Skript
zur Vorlesung “Rechnerarchitektur’

P. Marwedel
Informatik 12
Universität Dortmund

3. September 2000

Dieser Begleittext ist nur zur Benutzung durch die Teilnehmer der Vorlesung gedacht.
Es wird keinerlei Gewähr dafür übernommen, dass der Text frei von Urheberrechten ist.

Die vorliegende Version berücksichtigt Ergänzungen, Klarstellungen und Korrekturen des Sommersemesters 2000. Diese betreffen v.a. das Kapitel “Ein-/Ausgabe”.

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Gegenstand der Vorlesung

Willkommen zur Vorlesung "Rechnerarchitektur"!

Was ist Gegenstand der Vorlesung "Rechnerarchitektur"?

In der Vorlesung werden wir das äußere Erscheinungsbild und den inneren Aufbau von Rechnern in der notwendigen Tiefe behandeln. Die soeben erwähnten zwei Aspekte von Rechnern spiegeln sich auch in zwei gängigen Definitionen des Begriffs "Rechnerarchitektur" (RA) wider:

Def. nach Amdahl, Blaauw, Brooks (zitiert nach Bode [BH80a])

The term architecture is used here to describe the attributes of a system as seen by the programmer, i.e., the conceptual structure and functional behaviour, as distinct from the organization and data flow and control, the logical and the physical implementation.

Def. nach Stone (zitiert nach Bode [BH80a]):

The study of computer architecture is the study of the organization and interconnection of components of computer systems. Computer architects construct computers from basic building blocks such as memories, arithmetic units and buses. From these building blocks the computer architect can construct anyone of a number of different types of computers, ranging from the smallest hand-held pocket-calculator to the largest ultra-fast super computer. The functional behaviour of the components of one computer are similar to that of any other computer, whether it be ultra-small or ultra-fast.

By this we mean that a memory performs the storage function, an adder does addition, and an input/output interface passes data from a processor to the outside world, regardless of the nature of the computer in which they are embedded. The major differences between computers lie in the way of the modules are connected together, and the way the computer system is controlled by the programs. In short, computer architecture is the discipline devoted to the design of highly specific and individual computers from a collection of common building blocks.

Zur Unterscheidung bezeichnen wir RA nach Amdahl et al. als **externe** Architektur, RA nach Stone als **interne** oder Mikro-Architektur. Es kann auch die Gegenüberstellung der Begriffe in Tabelle 1.1 benutzt werden.

Die externe Rechnerarchitektur definiert die Programmier- oder Befehlssatzschnittstelle (engl. *instruction set architecture, (ISA)*) und damit eine (reale) Rechenmaschine bzw. ein *application program interface (API)*.

Die Schnittstellen kann man auch anhand eines Schichtenmodells darstellen (siehe Abb. 1.1).

In der Hardware-Beschreibungssprache VHDL (s.u.) heißt jeder Modulrumpf *architecture*, unabhängig vom gewählten Beschreibungsstil.

Programmierschnittstelle	interner Aufbau
externe Rechnerarchitektur Architektur Rechnerarchitektur	interne Rechnerarchitektur Mikroarchitektur Rechnerorganisation [Gil81]

Tabelle 1.1: Gegenüberstellung der Begriffe zur RA

Executables (Binärprogramme)

Betriebssystem

Befehlsschnittstelle *(instruction set architecture)*

Mikroarchitektur

Gatter

Abbildung 1.1: Schichtenmodell

In der Vorlesung werden wir zunächst auf die äußere RA eingehen. Als nächstes werden wir uns dem inneren Aufbau widmen. Speziell werden wir den Aufbau der Prozessoren (CPUs), des Hauptspeichers, der Ein/Ausgabeschnittstellen und der Peripherie behandeln. Abschließend werden wir uns mit Mehrprozessorsystemen beschäftigen.

