchronen Ereignisse. Sie werden über physikalische Signale an den Pins des Prozessors aktiviert. Ihnen gilt unser Hauptinteresse an dieser Stelle. Software-Interrupts sind Befehle, die im sequentiellen Befehlsstrom liegen. Ein Software-Interrupt hat große Gemeinsamkeiten mit einem Unterprogrammaufruf. Die wesentlichen Gemeinsamkeiten und Unterschiede sind im folgenden angegeben:

- Gemeinsamkeiten von SWI und Unterprogrammaufrufen:
  - Synchron zum sequentiellen Befehlsstrom,
  - Rücksprungadresse wird automatisch gesichert und bei RTS bzw. RTI benutzt.

- Unterschiede:
  - beim SWI werden häufig (z.B. beim MC6809) alle Register automatisch gesichert, beim Unterprogrammaufruf nur der PC,
  - Beim SWI muß keine Zieladresse angegeben werden, da sie durch die Initialisierung der Einsprungtabelle feststeht. Dies hat einen großen praktischen Vorteil: Falls sich die Adresse der Behandlungsroutine ändert, muß dies nur an einer einzigen Stelle in der Interrupteinsprungtabelle aktualisiert werden. In den Programmen, in denen der Software-Interrupt genutzt wird, ist keine Modifikation notwendig.

Der Software-Interrupt wird häufig beim Debugging zur Realisierung von Breakpoints angewandt oder zur Emulation von spezieller Hardware, wenn diese im System (noch) nicht physisch vorhanden ist.

Hardware-Interrupts lassen sich beim MC 6809 zunächst nach folgenden Gesichtspunkten unterscheiden:

1. Abschaltbare und nicht abschaltbare Interrupts,
2. Interrupts, die den vollständigen Prozessorzustand retten und solche, die nur einen Teil des Prozessorzustands retten.

RESET und NMI gehören zu den nicht abschaltbaren Interrupts. Dabei geht bei RESET der gesamte Prozessorzustand verloren und wird neu initialisiert. Reset wird meist in Situationen angewandt, in denen die CPU sich in einem, vom Anwender gesehen, nicht definierten Zustand befindet, eine für jeden, der mit Rechnern in Berührung kommt, bekannte Situation. NMI rettet den gesamten Prozessorzustand. Die Anwendungen für NMI sind z.B. die Rettung des Zustands bei Stromausfall (Power Fail). NMI ist ebenfalls sinnvoll, wenn ein kritischer (externer) Prozeß überwacht werden soll.

Die Interrupts IRQ und FIRQ werden durch eine Belegung des CCR an- bzw. abgeschaltet. Abb. 10.3 zeigt die entsprechenden Bits des CCR. Die Belegung kann als Bitmaske aufgefaßt werden, weshalb abschaltbare Interrupts auch als maskierbare Interrupts bezeichnet werden.

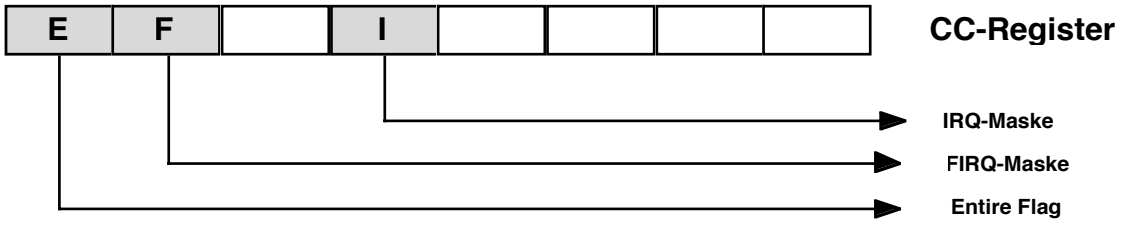


Abb. 10.3 Bitpositionen der Interruptmaske im Condition Code Register (CCR)

Bit *I* ist das Steuerbit für IRQ, wobei IRQ aktiviert ist, wenn *I*=0 ist. Die Maske zur Aktivierung ist deshalb xxx0 xxxx, zur Deaktivierung xxx1 xxxx. Die Deaktivierung wird durch eine ODER-Operation, die Aktivierung durch eine UND-Operation auf dem CCR mit der entsprechenden Maske durchgeführt. Entsprechendes gilt für den FIRQ, der durch das Bit *F* und die Masken x0xx xxxx und x1xx xxxx gesteuert wird. Das Entire Flag *E* gibt an, ob während einer Interruptbearbeitung der gesamte Registersatz (bei IRQ und NMI) oder nur Teile davon (bei FIRQ) gerettet wurden. Bei FIRQ wird nur der Programmzähler und das CCR gerettet. Deshalb erlaubt dieser Interrupt eine besonders schnelle Antwort auf externe Ereignisse. Für den IRQ, NMI und den FIRQ ist im folgenden die Anzahl der Zyklen für die Interrupt-Bearbeitung für den MC 6809 angegeben. Dabei wird die Interrupt-Berabeitung vom Auftreten des Interrupts bis zur Bearbeitung des ersten Befehls der Interrupt-Behandlungsroutine gerechnet.

Aufwand für die Interrupt-Behandlung:

	Zyklen
Normale Antwortzeit für IRQ und NMI	21
Normale Antwortzeit für FIRQ	12

In Abb. 10.4 ist der Ablauf der Interruptbehandlung und die daran beteiligten Systemkomponenten schematisch dargestellt.

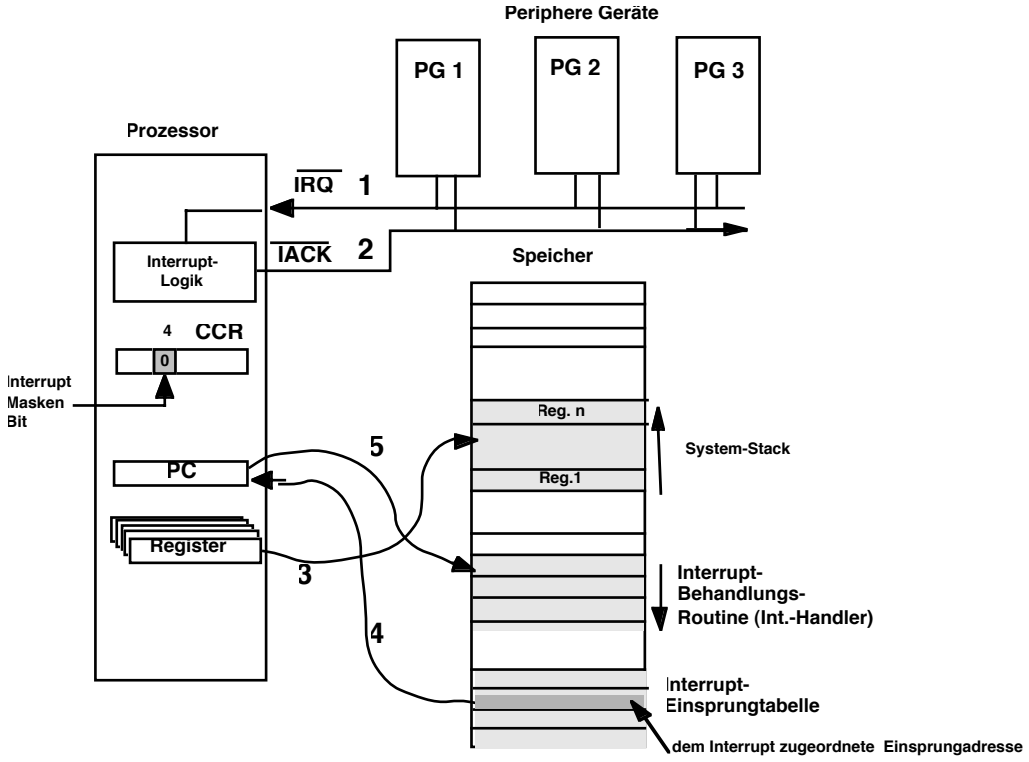


Abb. 10.2 Phasen der Unterbrechungsbearbeitung

PG1,..., PG3 stellen periphere Geräte dar. Sie sind über eine gemeinsame Interrupt-Leitung mit dem IRQ-Eingang des Prozessors (in diesem Beispiel nehmen wir den MC6809 an) verbunden. In der Phase **1** sendet ein Gerät eine Interrupt-Anforderung (Interrupt Request). Die Interrupt-Logik wertet das entsprechende Maskenbit im CCR aus und, falls der Interrupt nicht deaktiviert ist, legt ein Bestätigungssignal (Interrupt Acknowledge, IACK)[1] auf die entsprechende Leitung (Phase **2**). Im nächsten Schritt werden die Register auf den Systemstack geschrieben, wobei die in Abschnitt 7.4.3 beschriebene Stack-Ordnung gilt. Der Programmzähler wird von der Einsprungadresse des IRQ in der Interrupt-Einsprungtabelle mit der Adresse der Interrupt-Behandlungsroutine geladen (Phase **4**) und zeigt damit auf den ersten Befehl der Behandlungsroutine, der als nächster ausgeführt wird (Phase **5**).

Ein Beispiel soll die Programmierung der Interruptverarbeitung näher erläutern. Abb. 10.5 zeigt dazu die Systemkonfiguration.

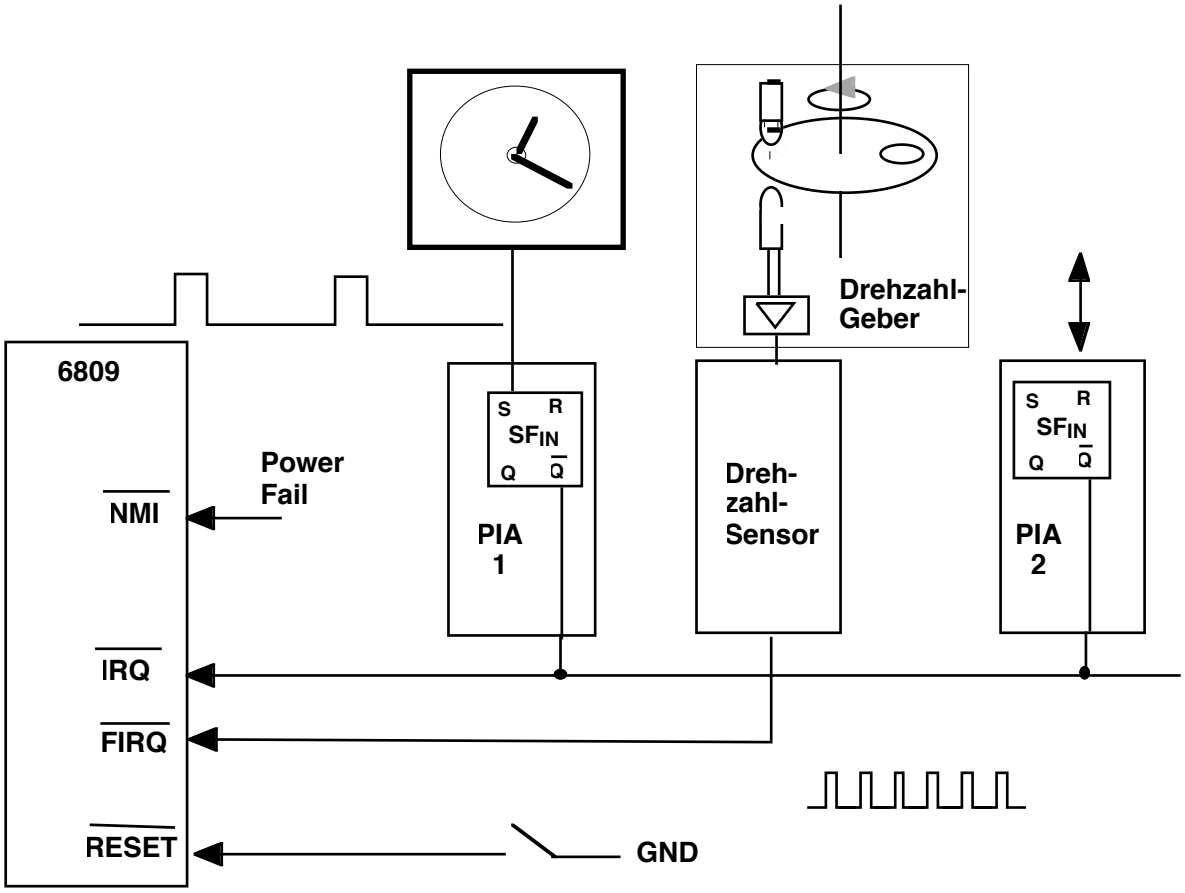


Abb. 10.5 Anordnung zur Drehzahlmessung

[1] Beim MC 6809 wird das IACK-Signal aus den Bus-Signalen BA (Bus Available) und BS (Bus State) abgeleitet. Ein IACK-Signal wird bei der Belegung BA=0, BS=1 erzeugt.

Die Aufgabe ist, die Drehzahl z.B. eines Motors oder eines Windgeschwindigkeitsmessers zu bestimmen. Der Drehzahlgeber besteht aus einer Scheibe mit einem Loch und einer Lichtschranke. Diese Anordnung erzeugt bei jeder Umdrehung einen Impuls, der nach geeigneter Impulsformung dem FIRQ-Eingang der CPU zugeführt wird.

Wir wählen den FIRQ-Eingang aus zwei Gründen:

1. Weil wir annehmen, daß die Impulse mit hoher Frequenz auftreten.
2. Weil die entsprechende Behandlungsroutine einfach ist, und wir deshalb kein oder nur wenige Register benötigen. Deshalb müssen auch vor der Ausführung der Behandlungsroutine keine Register gerettet werden, deren Inhalt möglicherweise durch die Behandlungsroutine zerstört würde.

Die Echtzeituhr, die einen periodischen Zeitimpuls erzeugt,wird über einen PIA angeschlossen. Das Handshake-Signal, das das SF_{IN} des PIA setzt, löst damit auch das IRQ-Signal aus. Dabei wird das SF_{IN} gesetzt. Dies ermöglicht es der CPU, bei mehreren angeschlossenen PIAs durch Abfrage festzustellen, wer den Interrupt ausgelöst hat.

In der Initialisierungsphase des Systems müssen die entsprechenden Einsprungadressen der Behandlungsroutinen von IRQ und FIRQ in die Interrupt-Einsprungtabelle eingetragen werden. Die entsprechende Belegung ist in Abb. 10.6 angegeben

FFFF	xx	Einsprungadresse für RESET
FFFE	xx	
FFFD	xx	Einsprungadresse für NMI
FFFC	xx	
FFFB	xx	Einsprungadresse für SWI
FFFA	xx	
FFF9	00	Einsprungadresse für IRQ
FFF8	F0	
FFF7	10	Einsprungadresse für FIRQ
FFF6	01	
FFF5	xx	Einsprungadresse für SWI2
FFF4	xx	
FFF3	xx	Einsprungadresse für SWI3
FFF2	xx	
FFF1	•	reserviert
FFF0	•	

Abb. 10.6 Organisation der Interrupt-Einsprungtabelle

Wir wollen nun zunächst das Programm betrachten, das die Umdrehungszählung realisiert. Es besteht aus einem Initialisierungsteil, der ausgeführt werden muß, bevor das Programm ablaufen kann (Abb. 10.7).

Initialisierung des Umdrehungszählers:

```
                BF    INTEN       EQU   %1011 1111     Setzt das F-Flag im CCR  (auf  0)

                                  ORG   $A000
Progr. Zeile
1               RCINIT            CLR   REVCNT         Initialisieren des Umdr. Zählers mit 0
2                                 ANDCC #INTEN         Aktivieren des Interrupts FIRQ im CCR
  •                               •
  •                               •
```

Interrupt Service Routine für den Umdrehungszähler:

```
                                  ORG   $0110
Progr. Zeile

1                                 INC     REVCNT   Incrementieren des Umdrehungszählers
2                                 RTI              Rückkehr aus der Interrupt Service Routine

3               REVCNT            RMB   1          Speicherplatz für den Umdrehungszähler
```

Abb. 10.7 Programm zur Umdrehungszählung

Der Initialisierungsteil aktiviert den FIRQ indem er das entsprechende Flag im CCR setzt (auf 0!). Dies wird durch die speziell dafür vorgesehene Instruktion ANDCC ausgeführt. Die Maske dazu wird im Definitionsteil durch die Assemblerdirektive: INTEN EQU %1011 1111 festgelegt.

Tritt ein Interrupt auf, wird der vollständige Registersatz, bzw. bei FIRQ nur PC und CCR gerettet. Danach werden IRQ und FIRQ automatisch (vom Prozessor) deaktiviert, damit die Behandlungsroutine ohne weitere Unterbrechung ausgeführt werden kann[2]. In der Behandlungsroutine wird der Umdrehungszähler REVCNT erhöht. Bei RTI wird der Inhalt der Register vom Stack geladen, so daß auch der Zustand des CCR vor dem Interrupt wiederhergestellt wird. Insbesondere sind IRQ und FIRQ wieder im Aktivierungszustand wie vor der Interrupt-Behandlung.

Die Behandlung des FIRQ ist sehr einfach, da nur eine einzelne Interruptquelle angeschlossen ist. Die gesamte Behandlung des FIRQ wird deshalb direkt in der Service-Routine für die An-

[2] Sollen während der Behandlungsroutine weitere Interrupts erkannt werden, muß das CCR expizit gesetzt werden.

wendung, d.h. den Umdrehungszähler erledigt. Sind mehrere mögliche Interrupt-Quellen angeschlossen, wie im Fall des IRQ, muß zuerst festgestellt werden, welche Interrupt-Quelle den Interrupt ausgelöst hat.

```
                0100    CLCKS       EQU     $100        Einsprung Service Routine der Uhr
                C003    PIA1SF      EQU     $C003       Statusflags des PIA 1
                .....   PIA2SF      EQU     ........    Statusflags des PIA 2

                                    ORG     $F000

1       F000    INTRPT              T       PIA1SF      Wurde Interrupt vom PIA1 ausgelöst ?
3                                   BNE     SERV1       JA, (Flags ≠ 0) springe zur Service Routine 1
4                                   TST     PIA2SF      Wurde Interrupt vom PIA 2 ausgelöst?
5                                   BNE     SERV2       JA, (Flags ≠ 0) springe zur Service Routine 2
6                                   •
7                                   •
8               SERV1               BSR     CLCKS       Springe zur Service Routine der Uhr
9                                   BRA     EXIT
10              SERV2               BSR      .....      Springe zur entsprechenden Service Routine
11                                  BRA     EXIT
12                                  •
13                                  •
14              EXIT                RTI                 Rückkehr zum normalen Ausführungsmodus
15
```

Abb.10.8 Allgemeine Behandlungsroutine für Interrupts

Zu diesem Zweck ist eine allgemeine Behandlungsroutine vorgesehen, aus der heraus die speziellen Service-Routinen aufgerufen werden. Abb. 10.8 zeigt ein solches Programmstück.

Die Initialisierung und die spezielle Behandlungsroutine für die Echtzeituhr ist in Abb. 10.9 angegeben. In der Initialisierungsroutine wird der Zeit-Zähler *TIMECT* auf seinen Ausgangswert 0 gesetzt und der IRQ aktiviert. Außerdem wird ein Flag *CNFLAG* initialisiert, das es dem Anwendungsprogramm ermöglicht, zu überprüfen, ob ein Interrupt stattgefunden hat. Dieses Flag wird in der Interrupt-Behandlungsroutine gesetzt.

```
                C001   PIAINP      EQU   $C001          PIA-Eingangsregister
                C003   PIA1SF      EQU   $C003          PIA-Status-Flags
                  EF   INTEN       EQU   %1110 1111     Maske zur Aktivierung des Interrupts

                              ORG   $1000
```

Initialisierung und Warteschleife:

```
Progr. Zeile

1                             CLR     TIMECT      Rücksetzten des Zeitzählers(Time-Counter)
2                             ANDCC #INTEN        Aktivieren des Interrupts IRQ im CCR
3                      CNTLP  CLR     CNFLAG      Rücksetzen des Flags
4                      WTCLK  TST     CNFLAG      Wurde ein Clock-Interrupt ausgelöst?
5                             BEQ     WTCLK       Nein (CNFLAG = 0), Warten
6                             •                           •
7                             •                   Aktivität, falls Clocktick erfolgte
8                             •                           •
9                          BRA     CNTLP
10                         •
11             TIMECT      RMB     1       Reservierung des Speicherplatzes für den Zeitzähler
12             CNFLAG      RMB     1       Reservierung von Speicherplatz für das CNFLAG
```

Interrupt Service Routine:

```
                                  ORG     $100
Progr. Zeile

1          0100                   LDA     PIAINP      Dummy Read setzt  Flag im PIA-Status zurück
2                                 INC     TIMECT      Incrementiert Zeitzähler
3                                 INC     CNFLAG      Setze Flag, signalisiert Increment des TIMECT
4                                 RTS                 Rückkehr zur generellen Interrupt-Routine
```

Abb. 10.9 Programm für die Echtzeituhr

In Abb. 10.10 sind Umdrehungszählung und Zeitzählung zu einem Programm zur Drehzahlmessung zusammengefaßt. Es sollen die Umdrehungen / Zeiteinheit gemessen, und bei Überschreitung ein Alarm ausgelöst werden. Dazu wird eine Zeitkonstante, welche die Ticks der Uhr pro Zeitintervall festlegt, in einer Schleife mit dem aktuellen Stand des Zeitzählers verglichen, der bei jedem Tick der Uhr hochgezählt wird.

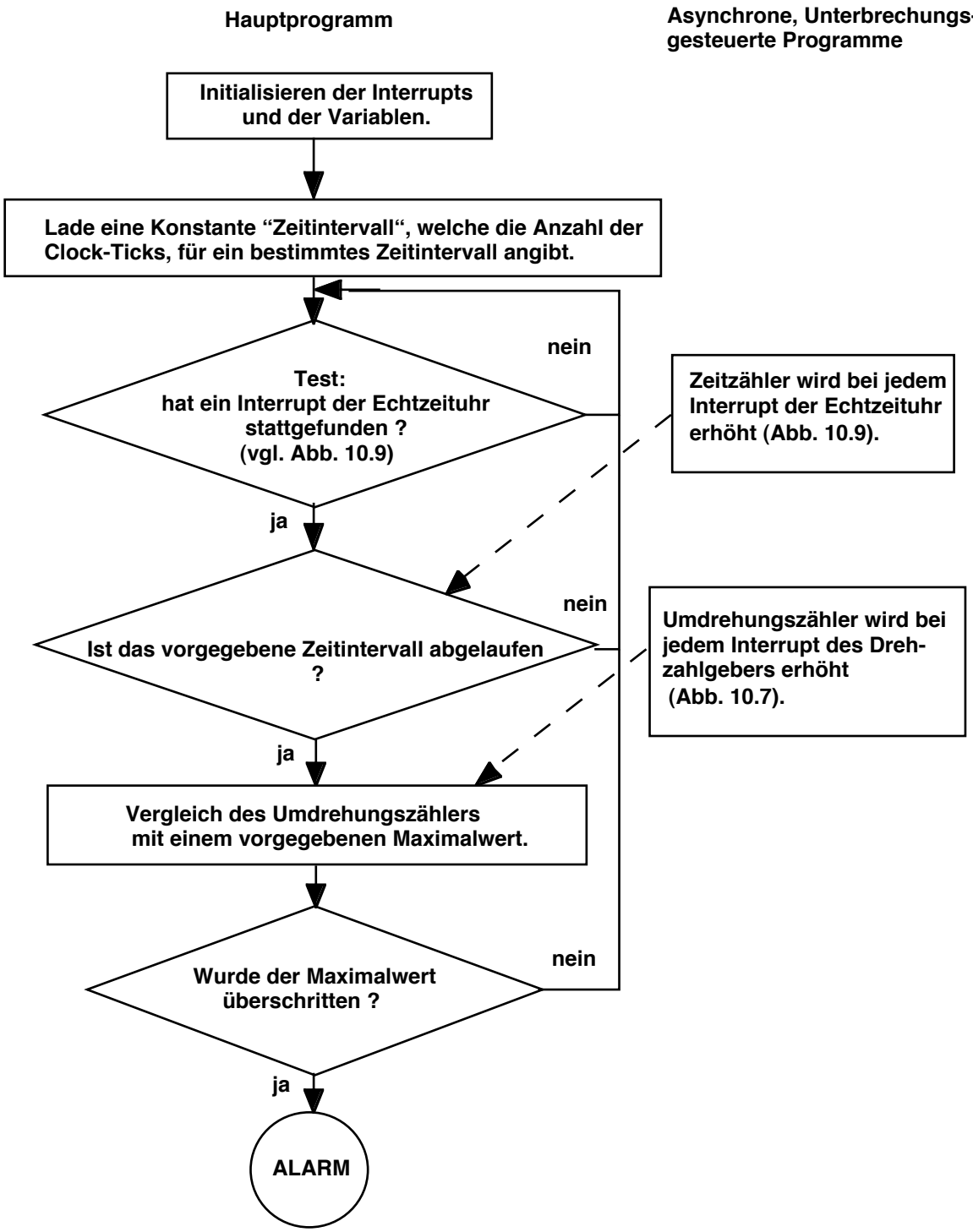


Abb. 10.10 Drehzahlmessung mit Alarmauslösung

Ist ein bestimmtes Zeitintervall abgelaufen, wird die Umdrehungszahl gelesen. Der Wert wird mit einem Maximalwert verglichen und bei Überschreitung wird ein Alarm ausgelöst. Die Zähler für die Ticks der Uhr und die Umdrehungen werden völlig asynchron zum Ablauf des Programms durch die entsprechenden Interrupt-Behandlungsroutinen aktualisiert.