Die Vorlesung "Rechnerarchitektur" (RA) ist eine Stammvorlesung im Studiengang Informatik und eine Wahlpflichtvorlesung im Studiengang "Angewandte Informatik". Sie ist eine von relativ wenigen Vorlesungen aus dem Bereich der Technischen Informatik an der Universität Dortmund. Aufgrund der typischen Berufsfelder heutiger Informatiker [CEF$^+$95] ist es in der Tat nicht (mehr) erforderlich, **jeden** Informatiker **umfangreich** in Technischer Informatik auszubilden. Fundierte **Grundkenntnisse** sind dennoch dringend erforderlich, damit Informatiker eine Vorstellung und ein **Grundverständnis** von den Geräten haben, mit denen sie umgehen. Diesem Ziel dient die Vorlesung "Rechnerarchitektur".

Warum sollte sich nun **jeder** Informatiker und **jede** Informatikerin mit dem Stoff dieser Vorlesung beschäftigen? Nun, die Bedeutung des Grundverständnisses der Geräte wurde bereits erwähnt. Dieses Grundverständnis wird u.a. bei folgenden Tätigkeiten eines Informatikers bzw. einer Informatikerin benötigt:

- bei der Geräteauswahl,
- bei der Fehlersuche,
- bei der Leistungsoptimierung,
- bei Zuverlässigkeitsanalysen,
- beim Neuentwurf von Systemen,
- beim *accounting*,
- bei der Codeoptimierung im Compilerbau,
- bei Benchmarkentwürfen,
- bei Untersuchung des Einflusses der Paging-Hardware,
- bei Sicherheitsfragen.

Letzlich sollten sich Informatiker bzw. Informatikerinnen auch nicht durch grobe Wissenslücken in zentralen Bereichen der Datenverarbeitung blamieren.

1.2 Der Von-Neumann-Rechner

Wir werden in dieser Vorlesung überwiegend die heute kommerziell erhältlichen und verbreiteten Rechner behandeln. Diese besitzen viele, aber nicht immer alle Merkmale des sog. *Von-Neumann-Rechners.* Wir halten eine Behandlung der kommerziell erhältlichen und verbreiteten Rechner in einer Vorlesung für sinnvoll, auch wenn aus Sicht mancher Anwendungen andere Konzepte günstiger wären. Solche anderen Konzepte werden kurz im Kapitel 2 behandelt werden. Darüber hinaus sollen sie, soweit von der Lehrkapazität her möglich, in weiterführenden Spezial-Veranstaltungen behandelt werden.

Da nicht immer klar ist, welche Eigenschaften mit dem Begriff des Von-Neumann-Rechners verbunden wird, sollen diese gemäß der ursprünglichen Erfindung durch von Neumann hier aufgelistet werden[1] :

1. Die Rechenanlage besteht aus den Funktionseinheiten Speicher, Leitwerk (engl. *controller*), Rechenwerk (engl. *data path*) und Ein/Ausgabeeinheiten (einschl. der notwendigen E/A-Geräte-Steuerungen). Abb. 1.2 zeigt diese Einheiten und deren Verschaltung innerhalb einer busorientierten Architektur.

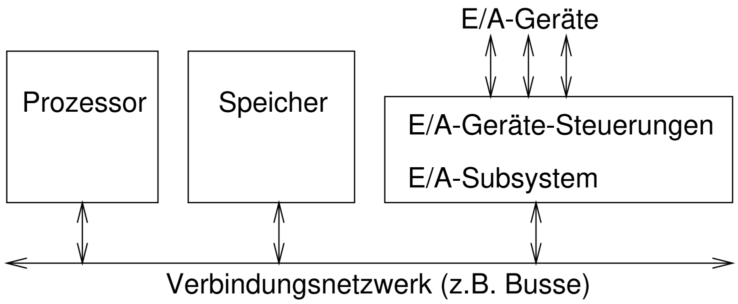


Abbildung 1.2: Funktionseinheiten des von-Neumann-Rechners

2. Die Struktur der Anlage ist unabhängig vom bearbeiteten Problem. Das Problem wird durch einen austauschbaren Inhalt des Speichers beschrieben, die Anlage ist also speicherprogrammierbar. Dies unterscheidet sie z.B. von durch passend gesteckte Verbindungen strukturierten Analogrechnern und auf spezielle Algorithmen zugeschnittenen VLSI-Chips.