Das Beispiel zeigt ein einfaches Interrupt-System. Dabei treten noch folgende Probleme auf:

1. Die Bestimmung des auslösenden Gerätes, wenn mehrere Geräte an eine Interrupt-Leitung angeschlossen sind, erfolgt durch explizite Abfrage des entsprechenden Status. Dieses Verfahren haben wir bereits im vorigen Kapitel als Polling kennengelernt. Dies kann bei vielen angeschlossenen Geräten zu einem Problem werden, da es entsprechend lange dauert.

2. unter den Interrupts gibt es keine Rangordnung, sieht man von der “First-Come-First-Served"-Ordnung ab, die durch die Sperrung des Interrupt-Systems während einer Interruptbearbeitung durchgesetzt wird.

10.2 Vektorisierte Unterbrechungsbehandlung

Das Interrupt-System des 68000 Prozessors löst beide oben angedeuteten Probleme. Es unterstützt die automatische Bestimmung des unterbrechenden Geräts und die Definition einer Rangordnung unter den Interrupt-Quellen. Abb. 10.11a und b zeigen das vollständige Interrupt-System und die vom Prozessor bereitgestellte Unterstützung.

In Abb. 10.11a ist das Prinzip des Interruptsystems dargestellt. Im Hauptspeicher wird an einer definierten Stelle eine Interruptvektortabelle (IVT) angelegt, deren Basisadresse entweder durch die Hardware festgelegt ist (beim 68000 sind fest die oberen 256 Worte des Adreßraums für die Interruptvektortabelle reserviert) oder, wie in der Abbildung gezeigt, durch ein Vektor-Basis-Register (z.B. 68020) spezifiziert wird. Die IVT enthält die Einsprungadressen für die unterschiedlichen Behandlungsroutinen, die jeweils für ein peripheres Gerät (oder eine Gruppe peripherer Geräte) vorgesehen sind. Jedes periphere Gerät hat einen zugeordneten Interruptvektor I, der als Index in die IVT genutzt wird, um die Behandlungsroutine auszuwählen. Durch ein IRQ-Signal (Interrupt Request) des peripheren Gerätes an den Prozessor wird der Interrupt-Zyklus initiiert. Akzeptiert der Prozessor den Interrupt, sichert er den Status des Prozessors und aktiviert die IACK-Leitung. Das auslösende periphere Gerät sendet daraufhin seinen Interruptvektor über den Datenbus, der zur Basis der IVT addiert, die Einsprungadresse indiziert. Die Adresse wird in den Programmzähler geladen und die Programmausführung in der Behandlungsroutine fortgesetzt.

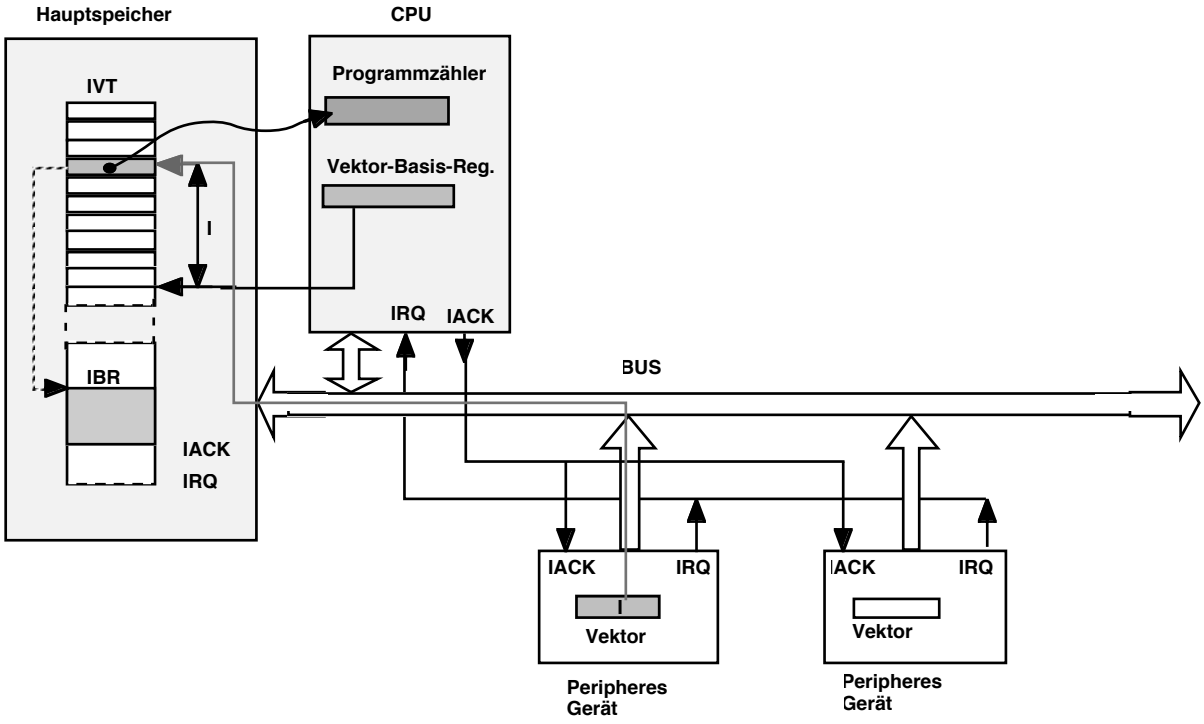


Abb. 10.11a Prinzip der vektorisierten Unterbrechungsbehandlung

Abb. 10.11b zeigt die Architekturunterstützung für Interrupts des Prozessors 68000.

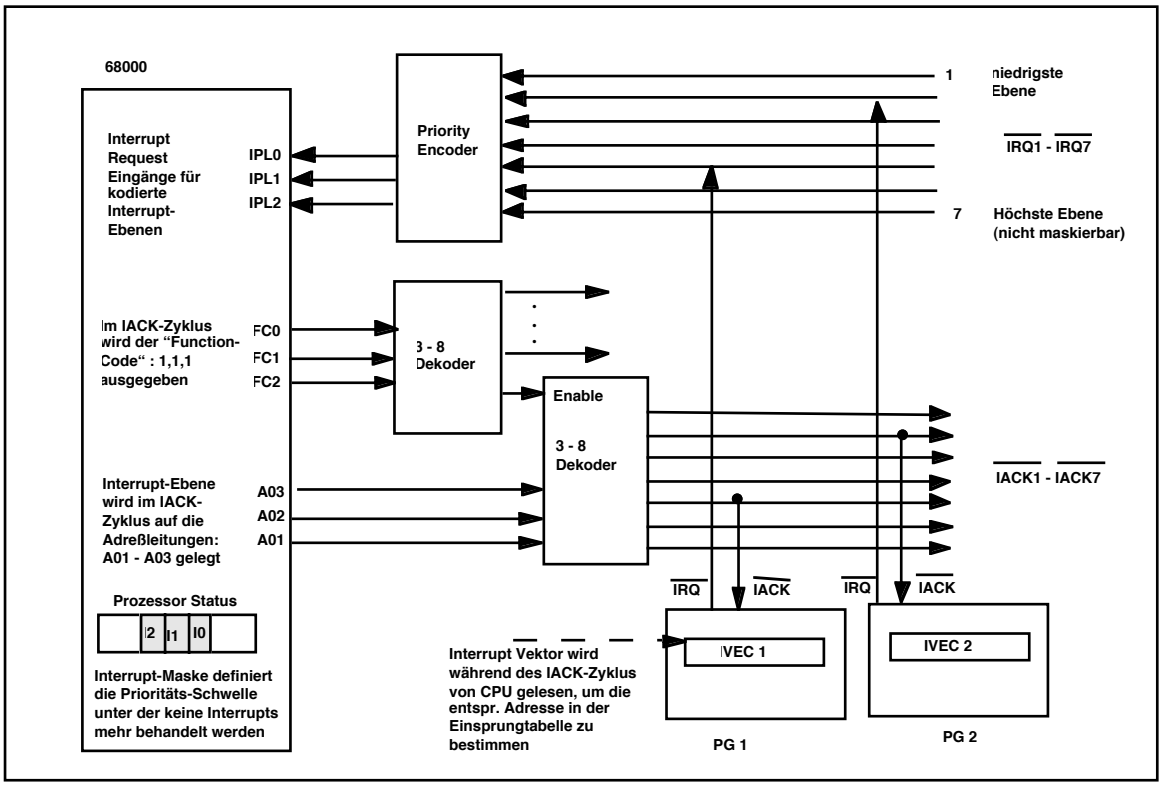


Abb. 10.11.b Vektorisiertes und Priorisiertes Interrupt-System des 68000

Dieser Prozessor unterstützt nicht nur einen vektorisierten Interrupt, sondern auch 8 verschiedene Interrupt-Prioritäten, die Interrupt-Ebenen genannt werden. Wird auf zwei unterschiedlichen Interrupt-Ebenen eine Anforderung signalisiert, so setzt sich die Anforderung der höheren Ebene durch. In Abb. 10.11b sind 7 Leitungen vorgesehen, auf denen Anforderungen an die entsprechende Ebene signalisiert werden können. Sie sind mit IRQ1 . . . IRQ7 bezeichnet. Die IRQ-Leitung eines peripheren Geräts (PG) ist jeweils mit einer dieser Leitungen verbunden. Der Prioritätskodierer (Priority Encoder) gibt die Kennung des höchsten anliegenden Interrupts als Binärzahl 001, 010,. . ., 111 an die Eingänge PL0, PL1, PL2 (PL: Priority Level) des Prozessors weiter. Die logische Schaltung eines Prioritätskodierers für 4 Eingänge ist in Abb. 10.12 wiedergegeben. An den Ausgängen A_1 und A_2 liegt die Adresse des höchstpriorisierten Eingangssignal I_0, I_1, I_2, I_3. Da für A_1 und A_2 auch die Belegung 00 möglich ist, zeigt ein zusätzlicher Ausgang s an, ob überhaupt ein Eingangssignal anliegt.

$$A_1 = \overline{I_3} \cdot I_2 + I_3 = I_2 + I_3$$
$$A_2 = \overline{I_3} \cdot \overline{I_2} \cdot I_1 + I_3 = I_3 + \overline{I_2} \cdot I_1$$
$$s = I_3 + I_2 + I_1 + I_0$$

Wahrheitstafel:

Eingänge				Adresse		activ
I_3	I_2	I_1	I_0	A_1	A_2	s
0	0	0	0	0	0	0
0	0	0	1	0	0	1
0	0	1	x	0	1	1
0	1	x	x	1	0	1
1	x	x	x	1	1	1

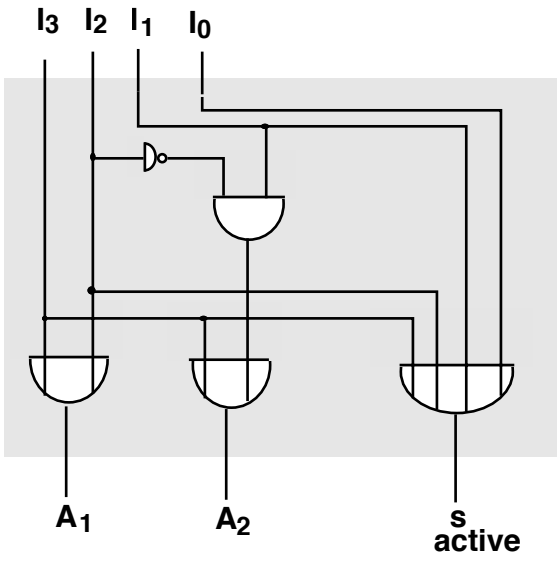


Abb. 10.12 Prioritätskodierer (Priority-Encoder)

Das Statusregister des 68000 besitzt eine Interrupt-Maske, die drei Bit umfaßt. Sie legt die Prioritätsebene fest, unter der keine Interrupts zur Bearbeitung angenommen werden. Nehmen wir an, daß der Prozessor gerade einen Interrupt-Request der Ebene 5 bearbeitet. Dis Maske im Statusregister wird dann auf 101 gesetzt. Kommt nun ein Interrupt-Request mit derselben oder einer niedrigeren Priorität wird er nicht berücksichtigt. Nur Interrupt-Requests deren Prioritäten über dieser Schwelle liegen (d.h. Ebene 6 und 7), werden vom Prozessor zur Bearbeitung angenommen. Der Status des gerade in Bearbeitung befindlichen Interrupt-Request wird dann

auf dem Stack gesichert, so daß die entsprechende Behandlungsroutine nach Beendigung der Bearbeitung des höher priorisierten Interrupts weitergeführt werden kann. Es immer möglich, daß ein höher priorisierter Interrupt einen niedriger priorisierten Interrupt unterbricht.

Nach Annahme eines neuen Interrupt-Requests startet der Prozessor einen sogenannten Interrupt-Acknowledge-Zyklus (IACK). Dabei werden die folgenden Aktivitäten durchgeführt:

- Auf den Statusleitungen F0, F1, F2 wird die Belegung 111 angezeigt, was bedeutet, daß der Prozessor einen System-Speicherzugriff durchführt (vgl. Kapitel 11). Die Belegung der Statusausgänge F0, F1, F2 wird als "Function Code" bezeichnet.
- Auf den niederwertigen Adreßleitungen wird die Prioritätsebene des akzeptierten Interrupts ausgegeben.

Der Funktionscode 111 wird durch einen Dekodierer in ein Steuersignal umgewandelt, das den Dekodierer der unteren Adreßleitungen aktiviert. Dadurch stehen sieben Interrupt-Acknowledge-Leitungen (IACK-Leitungen) zur Verfügung. Das Signal auf der IACK-Leitung signalisiert dem entsprechenden peripheren Gerät, daß sein Interrupt-Request akzeptiert wurde. Daraufhin legt das periphere Gerät eine 8-Bit-Kennung auf den Datenbus, den sogenannten Interrupt-Vektor, der vom Prozessor als Index in eine Einsprungtabelle interpretiert wird.

Der 68000 Prozessor hat eine Einsprungtabelle mit 256 Einträgen. Davon stehen dem Benutzer für Hardware-Interrupts etwa 200 zur Verfügung, so daß es bis zu dieser Grenze möglich ist, jedem peripheren Gerät einen individuellen Eintrag, und so eine spezielle Interrupt-Behandlungsroutine zuzuordnen. Die Indizierung und der Einsprung in diese Behandlungsroutine erfolgt automatisch wie beschrieben durch den Prozessor.

Da insgesamt 7 Prioritätsebenen aber sehr viel mehr Adressen für periphere Geräte existieren, ist es wahrscheinlich, daß mehrere Geräte auf einer gemeinsamen Prioritätsebene liegen. Ein Verfahren unter diesen eine weitere Rangordnung zu etablieren, ist das Daisy-Chaining-Verfahren, das in Abb. 10.13 skizziert ist. Eine IACK-Leitung wird in der gezeigten Weise durch eine Kette peripherer Geräte geschleift. Am Anfang der Kette ist das Gerät mit der höchsten Priorität positioniert. Hat es den Interrupt ausgelöst, unterbricht es die Kette, indem es über das Signal "Bypass" eine weitere Propagierung von IACK sperrt. Falls nicht, breitet sich das IACK-Signal entlang der Kette aus, bis das auslösende Gerät mit der höchsten Priorität gefunden wurde.

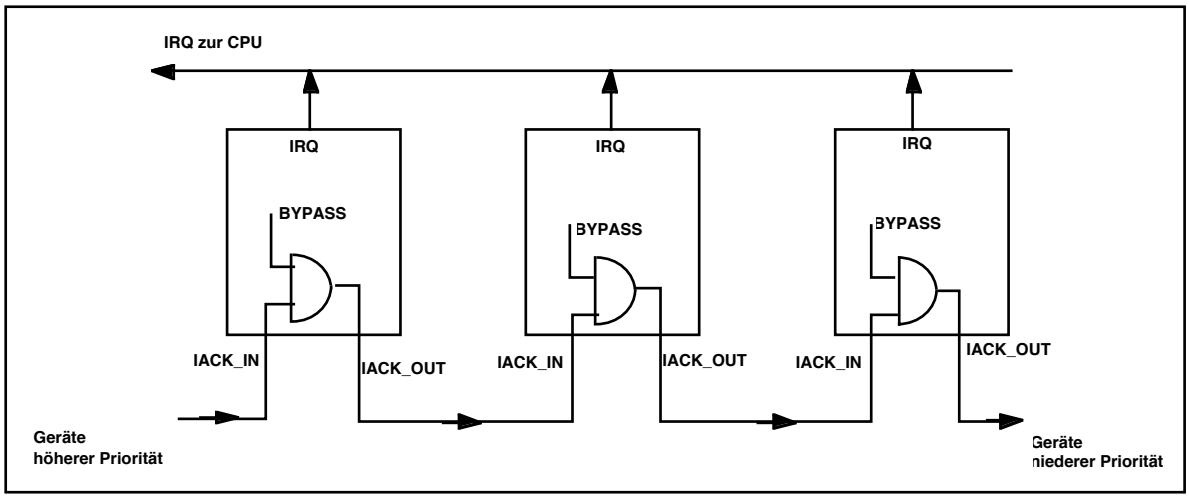


Abb. 10.13 Bestimmung des unterbrechenden Gerätes innerhalb einer Interrupt-Prioritäts-Ebene durch Daisy-Chaining

Zusammenfassend soll festgehalten werden:

- Ein vektorisiertes Interrupt-System dient der automatischen Bestimmung des auslösenden peripheren Geräts. Der Interrupt-Vektor ist eine Kennung des auslösenden Geräts, der als Index in eine Einsprungtabelle für Behandlungsroutinen verwendet wird.

- Prioritätsebenen dienen dazu, eine Rangfolge der anliegenden Interrupts festzulegen. Der Prozessor 68000 unterstützt sieben Prioritätsebenen. Interrupts höherer Priorität können jederzeit eine Interruptbehandlungsroutine unterbrechen, die einen Interrupt mit niedrigerer Priorität bearbeitet. Es ist externe Kodierungs/Dekodierungshardware erforderlich, um das vollständige System zu realisieren.

- Daisy-Chaining wird zur Bestimmung einer Rangfolge innerhalb einer Prioritätsebene genutzt. Es ist ein langsames aber ohne größere Kosten beliebig erweiterbares Verfahren. In einem einfachen Interrupt-System wie dem des 6809 kann Daisy-Chaining auch als Hardware-Alternative zum Polling eingesetzt werden.

11 Ein 32-Bit Prozessor

68020 Eigenschaften:

- "Echter" 32-Bit-Prozessor
- 4 GByte direkt adressierbarer Adreßraum
- 4 getrennte Adreßräume : 2 System [Instruktionen/ Daten]
  2 Benutzer [Instruktionen/ Daten]
- 32-Bit Adreß- und Datenregister

- 8 Datenregister
- 8 Adreßregister (Adreßregister #7 ist der Stackpointer für Unterprogrammaufrufe)
- 2 Supervisor-Stack Pointer (Master- und Interrupt-SP)
- 5 Spezial-Kontrollregister

- 18 Adressierungsarten
- 7 Datentypen

- Flexible Busstruktur
- Coprozessor-Schnittstelle
- Instruktions-Cache

68020 Befehlssatz:

Daten Transfer (Data Movement)
Arithmetische Operationen (Integer)
BCD Operationen
Logische Operationen
Shift und Rotate Operationen

Befehle zur Programmkontrolle

Einzelbit Befehle
Bitfeld Befehle

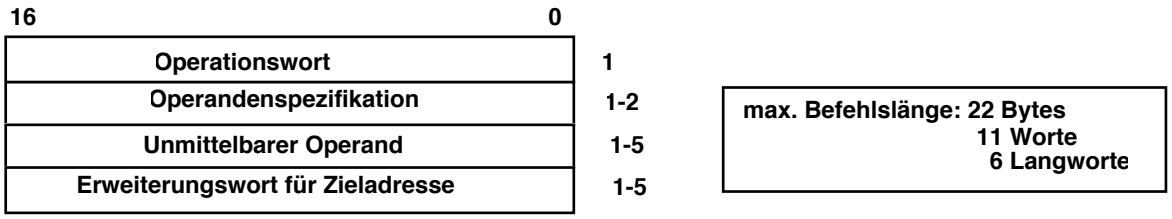
Befehle zur Systemkontrolle

Befehle zur Multiprozessorkommunikation

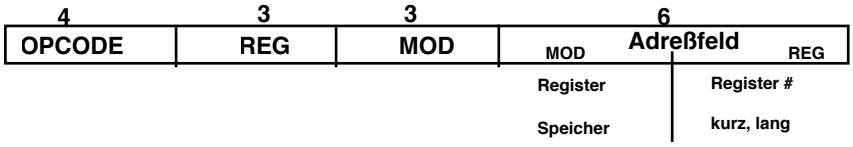
Coprozessor Befehle

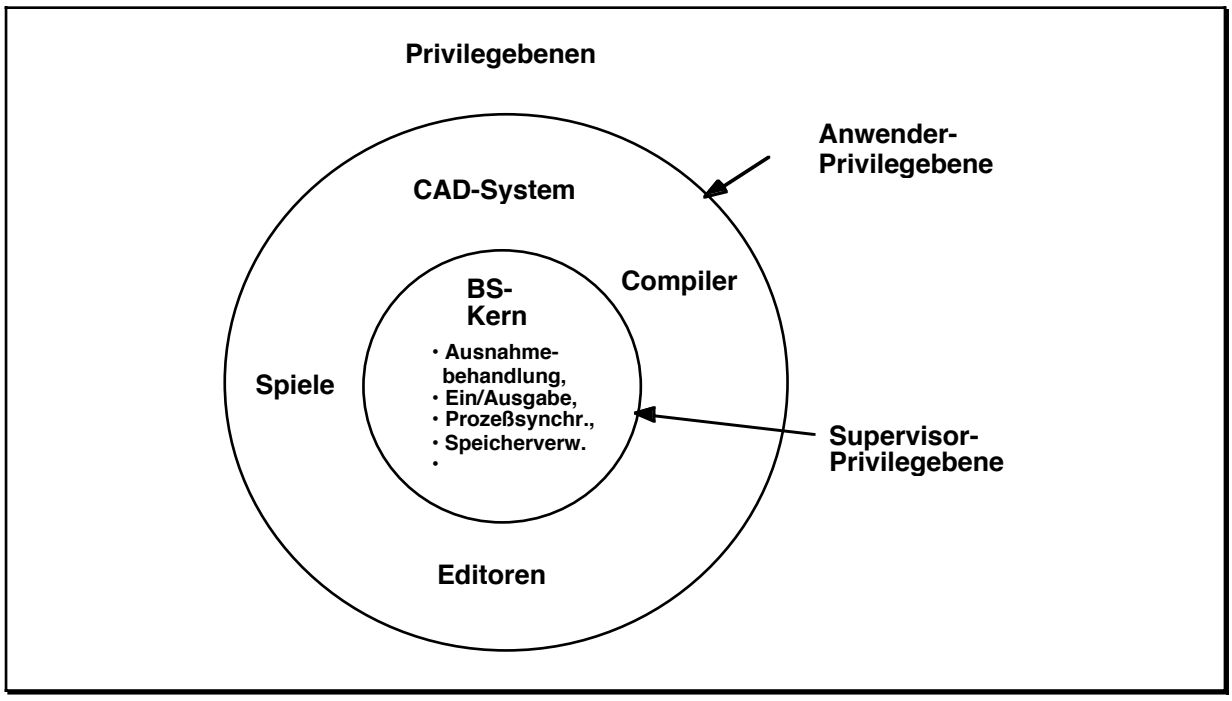
68020 Befehlsformat:

Genereller Aufbau eines 68020 Befehls:



Register/Register- oder Register/Speicher - Format





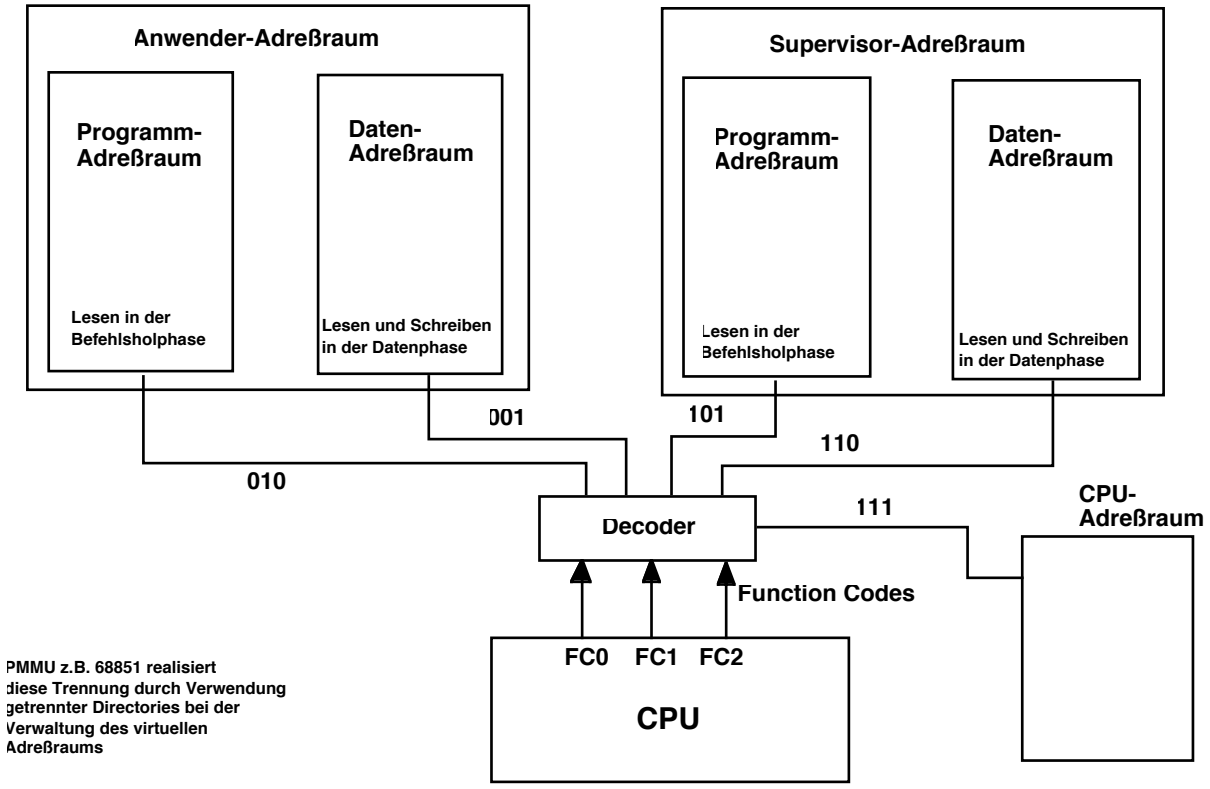
Prozessor Zustände:

- **Normaler Zustand**
- **Ausnahmezustand (exception processing state)**
- **Halt Zustand**

Privilegebenen:

- **Supervisor Ebene**
  - **S-Bit im Status Register ist gesetzt**
  - **Function Codes signalisieren den Supervisor Adreßraum**
  - **M-Bit im Status Register unterscheidet zwischen "Master State" und "Interrupt State". Interrupt State entspricht dem 68000/68008/68010 Supervisor State.**
- **Benutzer Ebene**
  mit externer Hardwareunterstützung kann der 68020 bis zu 256 Privilegebenen innerhalb der Benutzerebene realisieren.)

Unterscheidung der Adreßräume durch die Funktionscodes



Unterscheidung von:

Anwenderadreßraum für Programmcode
Anwenderadreßraum für Daten
Supervisoradreßraum für Programmcode
Supervisoradreßraum für Daten

Function Code Belegung:

FC0	FC1	FC2	Art des R/W-Zyklus
0	0	0	nicht definiert (reserviert)
0	0	1	User Data Space
0	1	0	User Program Space
0	1	1	nicht definiert (reserviert)
1	0	0	nicht definiert (reserviert)
1	0	1	Supervisor Data Space
1	1	0	Supervisor Program Space
1	1	1	CPU Space

Im CPU Space bearbeitet der Prozessor:

1. Interrupts
2. Traps
3. Breakpoints
4. Kommunikation mit einem Coprozessor
5. Moduloperationen

Zielkonflikte bei der Realisierung eines Registersatzes:

· Anzahl der Register:

· Zitat: Entweder ein Register (Acc) oder unendlich viele !

Unterschiede in der Bedeutung und im Gebrauch der Register:
- 1 Register: Zwischenspeicherung von Operanden zwischen aufeinanderfolgenden Befehlen
- meherere Register : Speicherung eines Working Sets (z.B. für eine Prozedur)
Ähnlichkeiten mit Cache, aber explizite Verwaltung d.h. ein Cache ist transparent und unterstützt das Modell eines (unendlich) großen Speichers.
Registerressourcen sind inhärent beschränkt und Bedürfen der expliziten Verwaltung durch das Anwenderprogramm

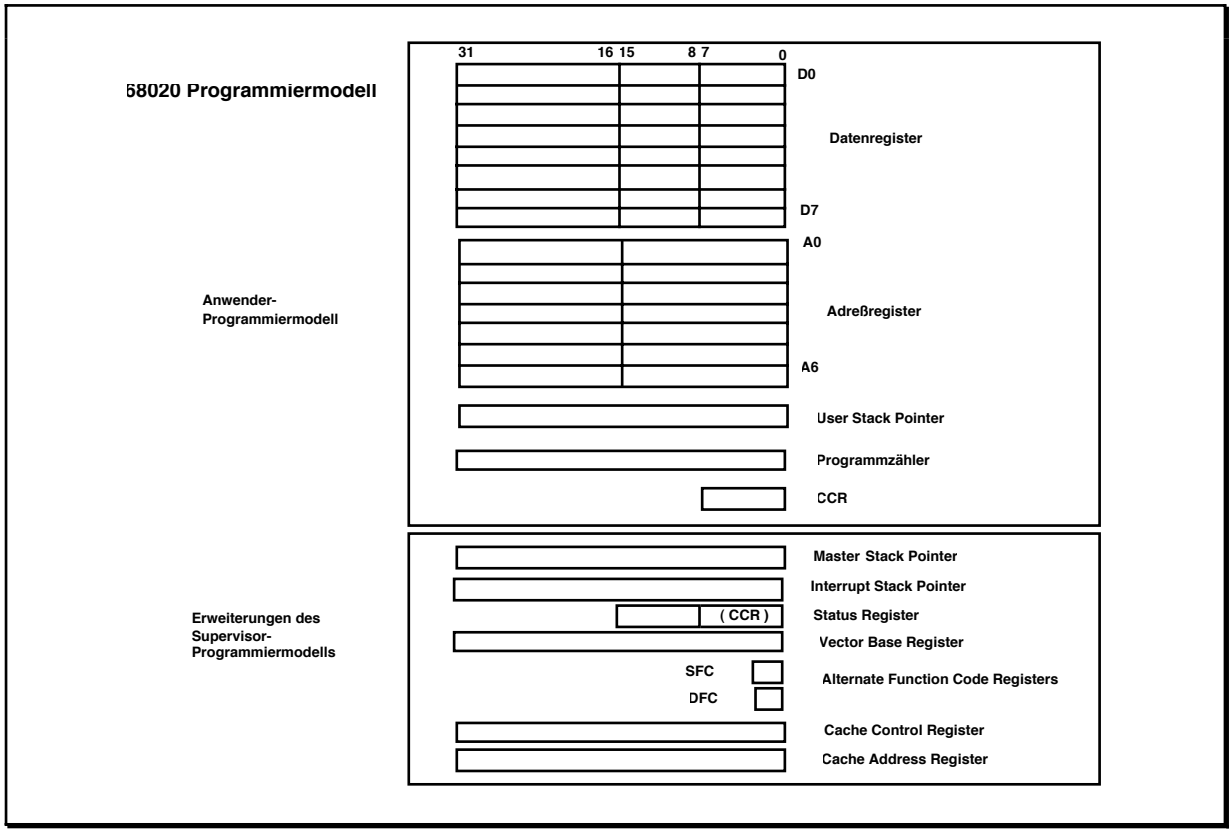
· Zielkonflikte zwischen: Länge des Befehlswortes, Anzahl der OPCODES, Informationen über Adressierungsarten

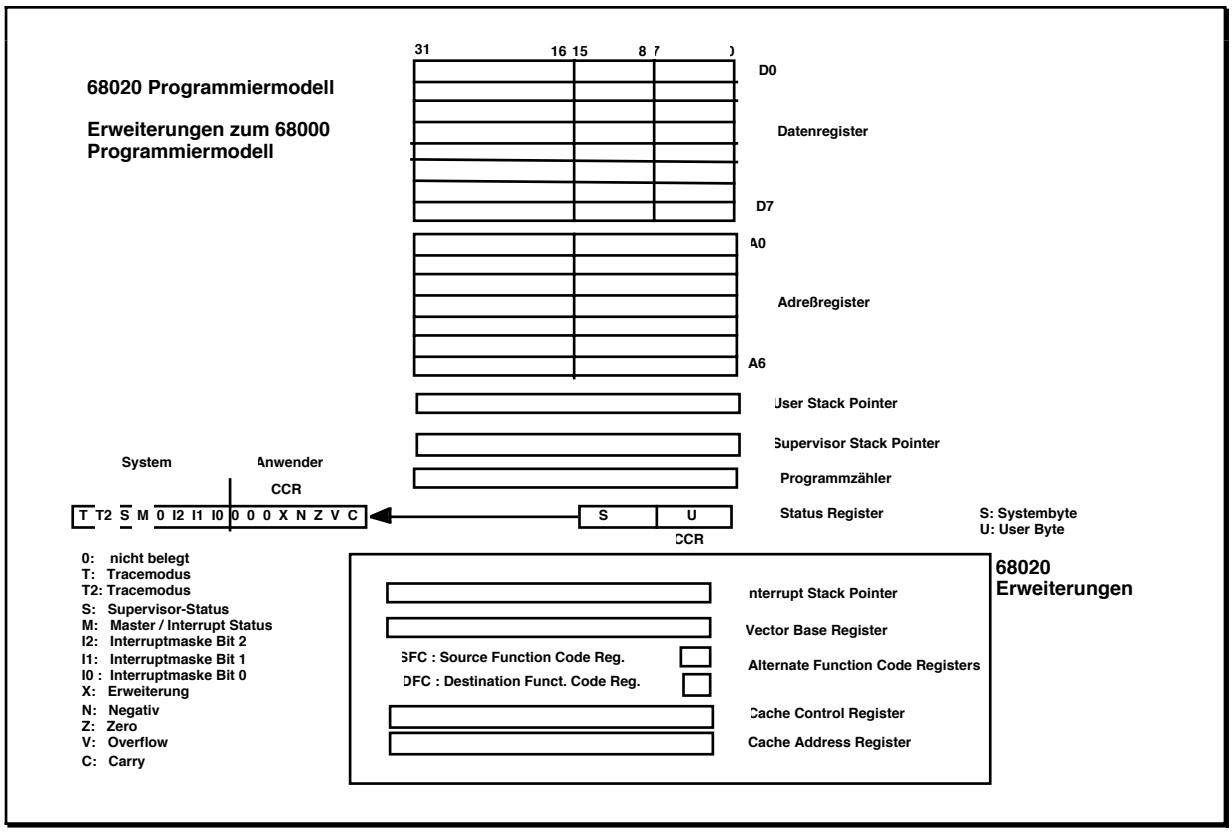
· Problem der Registerallokation und Verwaltung bei sehr vielen Registern

· Assembler-Programmierer kann durch Ausnutzung der programmsemantik eine sehr effiziente Registernutzung vornehmen, z.B. Bewahrung von Registerinhalten über Prozeduraufrufe hinweg oder sogar über Context-Wechsel

· Complier: Löschen aller Register bei Unterprogrammsprung

· Probleme beim "Retten" eines großen Registersatzes bei Unterprogrammsprüngen / Contextwechsel

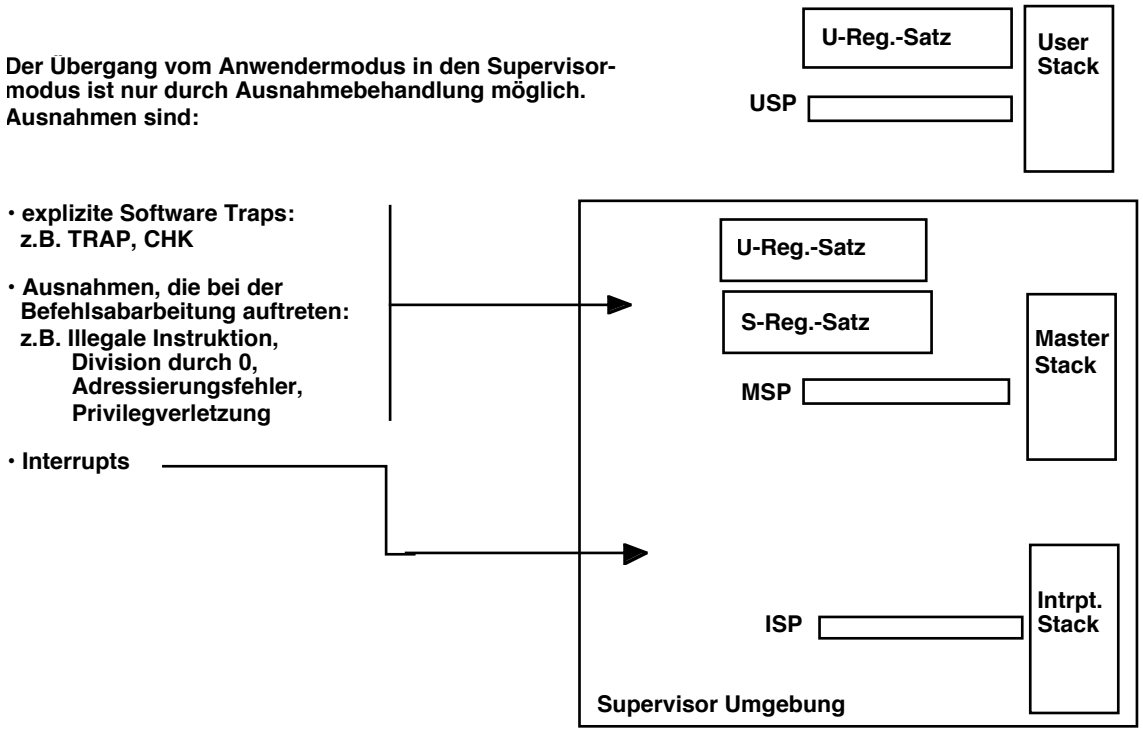

68020 Programmiermodell
Anwender-
Programmiermodell
31
16 15
8 7
0
D0
Datenregister
D7
A0
Adreßregister
A6
User Stack Pointer
Programmzähler
CCR
Erweiterungen des
Supervisor-
Programmiermodells
Master Stack Pointer
Interrupt Stack Pointer
(CCR)
Status Register
Vector Base Register
SFC
DFC
Alternate Function Code Registers
Cache Control Register
Cache Address Register


68020 Programmiermodell
Erweiterungen zum 68000
Programmiermodell
31
16 15
8 7
0
D0
Datenregister
D7
A0
Adreßregister
A6
User Stack Pointer
Supervisor Stack Pointer
Programmzähler
Status Register
S: Systembyte
U: User Byte
System
Anwender
CCR
T T2 S M 0 I2 I1 I0 0 0 0 X N Z V C
S
U
CCR
0: nicht belegt
T: Tracemodus
T2: Tracemodus
S: Supervisor-Status
M: Master / Interrupt Status
I2: Interruptmaske Bit 2
I1: Interruptmaske Bit 1
I0 : Interruptmaske Bit 0
X: Erweiterung
N: Negativ
Z: Zero
V: Overflow
C: Carry
nterrupt Stack Pointer
Vector Base Register
SFC : Source Function Code Reg.
DFC : Destination Funct. Code Reg.
Alternate Function Code Registers
Cache Control Register
Cache Address Register
68020
Erweiterungen

Ubergang vom Anwender in den Supervisor Zustand

Der Übergang vom Anwendermodus in den Supervisor-modus ist nur durch Ausnahmebehandlung möglich. Ausnahmen sind:

- explizite Software Traps: z.B. TRAP, CHK
- Ausnahmen, die bei der Befehlsabarbeitung auftreten: z.B. Illegale Instruktion, Division durch 0, Adressierungsfehler, Privilegverletzung
- Interrupts



68020 Register

Datenregister:

- alle Datenoperationen können unterschiedslos auf allen Datenregistern ausgef. werden
- unterstützen Operationen auf allen Datentypen des 68020
- können in einem indizierten Adressierungsmodus den Index enthalten

Adreßregister:

- werden als Basisregister bei der Adressierung verwendet
- werden als 32-Bit Einheiten behandelt und können nur Operationen auf 32-Bit Datentypen ausführen
- arithmetisch/logische Operationen auf Adreßreg. modifizieren nicht die Bedingungsflags im CCR
- Adreßregister #7 dient als Stack Pointer für Unterprogrammaufrufe ! (alle anderen Adreßregister können als SP verwendet werden)
- das Supervisor-Flag (S) und das Master/Interrupt-Flag (M) im Status Register entscheiden, welcher Stack-Pointer tatsächlich genutzt wird.

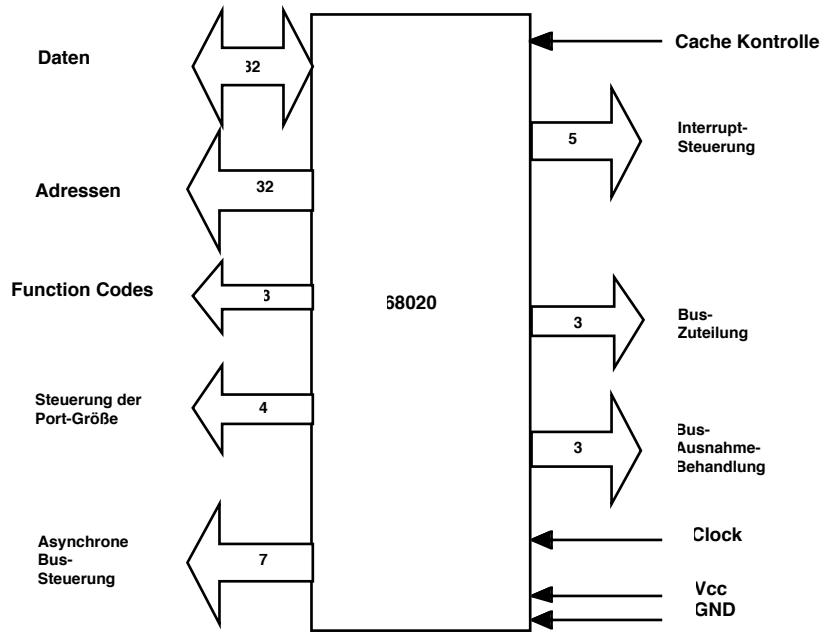
Bussysteme

- synchron - asynchron
- Daten und Adressen auf einem Bus (multiplexed) - getrennte Busse (non-multiplexed)
- Burst-Modus - ein Buszyklus/ein Datum
- Arbitrierter Bus (mehrere Bus-Master) - einzelner fester Bus-Master
- dynamische (automatische) Busanpassung - explizite (programmierte) Busanpassung
- automatische Fehlerbehandlung - programmierte Fehlerbehandlung

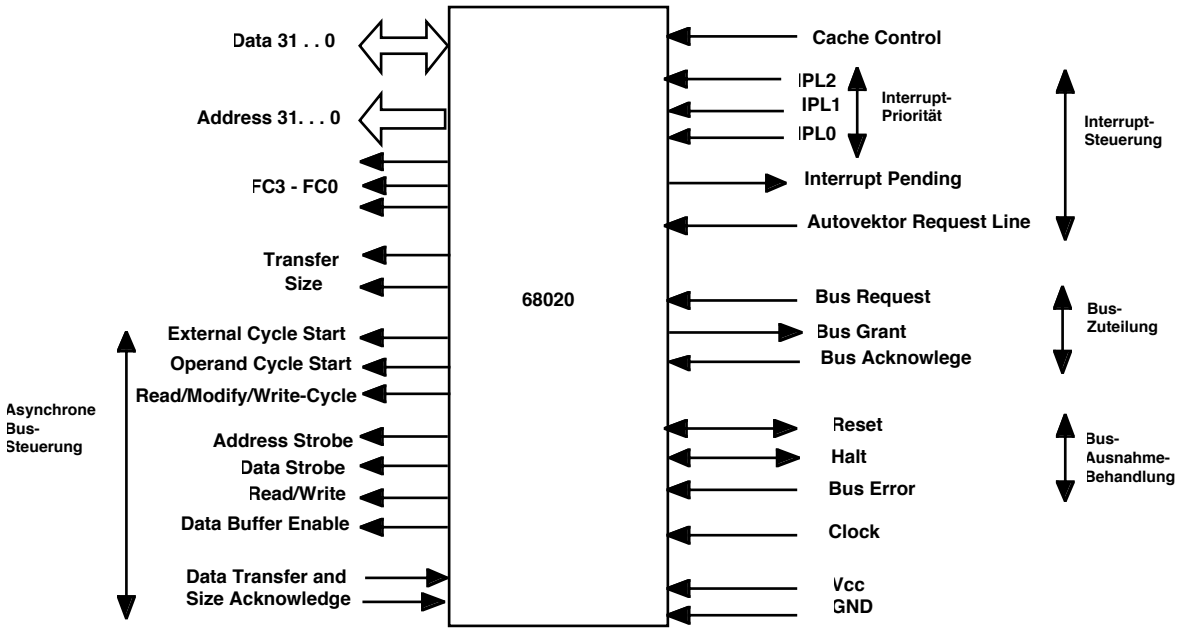
68020 Bussystem

- Parallele 32-Bit-Busse für Daten und Adressen
- getrennte Busse, kein Busmultiplexing
- Asynchrones Busprotokoll
- Master/Slave Konfiguration
- Bus Arbitration, mehrere Bus-Master
- Dynamische Busanpassung
- Unterstützung von nicht auf Wortgrenzen ausgerichteten Daten
- Flexible Behandlung von Busfehlern
- Automatische Wiederholung von Buszyklen (Retry)

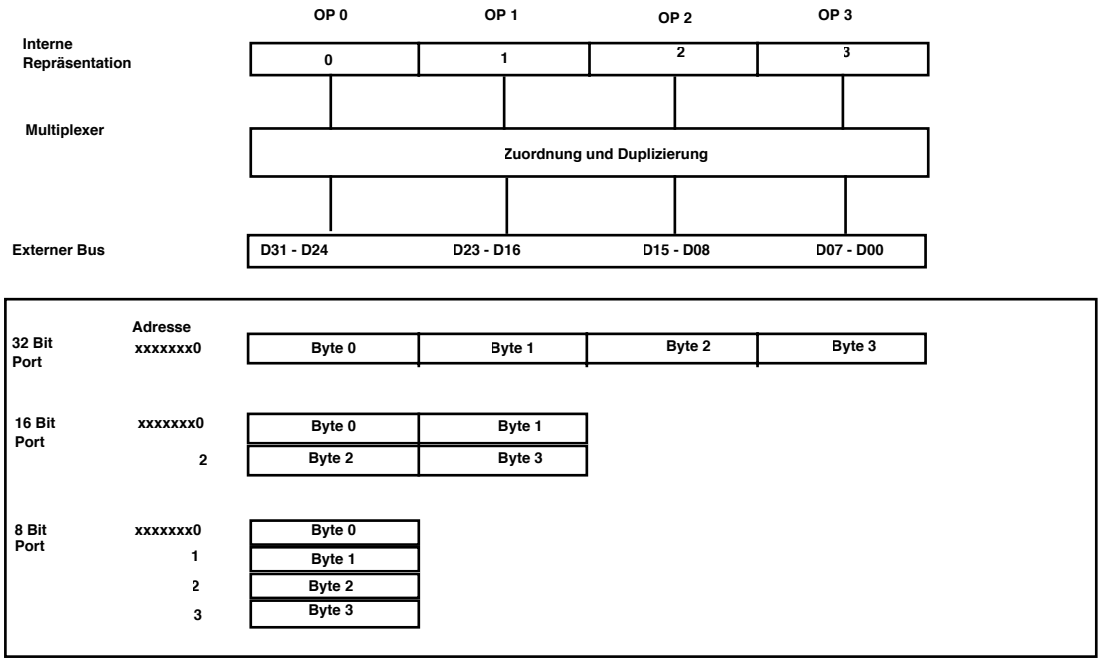
68020 Steuersignale



68020 Steuersignale (Detail)



Dynamische Busanpassung



Dynamische Busanpassung

68020 signalisiert über die Ausgänge SIZ0 und SIZ1 die Port-Größe für den durchzuführenden Datentransfer