3. Anweisungen und Operanden (einschl. der Zwischenergebnisse) werden in demselben physikalischen Speicher untergebracht. Ursprünglich setzte man große Erwartungen in die Möglichkeit, dadurch nicht nur Operanden, sondern auch Programme dynamisch modifizieren zu können. Die Vorteile von ablaufinvarianten Programmen (*pure code*) wurden erst später erkannt.

4. Der Speicher wird in Zellen gleicher Größe geteilt, die fortlaufend numeriert werden. Diese Nummern heißen Adressen.

5. Das Programm wird dargestellt als Folge von elementaren Anweisungen (Befehlen), die in der Reihenfolge der Ausführung in Zellen steigender Adressen gespeichert werden. Gleichzeitige Abarbeitung von Befehlen wurde von von-Neumann nicht in Aussicht genommen. Jeder Befehl enthält einen Operator (den Operationscode) und eine Angabe zur Bezeichnung von Operanden. Diese Angabe stellt in der Regel einen Verweis auf den Operanden dar. Die Adressierung der Befehle erfolgt über ein Befehlszählregister.

6. Abweichungen von der gespeicherten Reihenfolge der Befehle werden durch gesonderte Befehle (Sprungbefehle) ausgelöst. Diese können bedingt oder unbedingt sein.

7. Es werden Binärzeichen (im folgenden Bitvektoren genannt) bzw. Binärsignale verwendet, um alle Größen (Operanden, Befehle und Adressen) darzustellen.

8. Die Bitvektoren enthalten keine explizite Angabe des durch sie repräsentieren Typs (von Ausnahmen, wie z.B. PROLOG-Maschinen abgesehen). Es muß immer aus dem Zusammenhang heraus klar sein, wie Bitvektoren zu interpretieren sind.

[1] Quelle: [Jes75]

Für den von-Neumann-Rechner ist weiter typisch, dass die Interpretation von Bitvektoren als Zahlen oder Befehle durch Funktionen erfolgen muß, die aus dem Kontext bekannt sein müssen. Die explizite Abspeicherung von Datentyp-Tags zur Identifikation des Datentypes (siehe Abb. 1.3), wie sie bei den sog. Datentyp-Architekturen [Gil81] vorgesehen ist, hat sich im Allgemeinen nicht durchgesetzt.

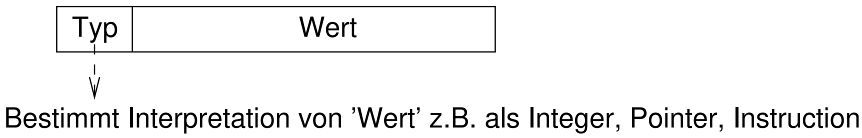


Abbildung 1.3: Explizite Datentyp-Speicherung

1.3 Literatur

Zur Gliederung des Informatikstudiums in Dortmund optimal passende Bücher über RA existieren z.Z. weder in deutscher noch in englischer Sprache.

Weltweit sehr weit verbreitet ist die Orientierung von RA-Vorlesungen an dem hervorragenden Buch "Computer Architecture – A Quantitative Approach" von Hennessy und Patterson [HP96]. Dieses Buch ist in der Lehrbuchsammlung enthalten. Dieselben Autoren haben ein zweites Buch mit dem Titel "Computer Organization & Design – The Hardware/Software Interface" [HP95] herausgebracht, welches zur Benutzung in einer früheren Studienphase gedacht ist. Für beide Bücher gibt es zahlreiche ergänzende Quellen, die über die Web-Seite des Verlages zu erreichen sind (www.mkp.com).