Codierung der "Size"-Ausgänge

SIZ1	SIZ0	übertragene Größe
0	0	Langwort (32 Bit)
0	1	Byte
1	0	Wort (16 Bit)
1	1	3 - Byte

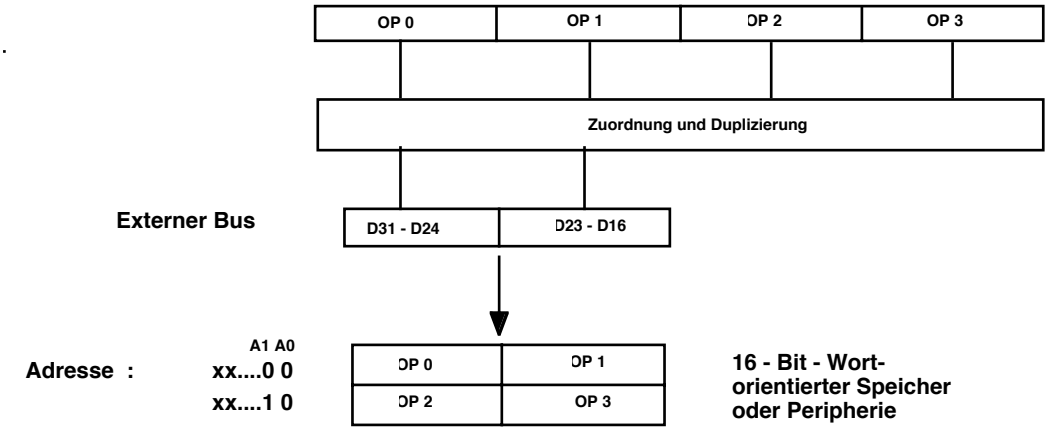
Peripheres Gerät signalisiert über die Eingänge DSACK0 und DSACK1 die vorhandene Port-Größe

Codierung der DSACK- Eingänge

DSACK1	DSACK0	Resultat
H	H	Einfügen von Wartezyklen
H	L	vollst. Zyklus - DBPS : 8 Bit
L	H	vollst. Zyklus - DBPS : 16 Bit
L	L	vollst. Zyklus - DBPS : 32 Bit

Dynamische Busanpassung

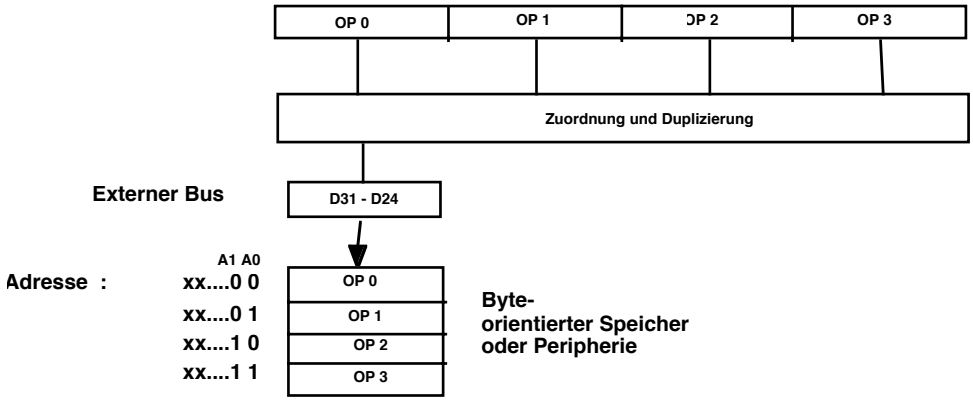
Beispiel - Transfer eines Langwortes über einen 16-Bit-Bus:



68020				Periph. Kontrolle		
SIZ1	SIZ0	A1	A0	DSACK1	DSACK0	
0	0	0	0	L	H	1.Zyklus : OP0, OP1 werden Übertragen
1	0	1	0	L	H	2.Zyklus : OP2, OP3 werden Übertragen

Dynamische Busanpassung

Beispiel - Transfer eines Langwortes über einen 8-Bit-Bus:

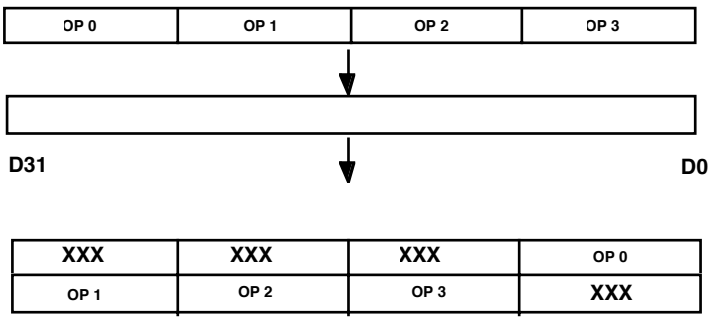


68020				Periph. Kontrolle		
SIZ1	SIZ0	A1	A0	DSACK1	DSACK0	
0	0	0	0	H	L	1.Zyklus : OP0 wird übertragen
1	1	0	1	H	L	2.Zyklus : OP1 wird übertragen
1	0	1	0	H	L	3.Zyklus : OP2 wird übertragen
0	1	1	1	H	L	4.Zyklus : OP3 wird übertragen

Dynamische Busanpassung

Beispiel - Nicht auf Wortgrenzen angepaßte Daten (misaligned Data)

Instruktionen müssen immer an Langwortgrenzen angepaßt sein. In der Befehlsholphase wird immer ein Langwort von einer geraden Adresse mit A0, A1 : 00 gelesen.



Langwort-orientierter Speicher

68020					Periph. Kontrolle		
SIZ1	SIZ0	A2	A1	A0	DSACK1	DSACK0	
0	0	0	1	1	L	L	1.Zyklus : OP0 wird übertragen
1	1	1	0	0	L	L	2.Zyklus : OP1-3 werden übertragen

68020 Lese-Zyklus

Bus Master — **Slave**

Adressierungsphase:

1. R/W ← Read
2. Funktionscode anlegen
3. Adresse auf A31-A0 legen
4. SIZ1, SIZ0 anlegen
5. Anzeigen, daß ein Buszyklus gestartet wird durch die Signale ECS und OCS. Vorläufige Anzeige eines Buszyklus. Wenn der 68020 einen Befehlszyklus startet, der Befehl aber schon im I-Cache ist, wird der Buszyklus abgebrochen.
6. Adreßstrobe aktivieren. Diese Signal bestätigt, daß ein Buszyklus durchgeführt wird.
7. Datenstrobe aktivieren
8. Data Buffer Enable aktivieren

Datenübergabe:

1. Adresse dekodieren
2. Daten auf Datenbus legen
3. DSACK0 und DSACK1 aktivieren

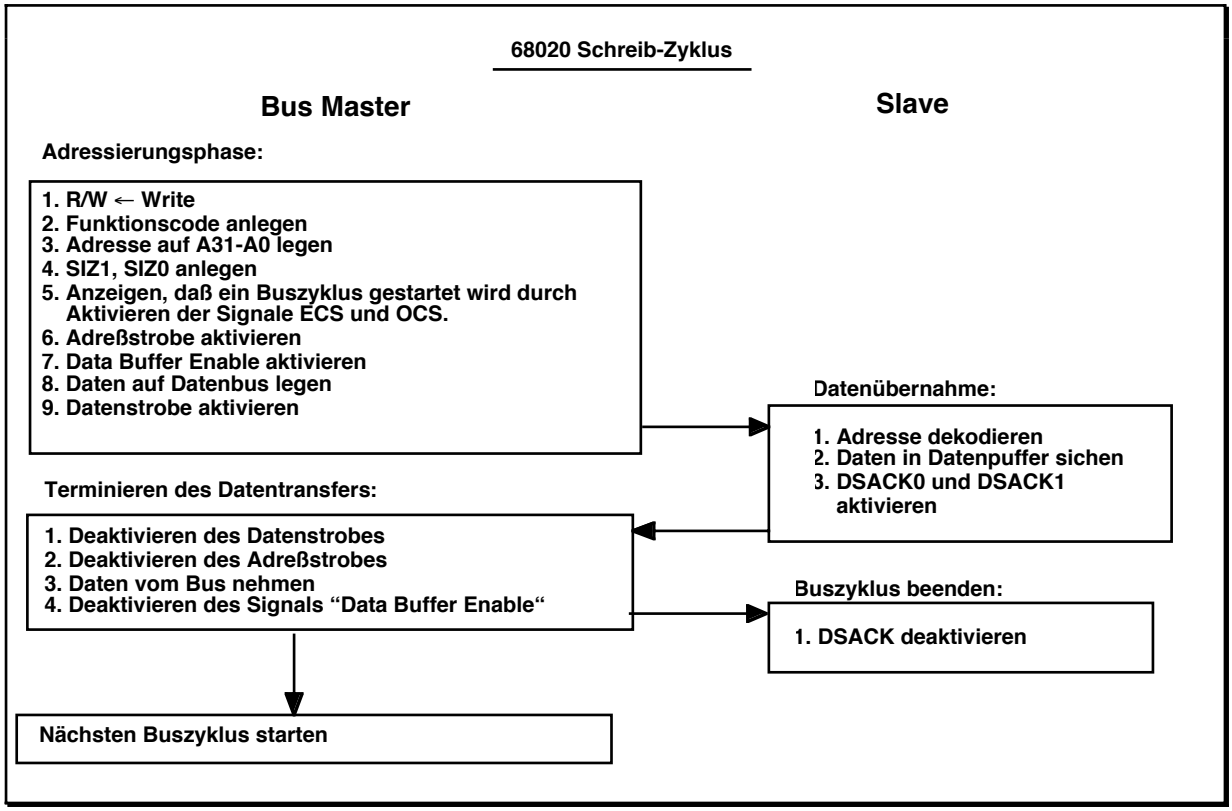
Datenübernahme:

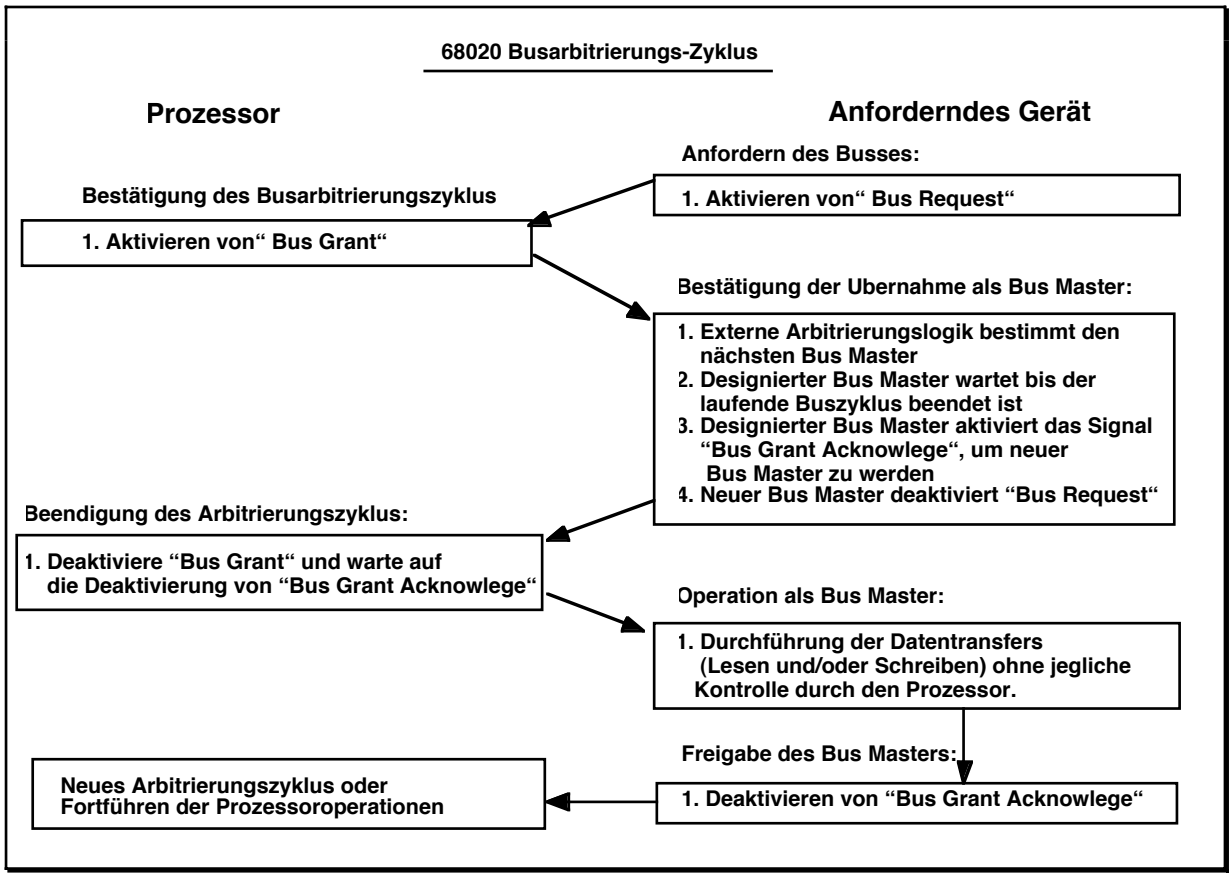
1. Daten in Datenpuffer speichern
2. Deaktivieren des Datenstrobes
3. Deaktivieren des Adreßstrobes
4. Deaktivieren des Signals “Data Buffer Enable“

Buszyklus beenden:

1. Daten vom Bus nehmen
2. DSACK deaktivieren

Nächsten Buszyklus starten


68020 Schreib-Zyklus
Bus Master
Slave
Adressierungsphase:
1. R/W ← Write
2. Funktionscode anlegen
3. Adresse auf A31-A0 legen
4. SIZ1, SIZ0 anlegen
5. Anzeigen, daß ein Buszyklus gestartet wird durch Aktivieren der Signale ECS und OCS.
6. Adreßstrobe aktivieren
7. Data Buffer Enable aktivieren
8. Daten auf Datenbus legen
9. Datenstrobe aktivieren
Datenübernahme:
1. Adresse dekodieren
2. Daten in Datenpuffer sichen
3. DSACK0 und DSACK1 aktivieren
Terminieren des Datentransfers:
1. Deaktivieren des Datenstrobes
2. Deaktivieren des Adreßstrobes
3. Daten vom Bus nehmen
4. Deaktivieren des Signals "Data Buffer Enable"
Buszyklus beenden:
1. DSACK deaktivieren
Nächsten Buszyklus starten


68020 Busarbitrierungs-Zyklus
Prozessor
Anforderndes Gerät
Anfordern des Busses:
1. Aktivieren von" Bus Request"
Bestätigung des Busarbitrierungszyklus
1. Aktivieren von" Bus Grant"
Bestätigung der Ubernahme als Bus Master:
1. Externe Arbitrierungslogik bestimmt den nächsten Bus Master
2. Designierter Bus Master wartet bis der laufende Buszyklus beendet ist
3. Designierter Bus Master aktiviert das Signal "Bus Grant Acknowlege", um neuer Bus Master zu werden
4. Neuer Bus Master deaktiviert "Bus Request"
Beendigung des Arbitrierungszyklus:
1. Deaktiviere "Bus Grant" und warte auf die Deaktivierung von "Bus Grant Acknowlege"
Operation als Bus Master:
1. Durchführung der Datentransfers (Lesen und/oder Schreiben) ohne jegliche Kontrolle durch den Prozessor.
Freigabe des Bus Masters:
1. Deaktivieren von "Bus Grant Acknowlege"
Neues Arbitrierungszyklus oder Fortführen der Prozessoroperationen

12 Coprozessoren

Coprozessoren: Funktionale Partitionierung in Hardware-Spezialeinheiten

Ziel: Realisierung von Spezialfunktionen wie: Fließkomma-Arithmetik, Vektoroperationen, Speicherverwaltung, Graphikfunktionen, HLL-Interpretation, etc.

Alternativen: Software-Lösung, Vertikale Verlagerung in die Mikroprogrammebene

Vorteile von Coprozessoren gegenüber vertikaler Verlagerung:

- **Durch funktionale Partitionierung wird die Komplexität des Entwurfs vermindert. Die Komplexität eines umfangreichen Mikroprogramms stellt schon bei konventionellen Prozessoren ein größeres Problem dar.**
- **Durch einen problemangepaßten Coprozessor können spezielle Aufgaben effizienter gelöst werden als durch einen mikroprogrammierbaren Universalprozessor.**
- **Coprozessor und CPU können (im Prinzip) nebenläufig arbeiten.**
- **Höhere Flexibilität, da durch der Partitionierung eine Isolation der Spezial-Funktionen vom Instruktionssatz der CPU erreicht wird. Dadurch können die Spezialfunktionen leichter und ohne Nebenwirkungen auf die CPU geändert werden.**
- **Der Instruktionssatz einer Standard-CPU wird durch einen Coprozessor so erweitert, daß eine volle Kompatibilität mit Standardsoftware erhalten bleibt. Spezielle Coprozessorbefehle können meist leicht emuliert werden.**

Kriterien zur Klassifizierung von Coprozessoren:

Ebene der Coprozessor-Funktionen:

- **Instruktions-Coprozessoren (Beisp. FPU)**
- **Funktions-Coprozessoren (Beisp. Graphik-Coproz., Interpr. von Progr. Sprachen)**
- **Programm-Coprozessoren (eigene Programmsteuerung, Beisp. IVORY, COLIBRI)**

Steuerung des Coprozessors:

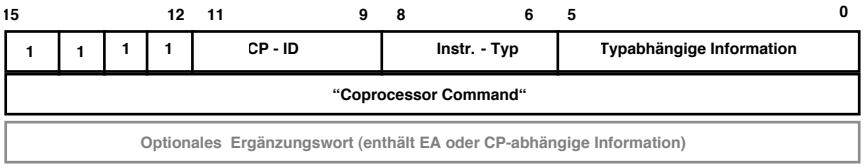
- **Ein Instruktionsstrom für CPU und Coprozessor (Transparente Erweiterung des Instruktionssatzes der CPU)**
- **Getrennte Instruktionsströme für CPU und Coprozessor (Autonome Coprozessoren)**

Anbindung an die CPU

- **Eigene Befehlsdekodierung (Instruction Tracker)**
- **Befehlsdekodierung wird von der CPU durchgeführt**
- **synchrones Protokoll mit der CPU**
- **asynchrones Protokoll mit der CPU**
- **CPU führt alle Speicherzugriffe durch**
- **Coprozessor kann selbst Speicherzugriffe durchführen(DMA-Coproz.)**

Parallelität : CPU und Coprozessoraktivität können nebenläufig arbeiten

Coprozessor Instruktion



CP-Operations-Wort

CP-Kommando-Wort

Bit 15 - 12 : F-Line Code. Dieser Code repräsentiert eine Coprozessor-Instruktion
Bit 11 - 9 : Croprozessor Indentifizierung. 8 Coprozessoren können selektiert werden.
Bit 8 - 6 : Typ der Coprozessor Instruktion: General, Branch (2), Conditional Instr. (2), CP - Trap, Save, Restore
Bit 5 - 0 : Information abhängig vom Instruktionstyp

Coprozessorschnittstelle des 68020

Registersatz zur Realisierung der Coprozessor-Schnittstelle

erforderlich

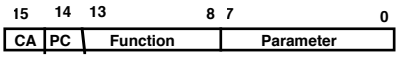
Response	Control
Save	Restore
Operation	Command
Reserved	Condition
Operand	

optional

Register Select	Reserved
Instruction Address	
Operand Address	

Coprozessor Interface Register (CIR)

Response Register:



Über dieses Register fordert der Coprozessor eine Funktion der CPU an. Die entsprechende Funktion wird im Feld "Function" spezifiziert. Evtl. Parameter werden im Feld "Parameter" abgelegt. Das Response Register ist für die CPU read-only. Das CA-Flag gibt an, daß die CPU nach Ausführung der Funktion das Response Register erneut lesen soll. Das PC Flag gibt an, daß der PC der CPU in das Register "Instruction Address" geschrieben wird, bevor die angeforderte Funktion ausgeführt wird

Control Register:

Der Coprozessor kann die CPU unterbrechen und eine Ausnahmebehandlungsroutine initiieren. Über das Control Register wird dieser Vorgang gesteuert. Die CPU setzt nach einem "Coprocessor Exception Request" die Flags XA oder AB.

- Setzen von XA : die CPU bestätigt einen "Coprocessor Exception Request"
- Setzen von AB : die CPU bricht eine Coprozessor-Instruktion ab.

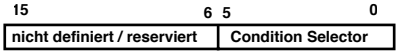
Operation Register:

Wenn der Coprozessor das Operationswort (erstes Wort der CP-Instruktion) anfordert, wird es von der CPU in das Operations Register geschrieben.

Command Register:

Die CPU initialisiert eine allgemeine Coprozessor Operation indem sie das "Command Word" (zweites Wort der CP-Instruktion) in das Command Register schreibt.

Condition Register:



Das Condition Register wird von der CPU mit einem Condition Code beschrieben, der vom Coprozessor für bedingte Instruktionen und Sprünge genutzt wird.