Teilweise wird sich die Vorlesung an diese Bücher anlehnen. Eine vollständige Orientierung soll allerdings unterbleiben, da die Bücher sehr stark auf die Fähigkeit zielt, selbst Prozessoren zu entwickeln. Diese Fähigkeit wird im Silicon Valley, im dem die Autoren arbeiten, auch zur Weiterentwicklung beispielsweise der Produkte der Firmen Sun, MIPS und Intel benötigt. Zu diesem Zweck wird in dem Buch fast ausschließlich der Entwurf einer sehr starken Fließbandverarbeitung in modernen Prozessoren untersucht. Dabei werden die Prozessoren eingebetteter Systeme nur wenig berücksichtigt. Europäische Informatiker werden hiervon in der Regel wenig Nutzen haben.

Ein didaktisch gutes deutsches Buch ist das Buch "Mikrorechnersysteme" von H. Bähring [Bae94]. Von diesem Buch besitzt die Lehrbuchsammlung 60 Exemplare. Eine zweite Auflage ist im Oktober 1994 (ohne größere Änderungen gegenüber der ersten Auflage) erschienen. Dieses Buch ist allerdings vom Stoff her z.Tl. stark elektrotechnisch geprägt.

Neuere Bücher enthalten v.a. eine ausführlichere Behandlung von Parallelrechnern. Genannt werden sollen hier die Bücher von Culler und Singh [CS99] sowie von Silc, Robic und Ungerer [SRU99].

In dieser Vorlesung werden wir den fünf Büchern jeweils geeigneten Stoff entnehmen. Zusätzlich werden wir Stoff zu speziellen Themen aus anderen Büchern verwenden. Diese Bücher werden jeweils angegeben werden. Schließlich werden wir im Interesse der Aktualität gelegentlich auf Zeitschriftenartikel zurückgreifen.

Die angegebenen Gründe für die Stoffauswahl machen diesen Begleittext notwendig. Dieser gibt den Stoff der Vorlesung weitgehend wieder. Die vorliegende Fassung des Sommersemesters 2000 unterscheidet sich von der Vorgängerversion u.a. durch

- eine Betonung der Befehlsschnittstelle,
- Ausweitung im Bereich der Multiprozessorsysteme,
- Aktualisierung bei Befehlssätzen und Technologie,
- Anpassung an die Darstellung des MIPS-Befehlssatzes und seine Realisierung in [HP95].
- Eine Überarbeitung des Kapitels "Ein-/Ausgabe"

Einige Themen können weiterhin nur stichwortartig behandelt werden und zur ausführlichen Erläuterung muß ggf. auf eine der angegebenen Quellen zurückgegriffen werden. **Dieser Text ist zur Benutzung in**

Kombination mit der Teilnahme an der Vorlesung konzipiert und nur bedingt für das Selbststudium geignet. Für das Selbststudium wäre in der Regel eine wesentlich ausführlichere Formulierung notwendig. Eine Prüfungsvorbereitung im Selbststudium erfordert auf jeden Fall, zu ausgewählten Themen die o.a. Bücher zu Rate zu ziehen.

Zur Zeit wird die Vorlesung alternierend von Herrn Lindemann und vom Autor dieses Skripts gehalten. Die Tabelle 1.2 zeigt einige Unterschiede zwischen den beiden Vorlesungen.

	Marwedel	Lindemann
Ausgeteiltes Material	Skript	Folien
Bezug zu Hennessy/Patterson	'normal'	stark
Behandlung von Leistungsbewertung	nein	ausführlich
Behandlung von Arithmetik	'normal'	wenig
Behandlung von Hardwaretechnik	ja	wenig
Behandlung von Parallelrechnern	'normal'	ausführlich
Details des Fließbandentwurfs	nein	ja
Behandlung von eingebetteten Prozessoren	ja	kaum

Tabelle 1.2: Vergleich der beiden angebotenen Rechnerarchitektur-Vorlesungen

1.4 Prüfungen

Prüfungen im Rahmen der Wahlpflichtkataloge über die Vorlesung RA sind sowohl für den Studiengang "Informatik" wie auch für den Studiengang "Angewandte Informatik" möglich. Im Falle des Studiengangs "Informatik" erfolgt die Prüfung zusammen mit der Prüfung über eine zweite Stammvorlesung. Typisch ist eine Kombination mit den Vorlesungen "Informationssysteme" oder "Betriebssysteme". Andere Kombinationen sind nach Absprache möglich.

Die Möglichkeit, Stammvorlesungen als Spezialvorlesung anrechnen zu lassen, läuft aufgrund einer Änderung der DPO Informatik aus.

1.5 VHDL-Notation

Die Beschreibung der Arbeitsweise von Rechnern erfordert die Verwendung einer klar definierten Sprache. Zu diesem Zweck sind verschiedene Hardware-Beschreibungssprachen (engl. *hardware description languages*, HDLs) entwickelt worden. DIe Motivation für die Benutzung von HDLs liegt in

- der präzisen Spezifikation,
- der Möglichkeit der Simulation,
- der Kommunikation zwischen Entwicklern,
- der automatisierten Erzeugung und der Überprüfung des Entwicklungsergebnisses,
- der Dokumentation des Arbeitsergebnisses,
- der Beschleunigung des Entwurfsprozesses.

Sehr verbreitet ist die Sprache VHDL (VHSIC Hardware Description Language) [IEE88, IEE92]. VHSIC wiederum steht für *very high speed integrated circuit*. Das VHSIC-Progamm geht auf eine Initiative des amerikanischen Verteidigungsministeriums (DoD) zurück.

VHDL ist für unsere Zwecke geeignet. Ein wesentlicher Grund für die Verwendung von VHDL in diesem Text ist, dass VHDL eine streng getypte Hardwarebeschreibungs-Sprache ist. Dadurch führt VHDL zu dem nützlichen Zwang, zwischen Folgen von Bits und deren Interpretion als Wert in einem anderen Bereich (wie z.B. dem der natürlichen Zahlen) immer streng zu unterscheiden.

VHDL ist allerdings eine sehr komplexe, auf der Sprache ADA basierende Sprache. Wir werden daher nur eine kleine Teilmenge der Sprache vorstellen. Speziell werden wir die zentralen Datentypen verwenden.

1.5.1 Datentypen integer und natural

Als Datentyp ist in VHDL der Datentyp `integer` vordefiniert. Die oberen Grenzen des mit diesem Typ darstellbaren Zahlenbereichs sind Implementations-abhängig. Leider unterstützen viele VHDL-Implementierungen nur die mit 32 Bit darstellbaren Zahlen. Mit Hilfe des Ausdrucks `integer'high` kann man auf die obere Grenze des jeweils darstellbaren Integer-Zahlenbereichs Bezug nehmen.

Auf dem Datentyp `integer` sind in VHDL die üblichen arithmetischen Operationen `+, <` usw. definiert.

Aus dem Datentyp `integer` können in VHDL mit Hilfe von `subtype`-Definitionen weitere Sub-Typen abgeleitet werden. Beispiel:

```
subtype natural  is integer range 0 to integer'high;
subtype positive is integer range 1 to integer'high;
```

Variablen von derartig abgeleiteten Typen sind zuweisungskompatibel.

1.5.2 Datentyp boolean

Als weiterer Datentyp ist der Typ `boolean` vordefiniert:

```
type boolean is (False, True);
```

`boolean` wird also durch Aufzählung der möglichen Werte definiert.