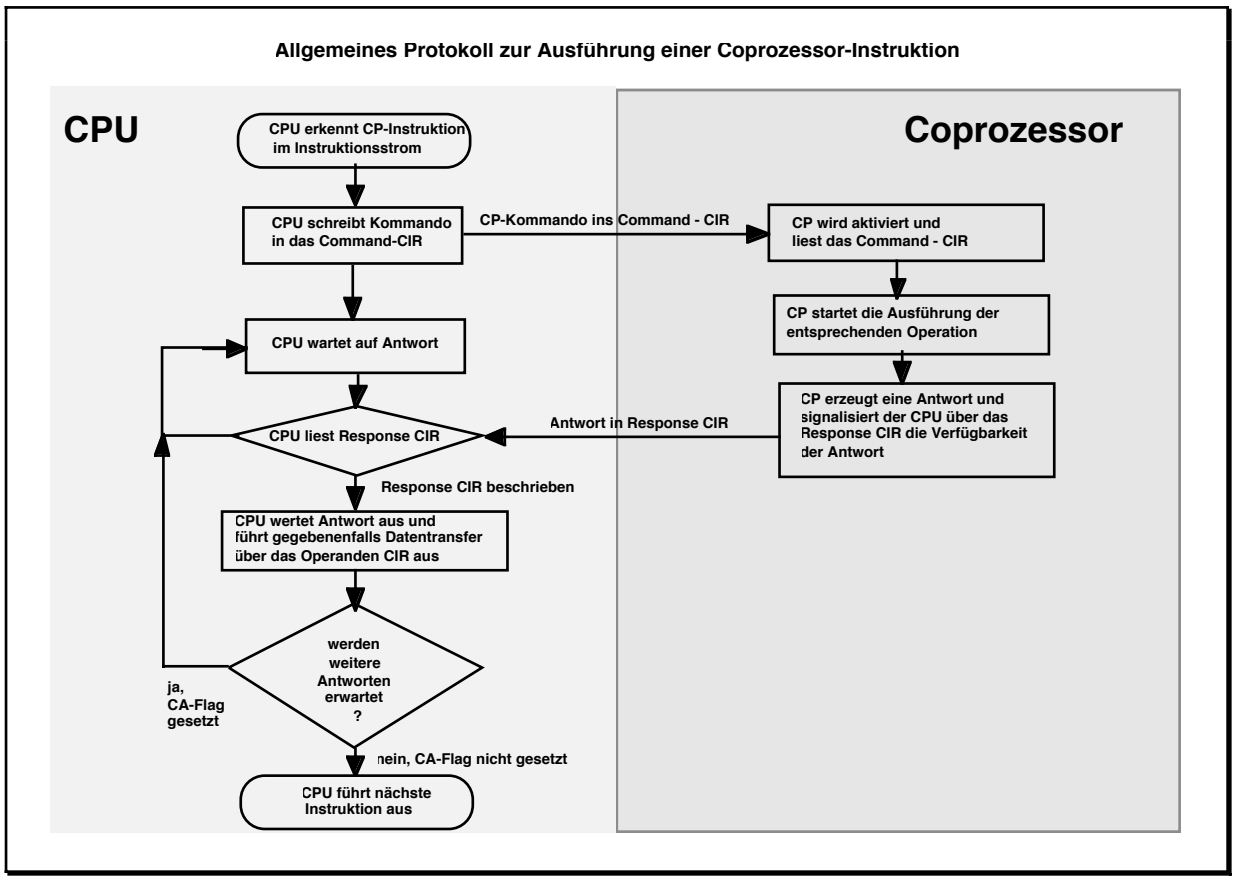
Operanden Register:

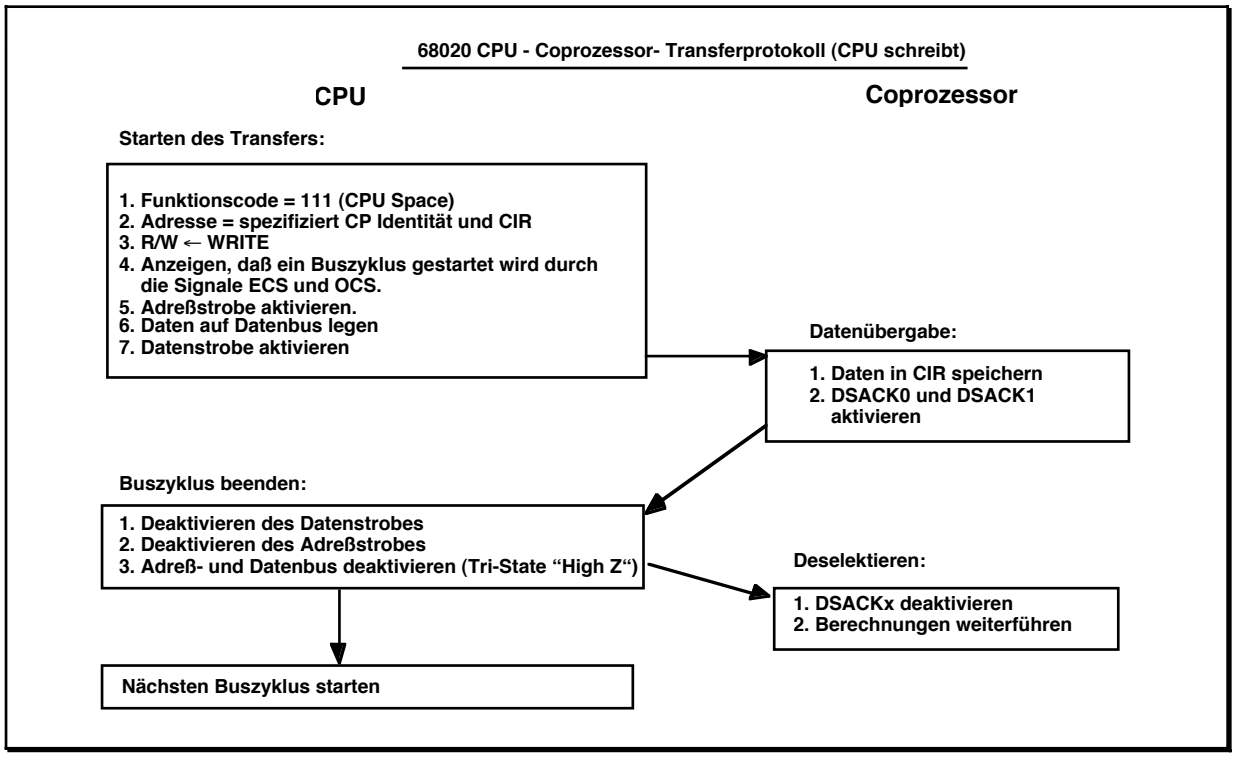
Das Operanden Register dient dem Datentransfer zwischen CPU und Coprozessor.

Basisbefehle des Coprozessors

Basisbefehle (Primitive Commands) des Coprozessors sind Befehle an die CPU, eine Funktion für den Coprozessor auszuführen. Die Coprozessor Basisbefehle werden zur Übermittlung an die CPU vom Coprozessor ins RESPONSE Reg. geschrieben.

- **Prozessor Synchronisation:**
  - Busy from previous instruction
  - Busy with current instruction
  - Proceed with next instruction (trace disabled)
  - Proceed with next instruction (trace enabled)
  - Proceed with execution, condition TRUE / FALSE
- **Instruction Manipulation**
  - Transfer Operation Word
  - Transfer words from instruction stream
- **Exception Handling**
  - Take privilege violation if S-Bit (Supervisor) is not set
  - Take pre-instruction exception
  - Take mid-instruction exception
  - Take post-instruction exception
- **General Operand Transfer**
  - Evaluate and pass <ea>
  - Evaluate <ea> and transfer data
  - Write to previous evaluated <ea>
  - Take address and transfer data
  - Transfer to / from top of stack
- **Register Transfer**
  - Transfer CPU Register
  - Transfer CPU Control Register
  - Transfer Multiple Registers
  - Transfer Multiple Coprocessor Registers
  - Transfer CPU SR and / or PC


Allgemeines Protokoll zur Ausführung einer Coprozessor-Instruktion
CPU
Coprozessor
CPU erkennt CP-Instruktion im Instruktionsstrom
CPU schreibt Kommando in das Command-CIR
CP-Kommando ins Command - CIR
CP wird aktiviert und liest das Command - CIR
CP startet die Ausführung der entsprechenden Operation
CPU wartet auf Antwort
CP erzeugt eine Antwort und signalisiert der CPU über das Response CIR die Verfügbarkeit der Antwort
Antwort in Response CIR
CPU liest Response CIR
Response CIR beschrieben
CPU wertet Antwort aus und führt gegebenenfalls Datentransfer über das Operanden CIR aus
werden weitere Antworten erwartet ?
ja, CA-Flag gesetzt
nein, CA-Flag nicht gesetzt
CPU führt nächste Instruktion aus

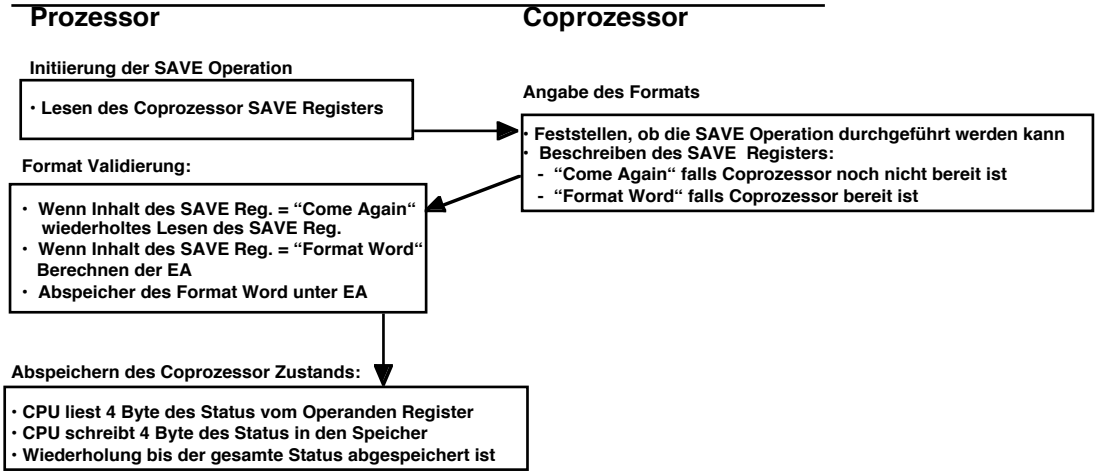

68020 CPU - Coprozessor- Transferprotokoll (CPU schreibt)
CPU
Coprozessor
Starten des Transfers:
1. Funktionscode = 111 (CPU Space)
2. Adresse = spezifiziert CP Identität und CIR
3. R/W ← WRITE
4. Anzeigen, daß ein Buszyklus gestartet wird durch die Signale ECS und OCS.
5. Adreßstrobe aktivieren.
6. Daten auf Datenbus legen
7. Datenstrobe aktivieren
Datenübergabe:
1. Daten in CIR speichern
2. DSACK0 und DSACK1 aktivieren
Buszyklus beenden:
1. Deaktivieren des Datenstrobes
2. Deaktivieren des Adreßstrobes
3. Adreß- und Datenbus deaktivieren (Tri-State “High Z“)
Deselektieren:
1. DSACKx deaktivieren
2. Berechnungen weiterführen
Nächsten Buszyklus starten

SAVE Register und RESTORE Register

Das SAVE Register und RESTORE Register werden für die Coprozessor-Instruktionen SAVE und RESTORE benutzt.
SAVE und RESTORE sichern den internen Zustand des Coprozessors (der im sogenannten "Coprocessor Internal State Frame" gehalten wird) im Speicher, bzw. stellen einen solchen gesicherten Zustand aus dem Speicher wieder her.

- SAVE:
  Bei der SAVE-Operation legt der Coprozessor das sogenannte "Format Word", das eine Beschreibung seines internen Zustands repräsentiert in das SAVE Register. Die CPU liest das SAVE Register und legt die im folgenden vom Coprozessor im Operanden Register zur Verfügung gestellten Daten in der entsprechenden Datenstruktur im Speicher ab.
- RESTORE:
  Bei der RESTORE-Operation legt die CPU das "Format Word" in das RESTORE Register. Der Coprozessor liest das RESTORE Register und verifiziert die angegebene Datenstruktur, die wiederhergestellt werden soll. Stimmt sie mit seinem internen Format überein, kann die Übertragung des gesicherten Coprozessor Zustands über das Operanden Register durchgeführt werden.

Protokoll zur Sicherung des Coprozessor Zustands (SAVE):



13 RISC-Prozessoren

Was ist ein Reduced Instruction Set Computer (RISC*) ?

* Der Begriff RISC wurde von Carlo Sequin (UCB) geprägt)

Motivationen für die RISC-Entwicklung:

IBM 801:	**Single-Cycle Instruktionen**
Berkeley RISC:	**Begrenzte Chipfläche**
Stanford:	**Überlappende Ausführung mit einfacher Kontrolle**

Gemeinsamer Ausgangspunkt für alle diese Forschungsprojekte war eine Analyse der Nutzung von Maschinenbefehlen und Adressierungsarten durch den Compiler. Eine Grundannahme für RISCs ist, daß die auf ihnen ablaufenden Anwendungsprogramme in einer Hochsprache (HLL) programmiert werden. Die Compiler-Technologie war und ist eine der Schlüsselelemente für die RISC-Technologie. Der Begriff RISC stellt tatsächlich eine "Entwurfs-Philosophie" dar, in der das Ziel höchste Leistung ist, die im Zusammenspiel von Hardware und einem optimierenden Compiler erreicht wird.

CISC-Architekturen versuchen, einen leistungsfähigen Maschinenbefehlssatz zu realisieren, bzw. die "Semantische Lücke" zwischen einer Hochsprache und der Architektur durch einen an die Hochsprache angepaßten Instruktionssatz zu schließen, so daß der Compiler nur noch einen minimalen Übersetzungsaufwand hat.

Im Gegensatz dazu wird in der RISC Technologie der Instruktionssatz in Hinblick auf Einfachheit und Regularität entworfen, so daß die Verwendung der Instruktionen durch den Compiler einfach und überschaubar ist.

Voraussetzungen: Fortschritte in der Speichertechnologie, Compilerbau

Was ist ein Reduced Instruction Set Computer (RISC) ?

Anfänge der RISC - Technologie:

1979	**IBM 801 Minicomputer**	**G.Mar Radin**	G. Mar Radin: The 801 Minicomputer *Proceedings of the International Symp. on Architectural Support for Programming Languages and Operating Systems,* Palo Alto, CA, 1982
	↓		
	IBM RT PC, POWER, Power PC		
1981	**Berkeley RISC**	**Carlo Sequin , David A. Patterson**	
	↓		D. A. Patterson: Reduced Instruction Set Computers *Communications of the ACM,* 28(1), 1985
	SPARC		
1982	**Stanford RISC**	**John Hennessy**	John Hennessy: VLSI Processor Architecture *IEEE Transactions on Computers,* C-33(11), 1984
	↓		
	MIPS		

Was ist ein Reduced Instruction Set Computer (RISC) ?

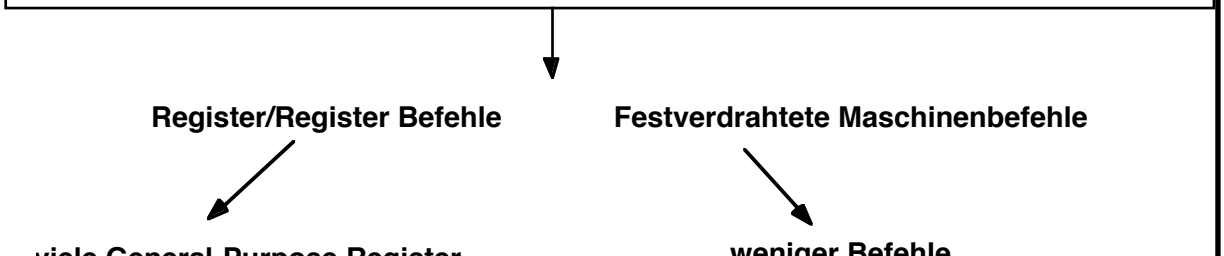
D. Tabak: *RISC-Architecture* , John Wiley & Sons, 1987

Anzahl der Instruktionen	≤ 50	Restriktionen
Anzahl der Adr. Modi	≤ 4	
Anzahl der Befehlsformate	≤ 4	
LOAD/STORE Architektur		Programmiermodell
Anzahl der Register	≥ 32	Reg.-Reg. Architektur
Single Cycle Instruktionen		Implementierungs-
Festverdrahtete Maschinenbefehle		Randbedingungen
HLL / Optimierende Compiler		Programmierung/ Progr. Umgebung

D. Tabak: min. 5 der Bedingungen müssen erfüllt sein

Was ist ein Reduced Instruction Set Computer (RISC) ?

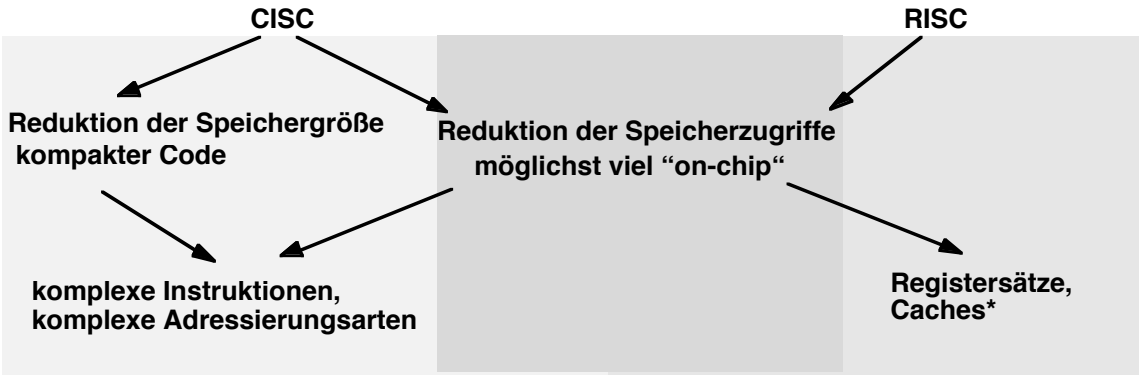
Zentrale Problemstellung: Wie kann man eine Architektur mit dem Ziel entwerfen, Instruktionen in einem Maschinenzyklus auszuführen.



Eine wesentliche Voraussetzung für Single Cycle Befehlsausführung ist Pipelining

Was ist ein Reduced Instruction Set Computer (RISC) ?

Lösungsansätze zur Verringerung der Speicherzugriffe

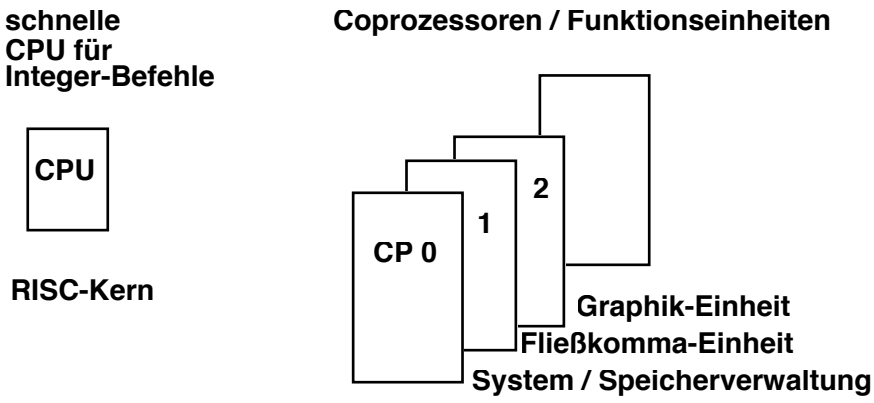


* insbesondere lösen Registersätze nicht das Problem, daß eine komplexe Instruktion durch eine Folge von einfacheren Instruktionen ersetzt werden muß, d.h. in der Befehlshol-phase mehr Speicherzugriffe notwendig sind. Dieses Problem wird durch ein Instruktions-Cache gelöst.

Was ist ein Reduced Instruction Set Computer (RISC) ?

Realisierung komplexerer Instruktionen:

Beisp.: Fließkomma-Instruktionen, Systemverwaltung, Graphik



Die RISC-Technologie erfordert funktionale Dekomposition auf der Architekturebene

Was ist ein Reduced Instruction Set Computer (RISC) ?

Folge der Realisierung einfacher Instruktionen:

Cons:

- **Anzahl der Instruktionen wächst**
- **Mehr Speicherplatz für Programme**
- **Größere Speicherbandbreite für den (Befehls-) Speicher**

Pros:

- **Einfachheit**
- **Höhere Taktraten**
- **Ausnutzung von Pipelinetechniken eher möglich**

- **Speicherbandbreite kann durch entsprechende Cachingtechniken erfolgreich erhöht werden**
- **Höhere Speicherbandbreite ist nur für die Befehle notwendig. Da der Befehlsstrom im wesentlichen sequentiell ist (Lokalität), lassen sich Caching-Techniken besonders erfolgreich einsetzen.**

Was ist ein Reduced Instruction Set Computer (RISC) ?

Maße für Ausführungszeiten in einem Rechner:

CPU Time

$$CT = TZ \cdot TD$$

CT: CPU - Zeit, Ausführungszeit für ein Programm
TZ: Gesamtzahl der Taktzyklen in einem Programm
TD: Dauer eines Taktzyklus

Clock Cycles / Instruction

$$CPI = TZ / IC$$

CPI: Zyklen pro Instruktion
TZ: Anzahl der Taktzyklen in einem Programm
IC: Anzahl der Befehle in einem Programm

$$CT = IC \cdot CPI \cdot D$$

CISC Ansätze versuchen die Gesamtzahl der Instruktionen (IC) zu verringern !

RISC Ansätze versuchen die Zyklen/Instruktion (CPI) zu verringern !

Was ist ein Reduced Instruction Set Computer (RISC) ?

Folge der Realisierung einfacher Instruktionen:

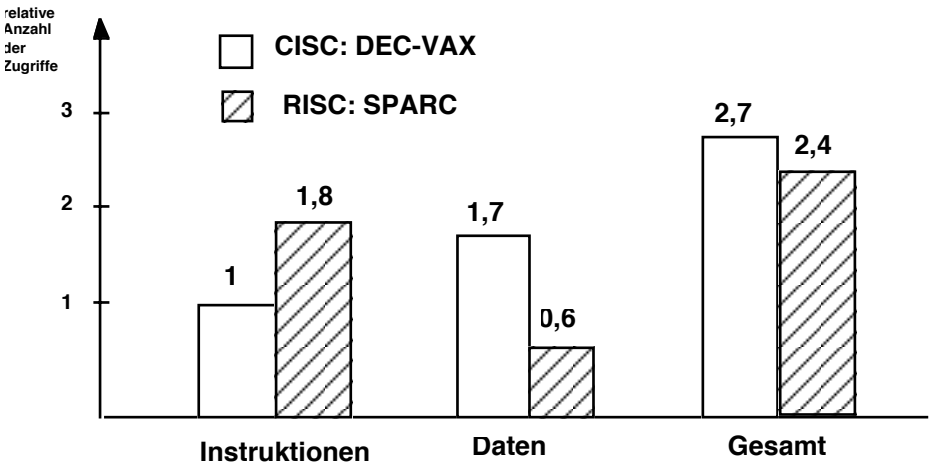
Vergleich 68020 (25 Mhz) und einer frühen Version des SPARC (16Mhz)

	68020	SPARC	Kommentar
IC (Anz.d.Instr.)	1.0	1,25	25% mehr Instruktionen
TZ (Taktzyklus)	40 ns	60 ns	50% längerer Takt
CPI	5,0 - 7,0	1,3 - 1,7	!!
CT (CPU-Zeit)	2	1	doppelt so "schnell"

Was ist ein Reduced Instruction Set Computer (RISC) ?

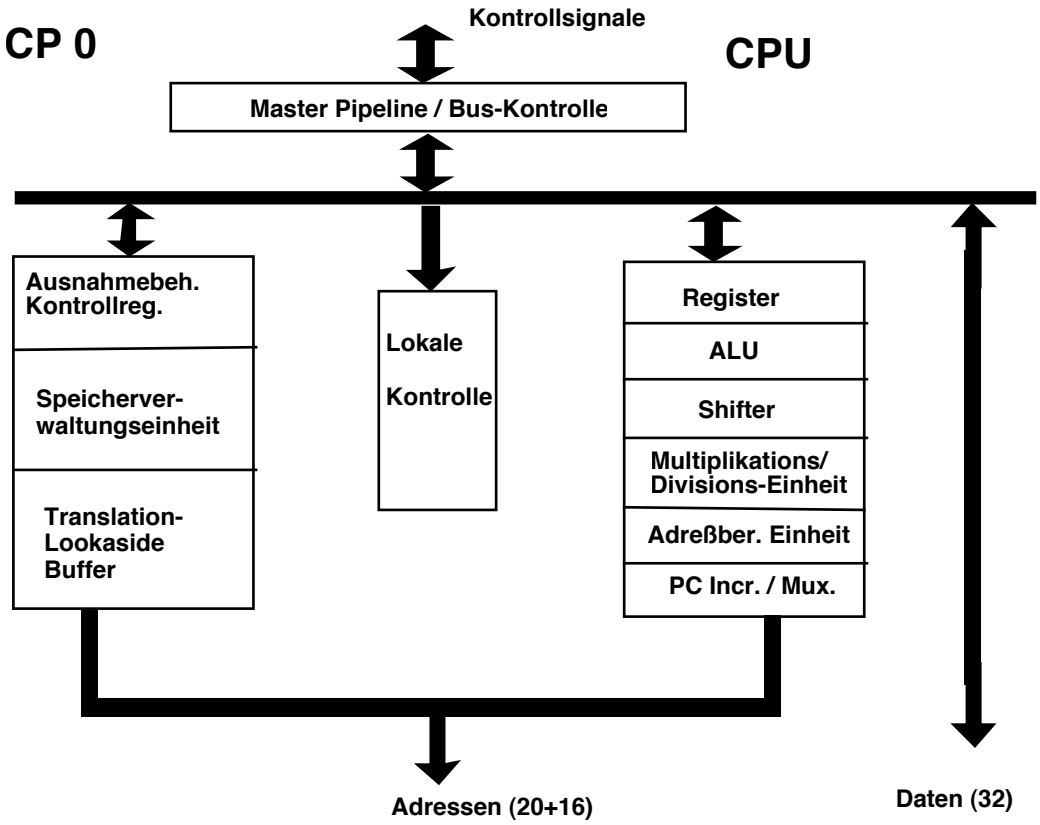
Folge der Realisierung einfacher Instruktionen:

Vergleich der Speichernutzung (Memory Traffic) bei einem CISC und einem RISC

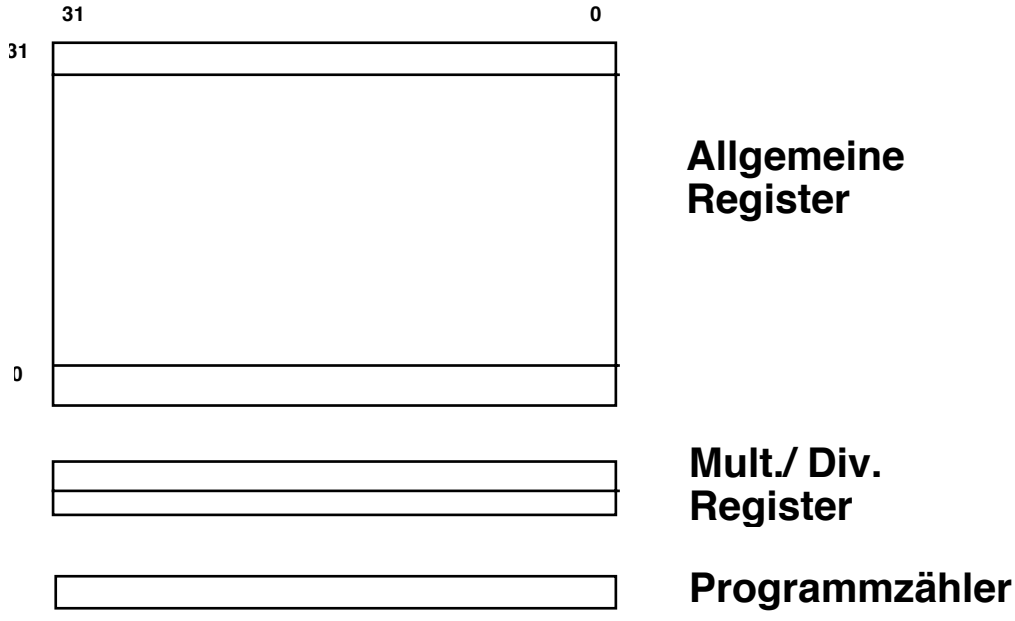


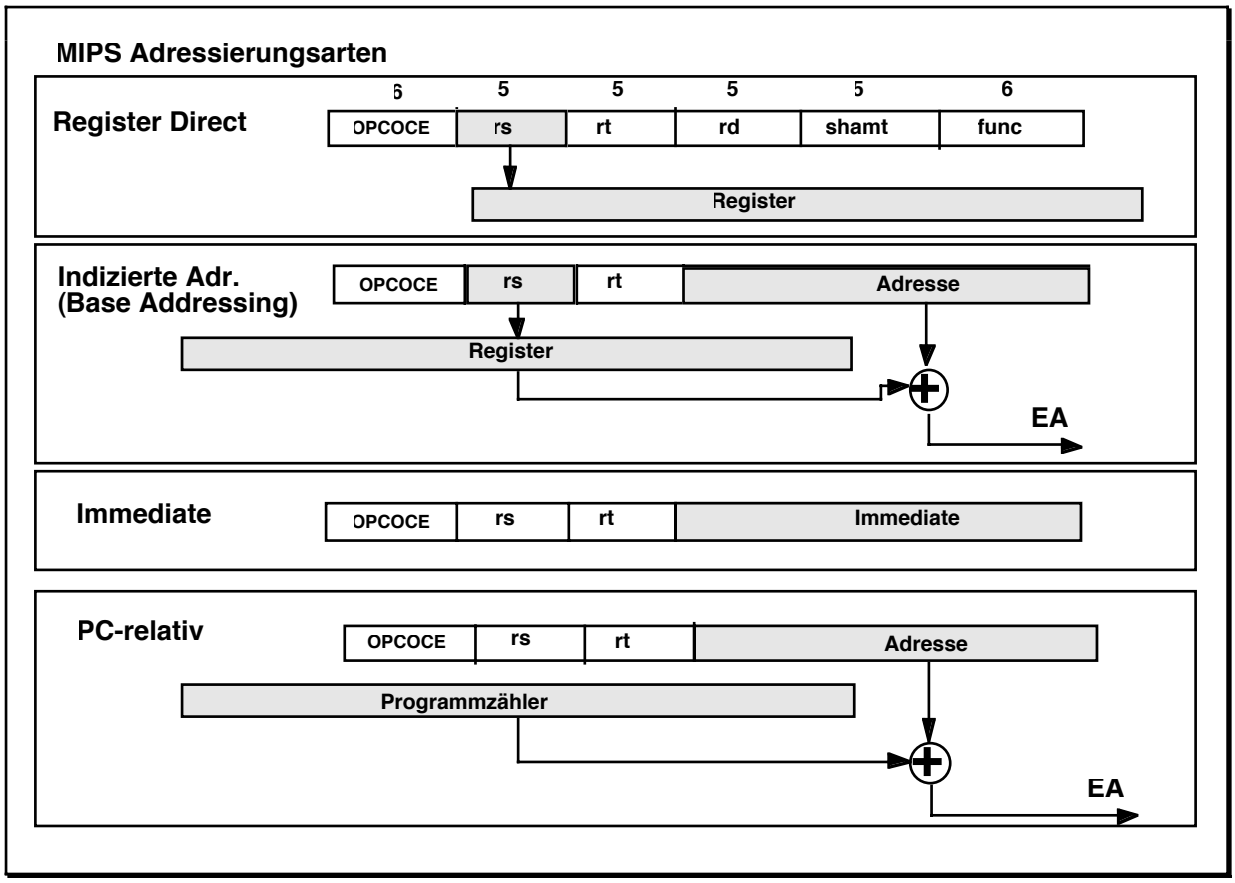
Alle Zugriffe sind auf die Anzahl der Zugriffe in der Befehlsholphase der VAX normiert

MIPS Architektur



MIPS Programmiermodell





13.1 Pipelining

Pipelining (Fließbandverarbeitung):

Pipelining has become the accepted implementation method for computers of virtually every class, while the serial one-at-a-time execution model is fading into history - existing only in beginning computer architecture textbooks.

Today's computer architect thinks in terms of architetcures that map well onto a pipeline, and implementors begin a design by determining the overall pipeline structure.

Shlomo Weiss, James E. Smith : POWER and PowerPC, 1994

Möglichkeiten der Parallelarbeit im Autobau

Jeder Arbeiter baut ein vollständiges Auto	Organisation von Gruppen, die parallel vollständige Autos bauen (Volvo Modell)	Organisation eines Fließbands, an dem in jedem Abschnitt eine spezielle Arbeit ausgeführt wird. An allen Abschnitten wird gleichzeitig gearbeitet	Organisations-Struktur
Hoher Grad an unterschiedlichen Fertigkeiten erforderlich geringe Spezialisierung	Mittlerer Grad an Fertigkeiten erforderlich mittlere Spezialisierung	Geringer Grad an unterschiedlichen Fähigkeiten erforderlich hohe Spezialisierung	Spezialisierungs-Grad
Robust gegen Ausfälle und unterschiedlich Arbeitsgeschwindigkeiten lokale Koordination/ globale Versorgung	Robust gegen Ausfälle und unterschiedlich Arbeitsgeschwindigkeiten lokale Koordination/ globale Versorgung	Empfindlich gegen Ausfälle und untersch. Arbeitsgeschw. globale Koordination/ lokale Versorgung	Robustheit und Aufwand zur Koordinierung der Arbeit
Erhöhung der Arbeitsleistung des Einzelnen	Erhöhung der Arbeitsleistung des Einzelnen Erhöhung der Anzahl der Mitarbeiter	Erhöhung der Arbeitsleistung des Einzelnen	Verminderung der Verzögerung Erhöhung der Arbeitsleistung/ Zeit
Erhöhung der Anzahl der Mitarbeiter	Erhöhung der Anzahl der Teams	Erhöhung der Anzahl der Fließbänder Erhöhung der Anzahl der Mitarbeiter durch die Einführung neuer Fließbandabschnitte	Erhöhung des Durchsatzes

Grundeigenschaften einer Pipeline:

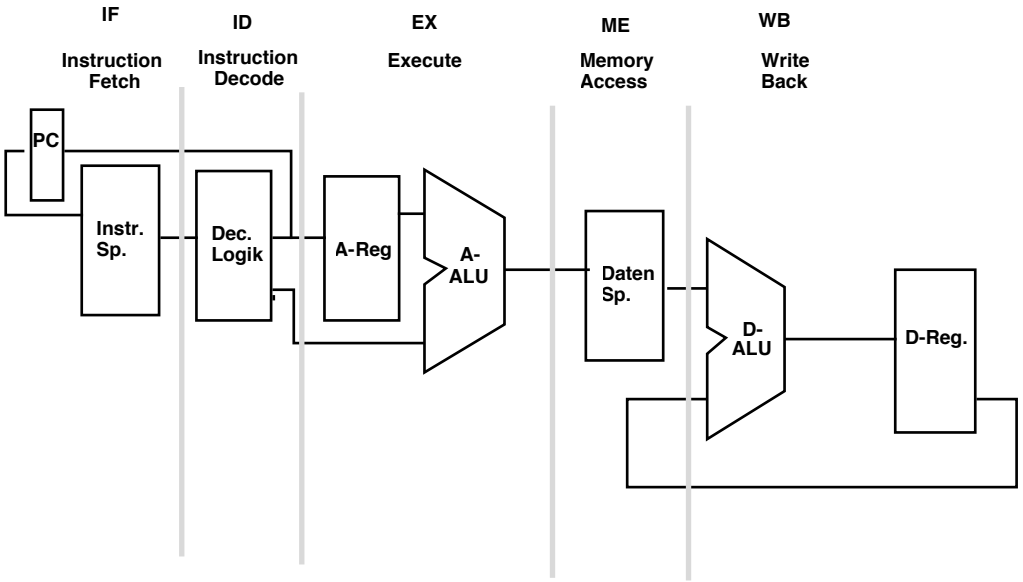
- **Gesamtarbeitsablauf wird in kleinere Einzelschritte unterteilt**
- **Jeder Einzelschritt erledigt eine einzige spezialisierte Aufgabe**
- **Jeder Einzelschritt braucht dieselbe Zeit**
- **Jeder Einzelschritt hängt von der erfolgreichen Bearbeitung des vorhergehenden Schrittes ab**

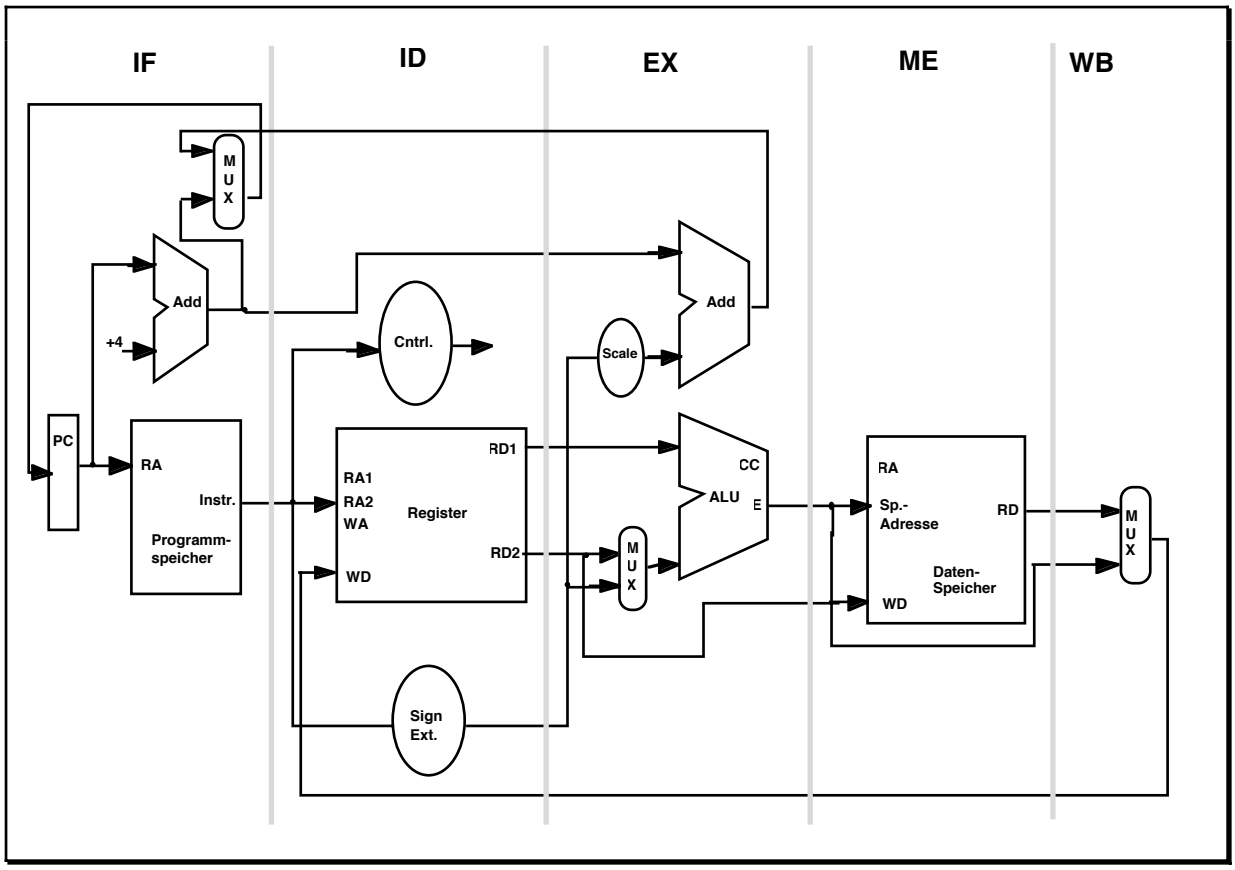
- **Die Anzahl der Abschnitte in der Pipeline bestimmt den Grad der Parallelität**
- **Die Durchlaufzeit durch eine Pipeline ist weitgehend unabhängig von der Anzahl der Abschnitte**
- **Der Durchsatz der Pipeline wächst mit der Anzahl der Abschnitte**

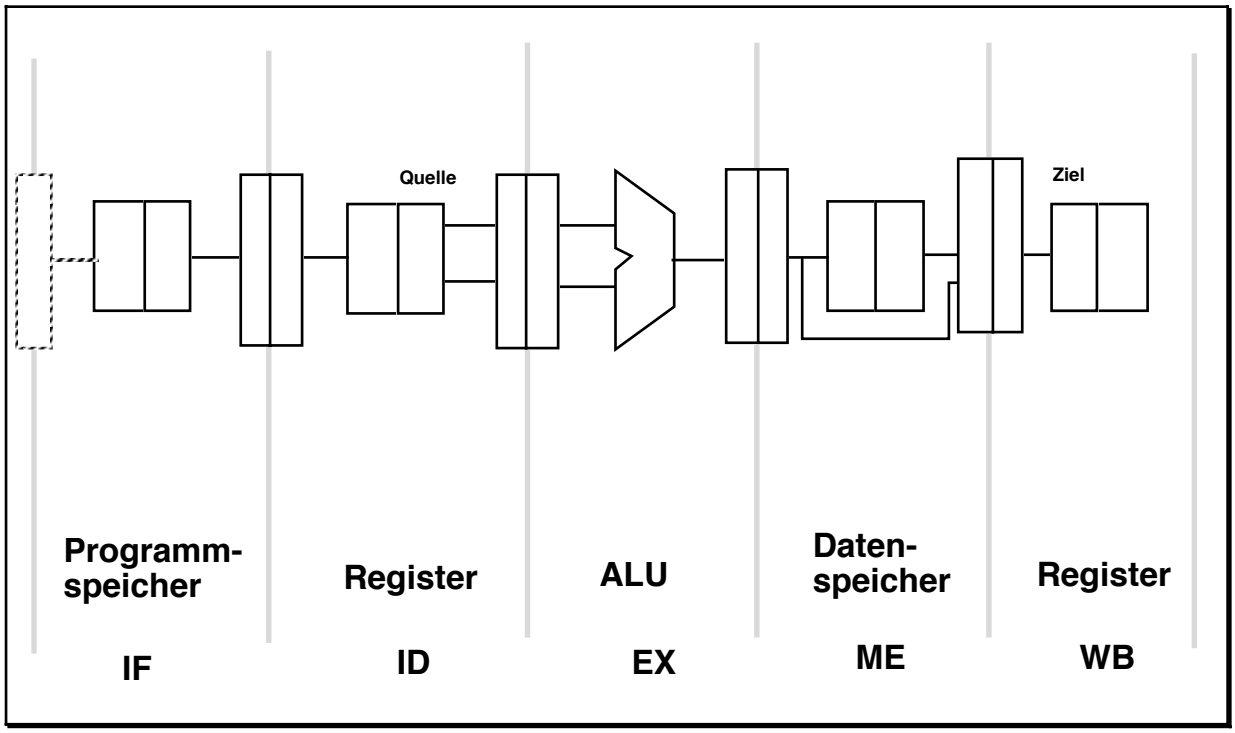
Hauptphasen der Befehlsausführung

- **IF (Instruction Fetch) Befehlsholphase : Die nächste Instruktion wird unter Benutzung des Programmzählers (PC) geladen.**
- **ID (Instruction Decode and Operand Fetching) Decodierungs - und Operanden Auswertungsphase : Der OPCODE und die Operanden in der Instruktion werden ausgewertet und die entsprechenden Kontrollsignale erzeugt. Register werden selektiert und die entprechenden Operanden gelesen.**
- **EX (Instruction Execution) Ausführungsphase : Die durch den OPCODE spezifizierte Operation wird ausgeführt. Bei Instruktionen, die einen Speicherzugriff erfordern, wird in dieser Phase die effektive Adresse berechnet.**
- **ME (Memory Access) Speicherzugriffsphase : Daten werden aus dem Speicher geladen oder in den Speicher geschrieben. Bei RISCs ist meist ein Datencache vorhanden.**
- **WB (Write Back) Zurückschreiben der Ergebnisse in das Dest. Reg. : Das Resultat einer Berechnung wird in ein Zielregister geschrieben.**

Phasen der Befehlsausführung




IF
ID
EX
ME
WB
M
U
X
Add
+4
Cntrl.
Scale
Add
PC
RA
Instr.
Programm-
speicher
RA1
RA2
WA
WD
Register
RD1
RD2
CC
ALU
E
M
U
X
RA
Sp.-
Adresse
RD
Daten-
Speicher
WD
M
U
X
Sign
Ext.


Quelle
Ziel
Programm-
speicher
Register
ALU
Daten-
speicher
Register
IF
ID
EX
ME
WB

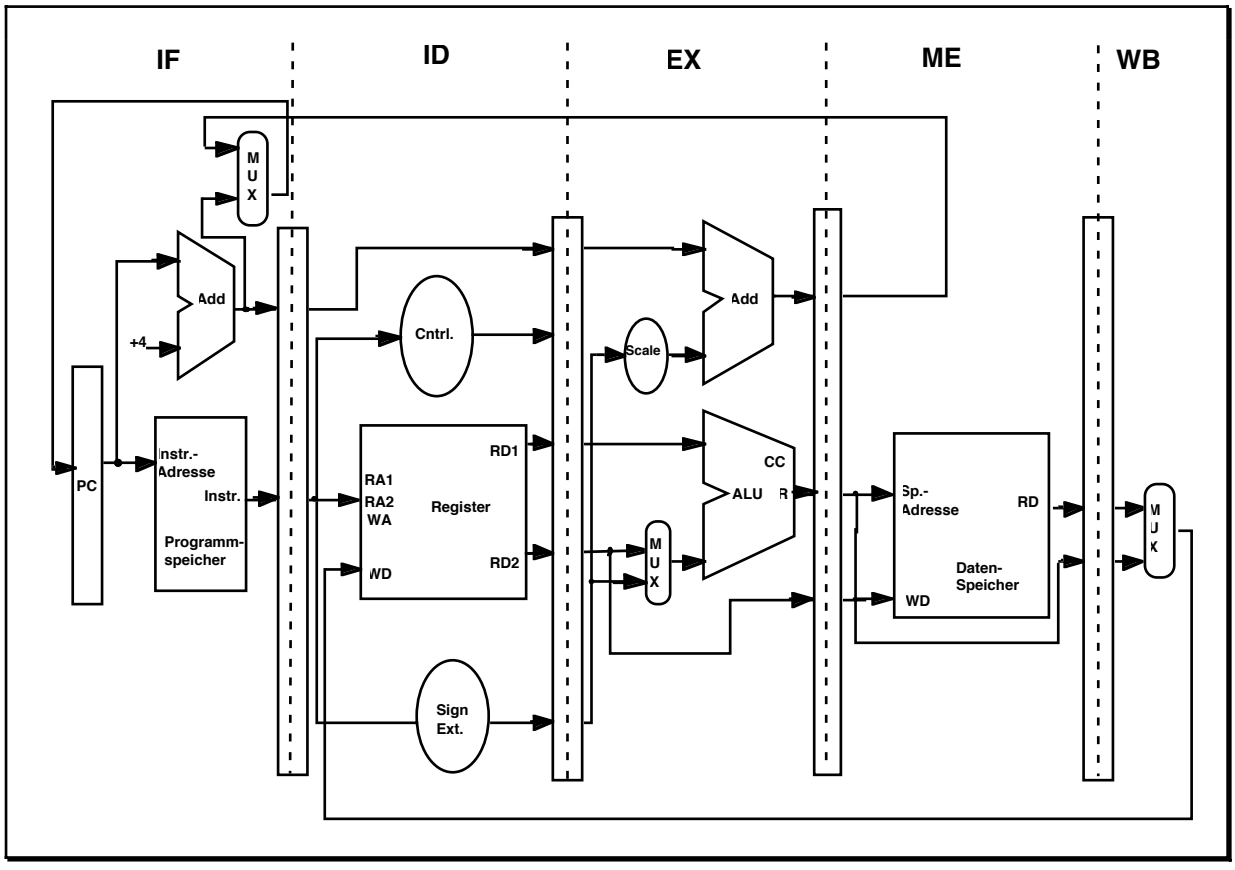
Pipeline-Register

Während der Bearbeitung eines Einzelschritts in Stufe S der Pipeline müssen die Eingangssignale {E_S} in diese Stufe stabil sein. Die Eingangs-Signale {E_s} für Stufe S sind die Ausgangssignale {A_{s-1}} der Stufe S-1.

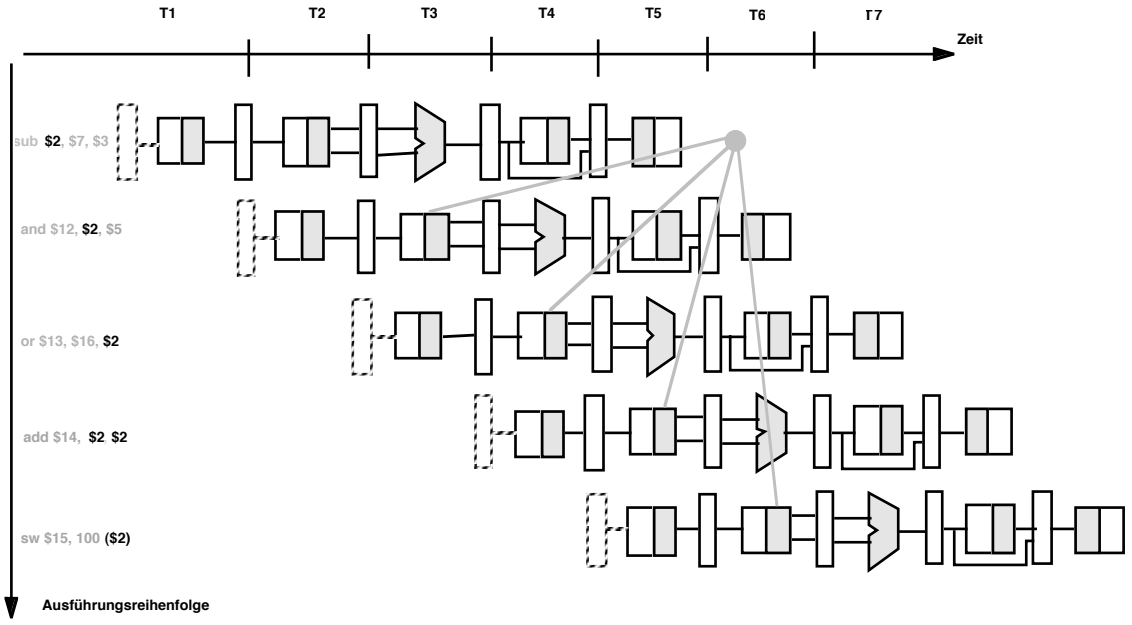
Da die vorhergehende Stufe S-1 aber bereits mit der Bearbeitung des nächsten Auftrags beschäftigt ist, müssen alle Signale {E_s} zwischengespeichert werden.

Die Zwischenspeicherung entkoppelt zwei aufeinanderfolgende Pipeline-Stufen.

Zur Zwischespeichrung werden sogenannte *Pipeline-Register* eingesetzt.



Pipeline - Hindernisse: Datenabhängigkeiten



Pipeline - Hindernisse: Datenabhängigkeiten

Datenabhängigkeiten, die nicht erkannt und behandelt werden, führen zur fehlerhaften Ausführung von Programmen !

Möglichkeiten, mit Datenabhängigkeiten umzugehen?

Software: Der Compiler erzeugt keine abhängigen Instruktionen, d.h. er fügt so viele NOPs ein, daß die Abhängigkeiten nicht auftreten. Er muß also bei jeder Instruktion, die er erzeugt, überprüfen, ob eine Datenabhängigkeit zu einer vorhergehenden Instruktion vorliegt. (eine frühere Version des MIPS-Prozessors nutzte diese Möglichkeit, um seine Hardwarekomplexität klein zu halten)

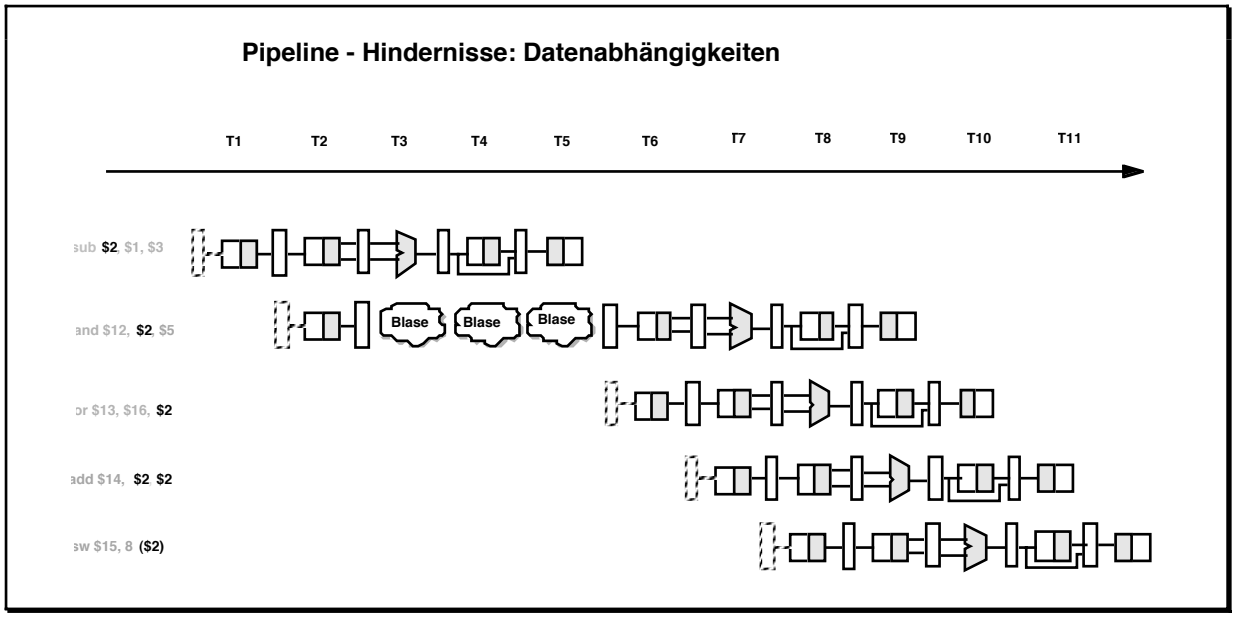
Hardware: Ausnutzung der Bedingungen für die Datenabhängigkeit, um eine automatische Erkennung der Abhängigkeiten während der Laufzeit zu ermöglichen.

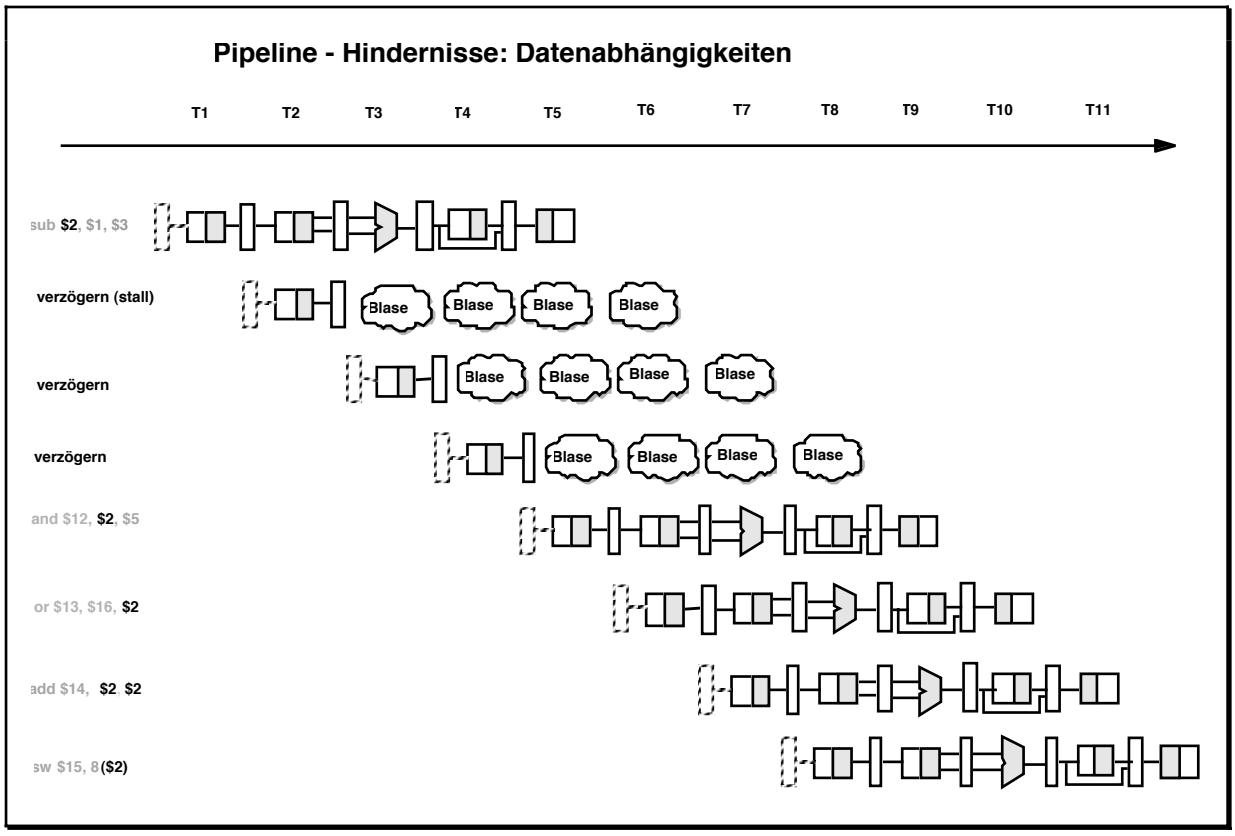
Beispielprogramm:

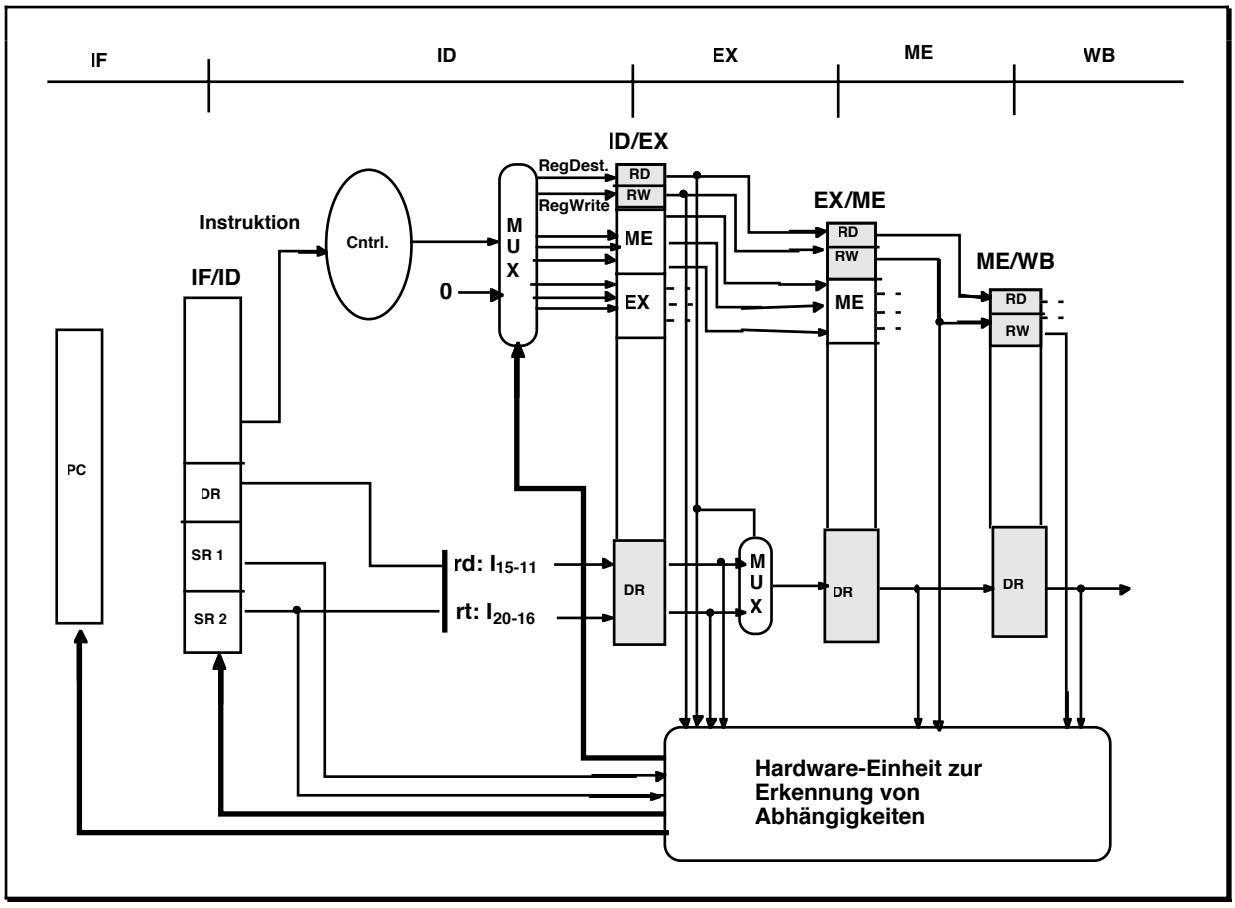
```
lw $10,9($1)
sub $2, $7, $3
and $12, $2, $5
or $13, $10, $2
add $14, $2, $2
sw  $15, 8($2)
```

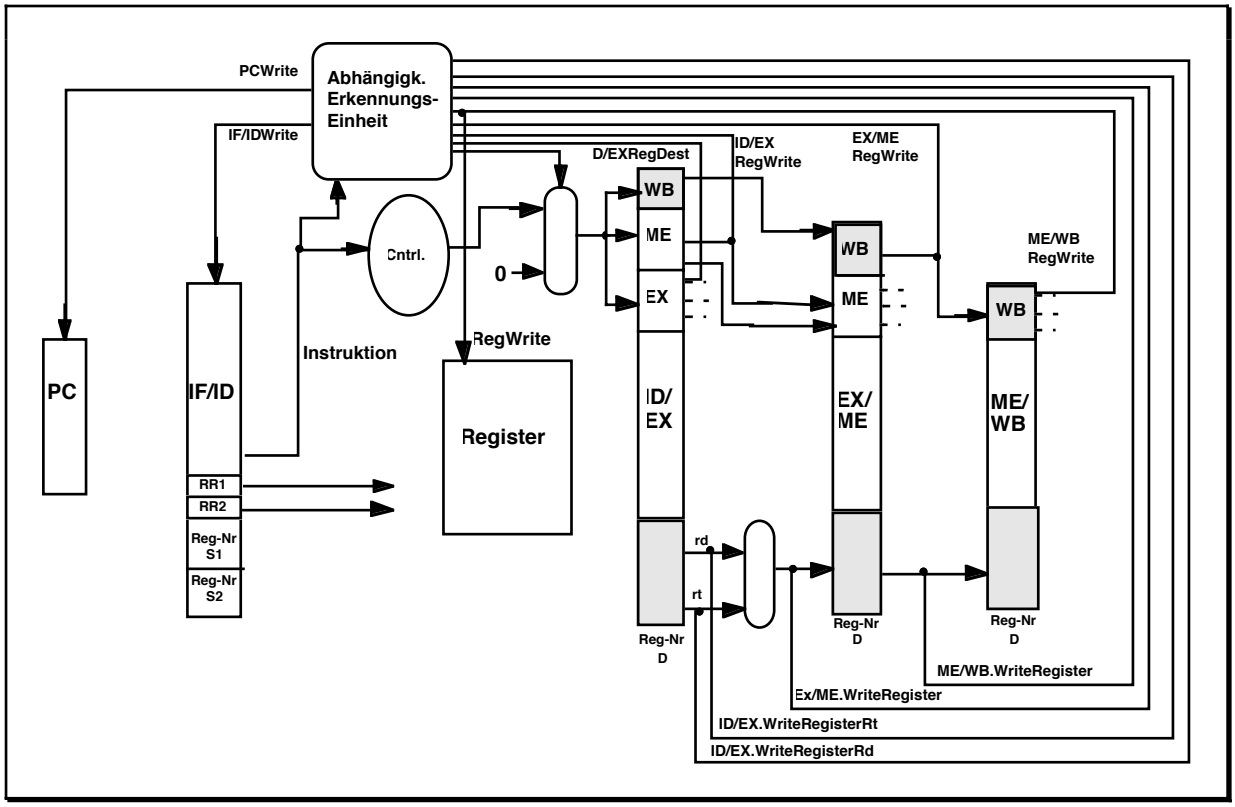
→

```
lw $10,9($1)
sub $2, $7, $3
nop
nop
nop
and $12, $2, $5
or $13, $10, $2
add $14, $2, $2
sw  $15, 8($2)
```

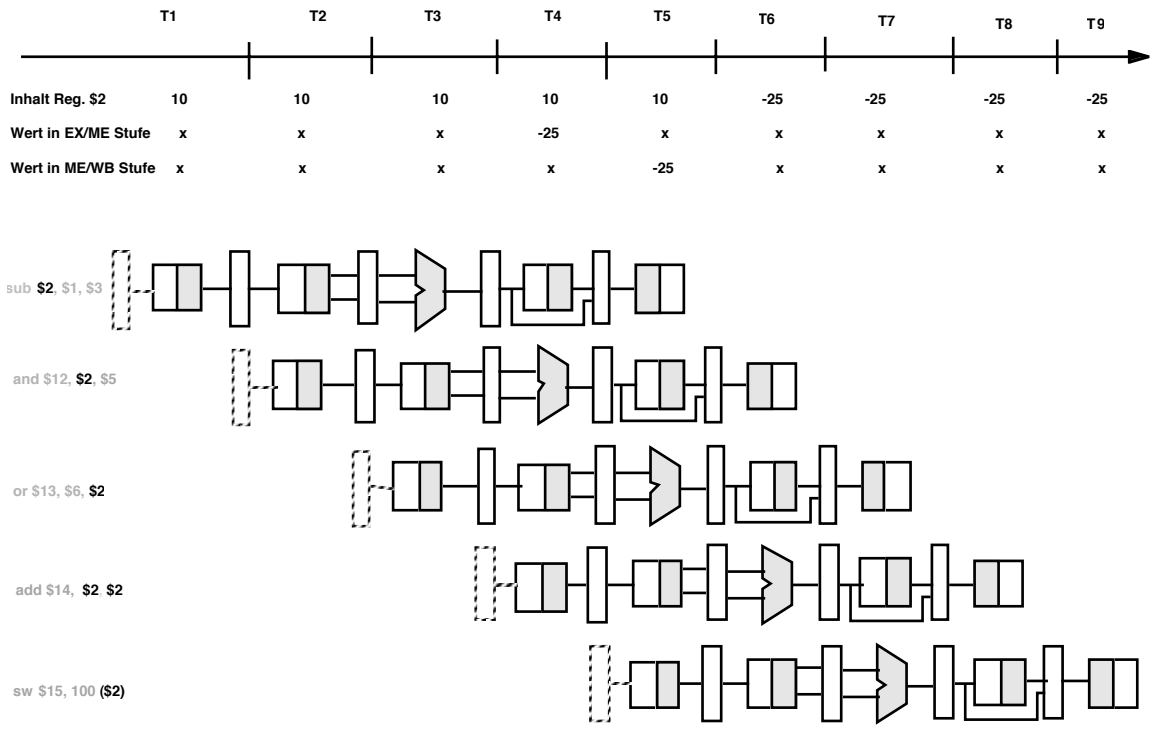

Pipeline - Hindernisse: Datenabhängigkeiten
T1
T2
T3
T4
T5
T6
T7
T8
T9
T10
T11
sub $2, $1, $3
and $12, $2, $5
or $13, $16, $2
add $14, $2, $2
sw $15, 8 ($2)
Blase
Blase
Blase


Pipeline - Hindernisse: Datenabhängigkeiten
T1
T2
T3
T4
T5
T6
T7
T8
T9
T10
T11
sub $2, $1, $3
verzögern (stall)
verzögern
verzögern
and $12, $2, $5
or $13, $16, $2
add $14, $2, $2
sw $15, 8 ($2)
Blase
Blase
Blase
Blase
Blase
Blase
Blase
Blase
Blase
Blase
Blase
Blase

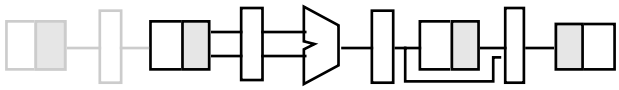

IF
ID
EX
ME
WB
ID/EX
RegDest.
RegWrite
RD
RW
ME
EX
EX/ME
ME/WB
Instruktion
Cntrl.
M
U
X
0
IF/ID
PC
DR
SR 1
SR 2
rd: I15-11
rt: I20-16
Hardware-Einheit zur
Erkennung von
Abhängigkeiten

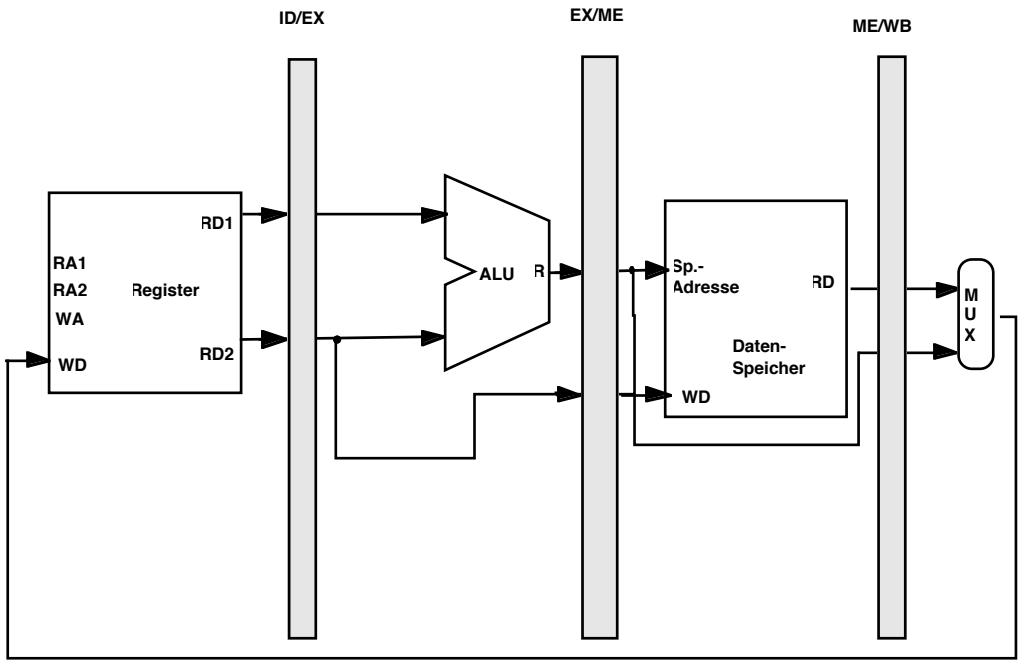

PCWrite
Abhängigk.
Erkennungs-
Einheit
IF/IDWrite
D/EXRegDest
ID/EX
RegWrite
EX/ME
RegWrite
ME/WB
RegWrite
Cntrl.
0
WB
ME
EX
Instruktion
RegWrite
PC
IF/ID
Register
ID/
EX
EX/
ME
ME/
WB
RR1
RR2
Reg-Nr
S1
Reg-Nr
S2
rd
rt
Reg-Nr
D
ME/WB.WriteRegister
Ex/ME.WriteRegister
ID/EX.WriteRegisterRt
ID/EX.WriteRegisterRd

Datenabhängigkeiten zwischen den Stufen der Pipeline

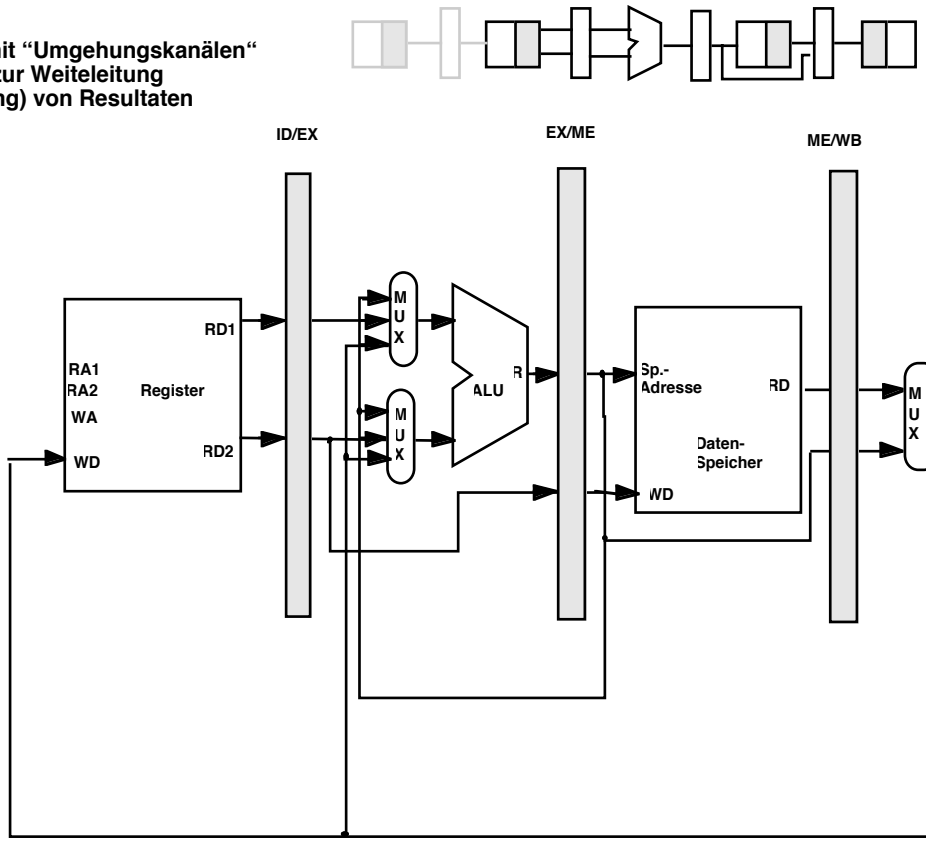


Relevante Stufen der Pipeline ohne Kontrollsignale

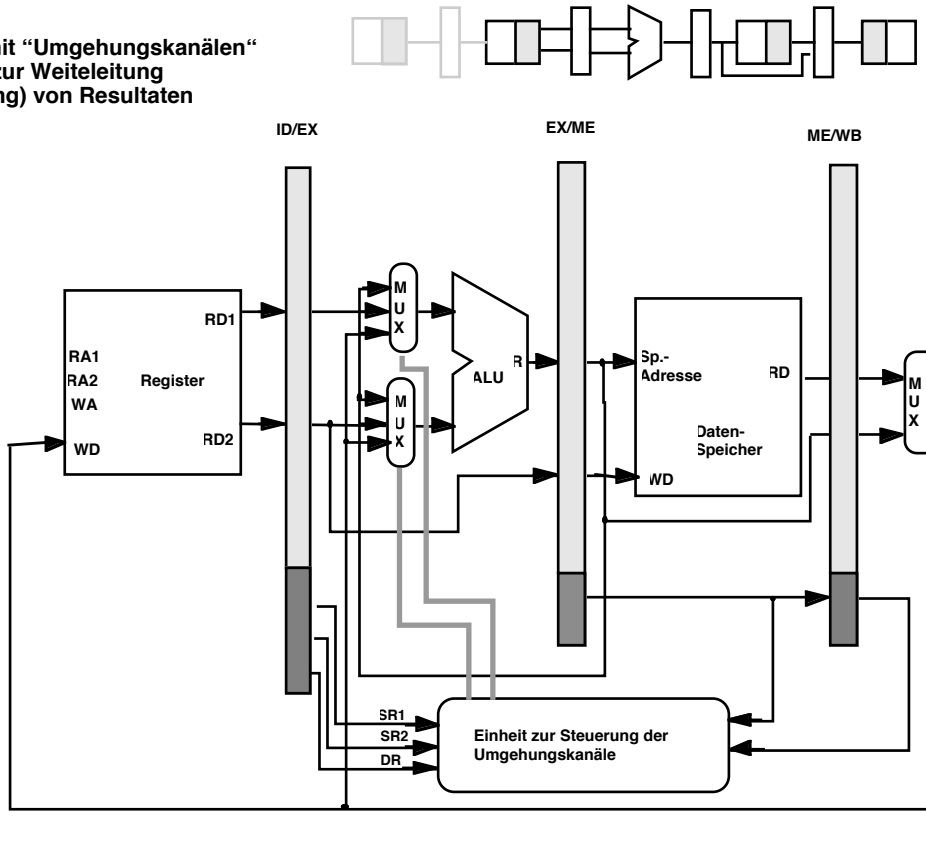


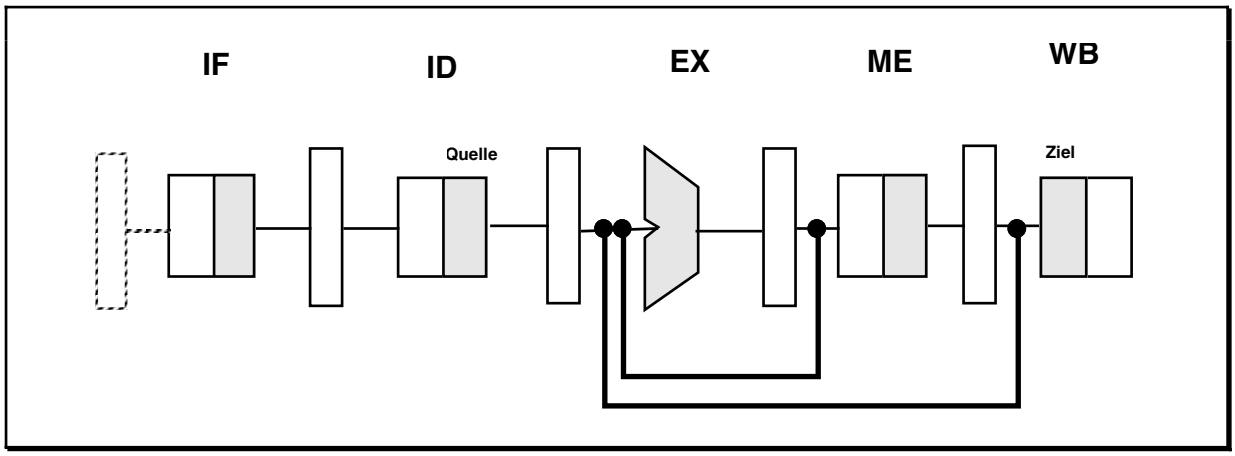


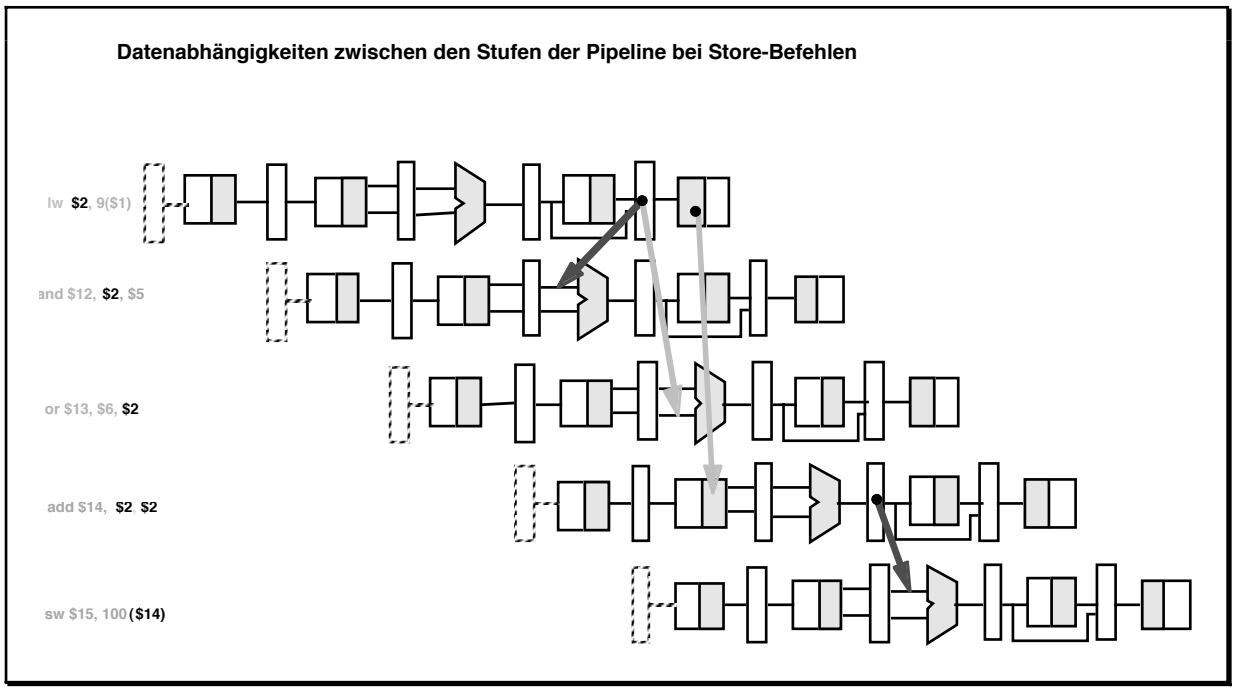
Pipeline mit "Umgehungskanälen" (Bypass) zur Weiteleitung (Forwarding) von Resultaten



Pipeline mit "Umgehungskanälen" (Bypass) zur Weiteleitung (Forwarding) von Resultaten




IF
ID
EX
ME
WB
Quelle
Ziel


Datenabhängigkeiten zwischen den Stufen der Pipeline bei Store-Befehlen
lw $2, 9($1)
and $12, $2, $5
or $13, $6, $2
add $14, $2, $2
sw $15, 100($14)

Auflösung rückwärtsgerichteter Datenabhängigkeiten zwischen den Stufen der Pipeline bei Load-Befehlen

1. Softwarelösung - Einfügen von NOPs

2. Modifikation der Einheit zur Erkennung von Datenabhängigkeiten

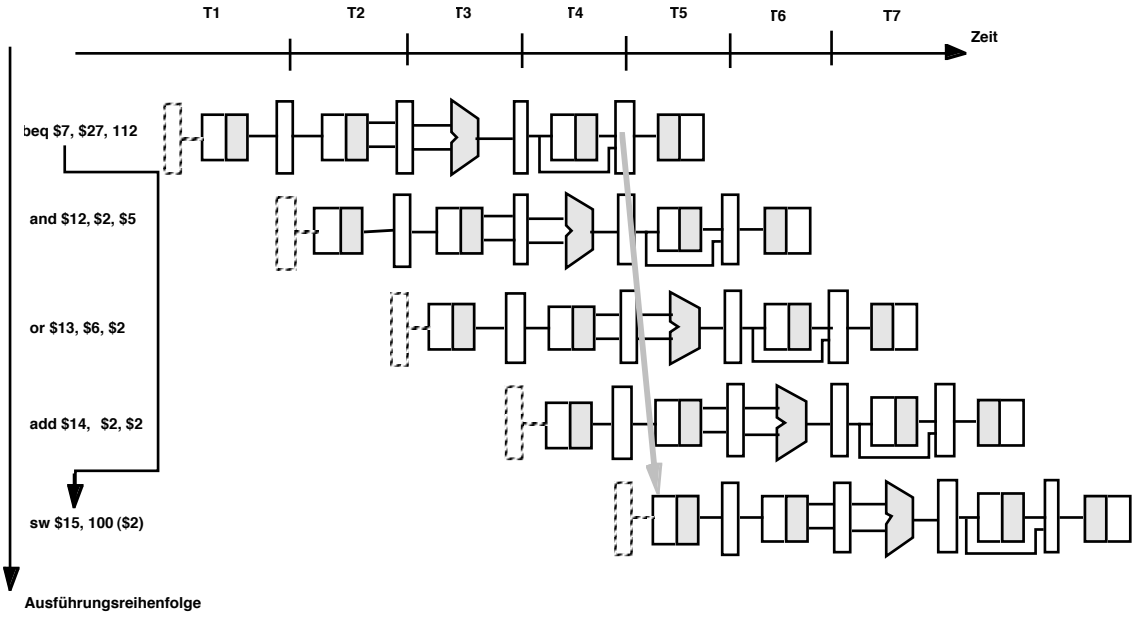
Bedingung:

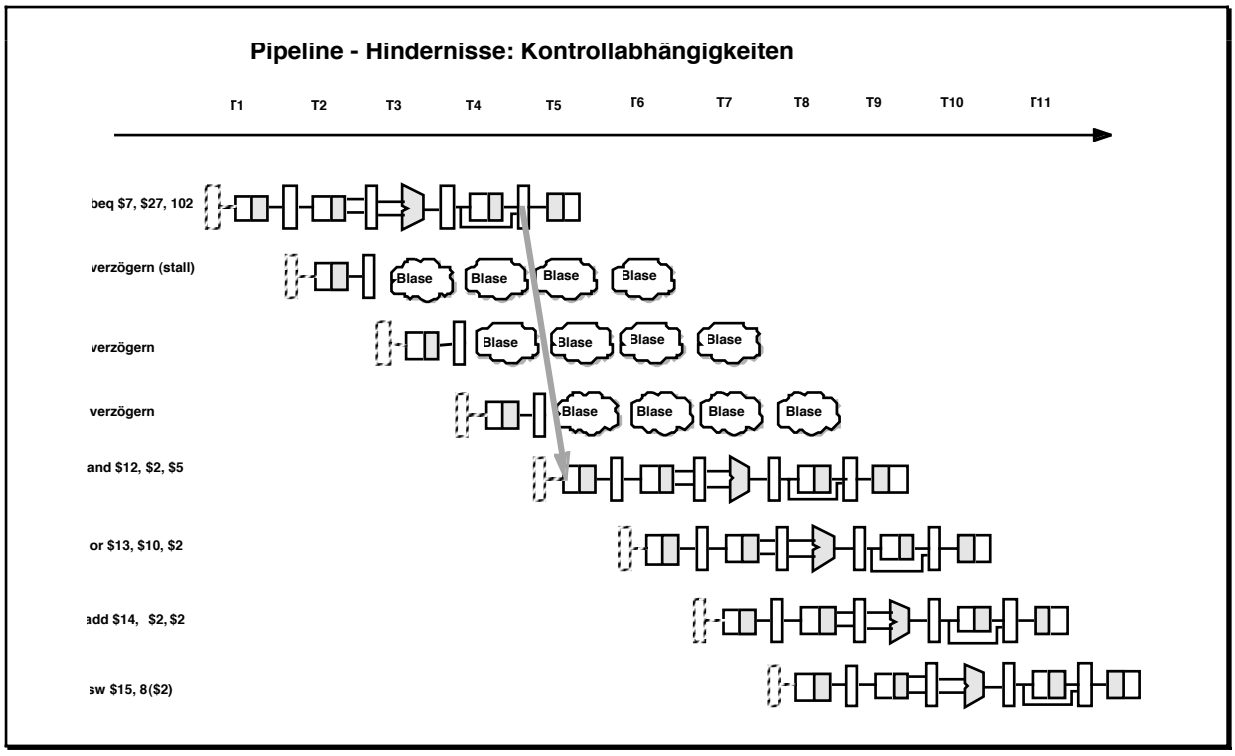
Voraussetzung: Bei einer LOAD-Operation gilt für das Kontrollsignal : ID/EX.RegDst = 0.

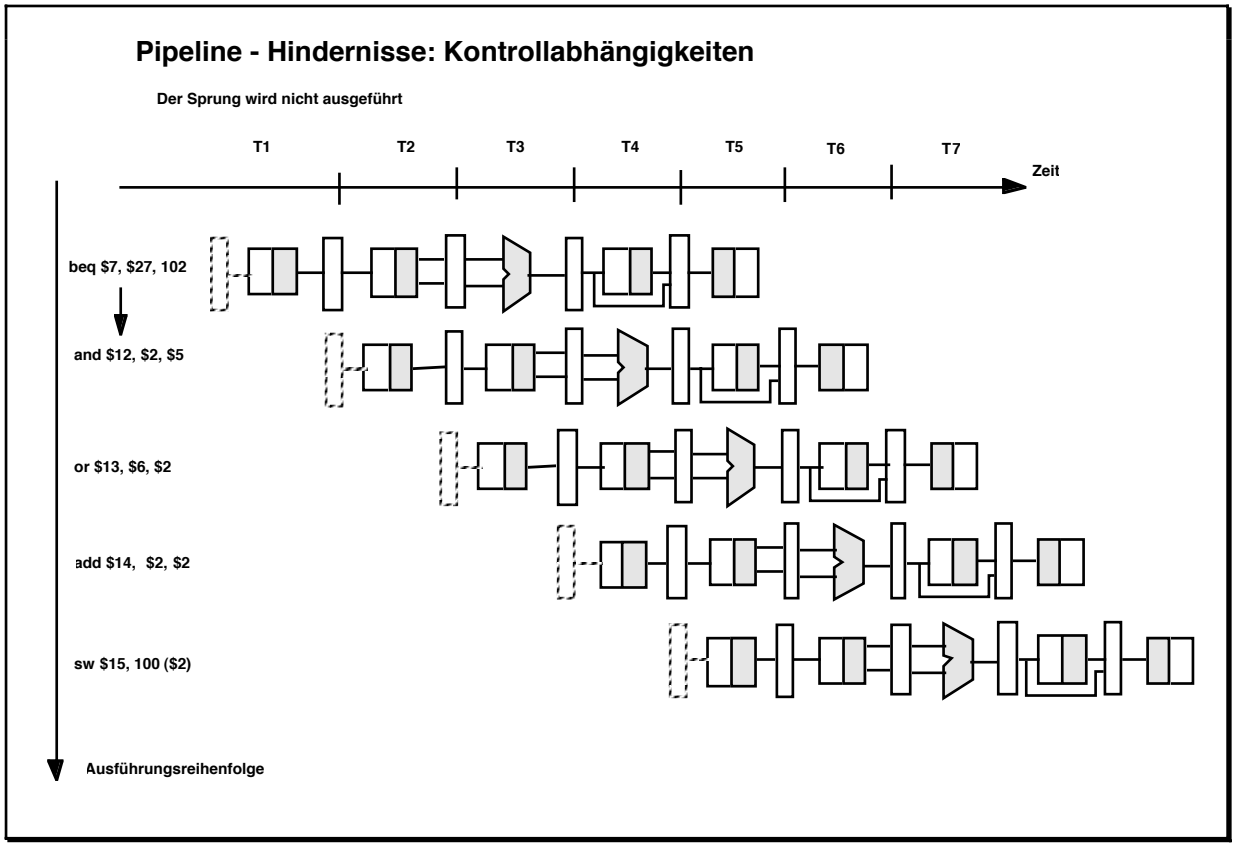
Für die Erkennung der entsprechenden Abhängigkeit gilt deshalb die Bedingung:

IF
(ID/EX.RegWrite • (ID/EX.RegDest = 0) • ((ID/EX.WriteRegRT = IF/ID.ReadRegister1)
+ (ID/EX.WriteRegRT = IF/ID.ReadRegister2)))
THEN
(Verzögern der nächsten Instruktion)

Pipeline - Hindernisse: Kontrollabhängigkeiten




Pipeline - Hindernisse: Kontrollabhängigkeiten
T1
T2
T3
T4
T5
T6
T7
T8
T9
T10
T11
beq $7, $27, 102
verzögern (stall)
Blase
verzögern
verzögern
and $12, $2, $5
or $13, $10, $2
add $14, $2, $2
sw $15, 8($2)


Pipeline - Hindernisse: Kontrollabhängigkeiten
Der Sprung wird nicht ausgeführt
T1
T2
T3
T4
T5
T6
T7
Zeit
beq $7, $27, 102
and $12, $2, $5
or $13, $6, $2
add $14, $2, $2
sw $15, 100 ($2)
Ausführungsreihenfolge

Pipeline - Hindernisse: Kontrollabhängigkeiten

Auflösung von Kontrollabhängigkeiten:

- **Verzögern**

  Zur automatischen Erkennung von Sprüngen muß lediglich der OPCODE ausgewertet werden. Andere Abhängigkeiten zu Instruktionen in der Pipeline bestehen NICHT !

- **Sprung-Voraussage (Branch Prediction)**

- **Verzögerter Sprung (Delayed Branch) : Die Operation unmittelbar hinter dem Sprung wird immer ausgeführt, unabhängig davon, ob der Sprung ausgeführt wird oder nicht.**

- **“Ausrollen“ von Schleifen (Loop Unrolling)**

Zusammenfassung: Pipelining

- **Pipelining ist der Schlüssel zur “Single Cycle“ Architektur**
- **Single cycle ist ein Ziel, das nicht erreicht werden kann**
- **Pipelining nutzt Parallelismus auf der Befehlsebene,**
- **Voraussetzung: angepaßter Instruktionssatz - einfache reguläre Instruktionen**
- **Problem beim Umstieg von CISC auf RISC mit Objekt-Code (Aufwärts-)Kompatibilität**
- **Pipelining für den Programmierer (weitgehend) transparent**
  Tansparent ??? Enge Zusammenarbeit mit dem Copiler ist notwendig

Datenabhängigkeiten:

- **Datenabhängigkeiten können automatisch erkannt werden**
- **Datenabhängigkeiten können durch Umgehungen aufgelöst werden**
- **LOAD ist ein Problem**

Kontrollabhängigkeiten:

- **Sprungabhängigkeiten können nur im Zusammenspiel zwischen SW und HW zufriedenstellend aufgelöst werden**
- **Problem mit Ausnahmebehandlung**

Ausblick:

Superpipelining:	**Anzahl der Pipelinestufen wird erhöht. Dadurch werden die Aufgaben der einzelnen Pipeline-Stufen einfacher. Damit kann auch der Takt kann erhöht werden.**
Parallele Pipelines:	**Anzahl der Pipelines wird erhöht. Erlaubt den Einsatz von Pipelines für spezielle Funktionen.**
Superskalare Pipelines:	**Mehrere Instruktionen werden gleichzeitig geholt und decodiert**

13.2 Leistungsbewertung

Leistungsbewertung:

	Sitzplätze	Reichweite km	Verbrauch l/100km	Geschw. km/h	Durchsatz F·km/h	Oko V/F
Diesel-Pkw	5	850	7	160	800	1,4
S-Klasse	5	600	25	250	1250	5
Formel I	1	150	60	320	320	60
Linienbus	40	1200	20	100	4000	0,5

Andere Leistungsfaktoren:
- **Beschleunigung**
- **Zeit/Strecke (km)**

Wie schnell muß ein Formel I Wagen fahren, um denselben Durchsatz zu erreichen wie ein Omnibus?

Leistungsbewertung:

$$P = \frac{1}{T}$$

P = Leistung (Performance)
T = Ausführungszeit (Execution Time)

Vergleich der Leistung zweier Rechner A und B

$$P_A < P_B \Rightarrow \frac{1}{T_A} < \frac{1}{T_B} \Rightarrow T_A > T_B$$

Verhältnis n der Leistung zweier Rechner A und B

$$\frac{P_A}{P_B} = n \qquad \frac{P_A}{P_B} = \frac{T_B}{T_A} = n$$

Leistungsbewertung:

CPU-(Ausführungs-) Zeit = CPU-User-Zeit + CPU-System-Zeit
Verbrauchte Zeit (elapsed time) = Antwortzeit für den Benutzer

Beziehungen zwischen verschiedenen Meßgrößen:

CPU- Zeit für ein Programm:	**T**
CPU- Taktzyklen für ein Programm:	**Z**
Taktrate = Anzahl derTaktzyklen / Sekunde :	**R**
Zykluszeit = Sekunde / Anzahl der Taktzyklen:	**ZZ**

$$T = Z \cdot \frac{1}{R} = Z \cdot ZZ$$

Anzahl der Instruktionen:	**N**
Mittlere Anzahl der Taktzyklen/Instr.:	**CPI**

$$CPI = \frac{Z}{N}$$

$$Z = N \cdot CPI$$

Leistungsbewertung:

Relation zwischen den wesentlichen Faktoren zur Leistungsbewertung

$$T = \frac{\text{Sekunden}}{\text{Programm}} = \frac{\text{Instruktionen}}{\text{Programm}} \cdot \frac{\text{Taktzyklen}}{\text{Instruktion}} \cdot \frac{\text{Sekunden}}{\text{Taktzyklus}}$$

gemessene Ausführungszeit

$$T = \frac{\text{Sekunden}}{\text{Programm}} = N \cdot CPI \cdot ZZ$$

Leistungsbewertung: Das Maß : MIPS (Million Instructions/ Second)

$$MIPS = \frac{N}{T \cdot 10^6} = \frac{N}{Z \cdot ZZ \cdot 10^6} = \frac{N \cdot R}{N \cdot CPI \cdot 10^6} = \frac{R}{CPI \cdot 10^6}$$

$$T = \frac{N \cdot CPI}{R} = N \frac{CPI}{R} = N \frac{1}{\frac{R}{CPI}} = N \frac{1}{\frac{R}{CPI \cdot 10^6} \cdot 10^6} = \frac{N}{MIPS \cdot 10^6}$$

Leistungsbewertung: Das Maß : MIPS (Million Instructions/ Second)

Peak MIPS vs. Relative MIPS (R-MIPS)

Peak MIPS: **Höchste erreichbare MIPS-Rate.**
Kann in realistischen Anwendungsprogrammen nicht erreicht werden

R-MIPS:

T_{ref} : Ausführungszeit für das Programm auf der Referenzmaschine
T_{ex} : Ausführungszeit für das Programm auf der zu bewertenden Maschine
$MIPS_{ref}$: anerkannte MIPS-Rate der Referenzmaschine

$$R\text{-}Mips = \frac{T_{ref}}{T_{ex}} \cdot MIPS_{ref}$$

z.B. VAX 11/780 wurde häufig als 1-MIPS Referenzmaschine benutzt

Leistungsbewertung:

Erhöhung der Rechengeschwindigkeit (Speedup):

$$\text{Speedup} = \frac{P_V}{P_N} = \frac{T_V}{T_N}$$

P_V (T_V) : Leistung (Ausführungszeit) vor der Verbesserung
P_N (T_N) : Leistung (Ausführungszeit) nach der Verbesserung

vgl. Definition des Leistungsverhältnisses zweier Maschinen.

$$T_N = \frac{T_I}{G_I} + T_R$$

T_I : Ausführungszeit der Instr., die von den Verbesserungen betroffen sind
R_I : Größe der Verbesserungen
T_R: Ausführungszeit der Instr., die von den Verbesserungen nicht betroffen sind

14 INDEX