1.5.3 Datentypen bit und bit_vector

Weiterhin ist der Datentyp `bit` vordefiniert. Literale des Typs `bit` werden durch einfache Anführungszeichen bezeichnet. Beispiel:

```
'0'
```

Der Datentyp `bit` ist in VHDL als Aufzählungstyp definiert durch

```
type bit is ('0','1');
```

Als weiterer Datentyp ist der Typ `bit_vector` vordefiniert als

```
type bit_vector is array (natural range <>) of bit;
```

Dabei kennzeichnen die Klammern <> den ausgelassenen Index eines sogenannten *unconstrained array*, eines Arrays, dessen Indexgrenzen durch einen Teilbereich der natürlichen Zahlen später festgelegt werden können. Dies geschieht beispielsweise in einer Definition der Art

```
variable instruction : bit_vector (31 downto 0);
```

Das Schlüsselwort `downto` zeigt dabei einen **absteigenden** Indexbereich an. VHDL erlaubt zusätzlich auch **aufsteigende** Indexbereiche. **Zur Verweidung von Konfusion werden wir stets absteigende Indexbereiche mit der unteren Indexgrenze 0 benutzen**. Solche Indexbereiche werden auch in den meisten auf VHDL aufbauenden Standards, wie z.B. dem *VHDL math package* eingesetzt.

Literale des Typs `bit_vector` werden in doppelten Anführungszeichen eingeschlossen. Beispiel:

```
"01010101"
```

In VHDL kann auch explizit angegeben werden, dass es sich bei einem Literal um ein Binärliteral handelt. Dies geschieht durch Voranstellen von `B`:

```
B"01010101"
```

Auf den Daten `bit` und `bit_vector` sind in VHDL die üblichen logischen Operationen sowie die Konkatenation vordefiniert. Letztere wird durch ein `&`-Zeichen dargestellt. Beispiele:

```
"01010101" & '0'
"01010101" & "01010101"
'1' & '0'
```

In Kapitel 2 werden wir beschreiben, wie Bitvektoren mittels der Funktionen `nat` und `int` als natürliche bzw. ganze Zahlen interpretiert werden können.

1.5.4 Verwendung von VHDL im Skript

Weitere Sprachelemente von VHDL sind selbsterklärend, bzw. werden jeweils zusammen mit der ersten Benutzung erläutert werden. Überwiegend werden wir uns auch auf die Verwendung der Datentypen `integer, boolean, bit, bit_vector`, der Subtypen `positive` und `natural`, und der Operationen bzw. Funktionen `nat, int, bit_vect, +` (siehe Kap. 2) und `&` beschränken. Die genaue Syntax wird im folgenden nicht wesentlich sein. Ebenfalls werden wir die Möglichkeit, Zeitverhalten in VHDL zu beschreiben, in diesem Text nicht benötigen.

Abweichend vom VHDL-Standard werden wir aus Platzgründen statt **downto** gelegentlich den Doppelpunkt verwenden, wie z.B. in "`a (5:3)`". In Formeln werden Indizes auch in der üblichen mathematischen Notation angegeben werden.

Kapitel 2

Die Befehlsschnittstelle

I think there is a world market for maybe five computers
[Thomas Watson, Vorstandsvorsitzender der Fa. IBM, 1943]

In den folgenden beiden Kapiteln werden wir Prozessoren behandeln.

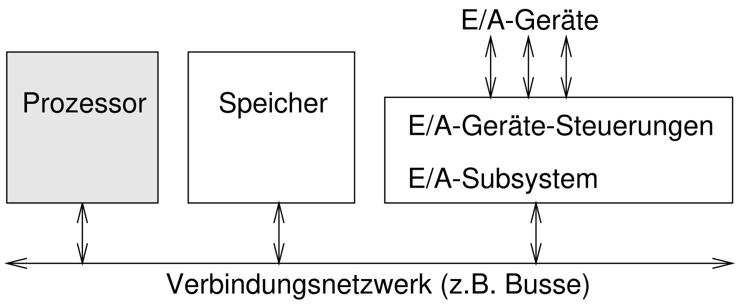


Abbildung 2.1: Stoff der folgenden beiden Kapitel

Dabei werden wir zunächst die externe Sicht auf Prozessoren behandeln. Diese umfasst eine **logische** Sicht auf das Gesamtsystem eines Prozessors.

Die logische Sicht von Prozessoren enthält

- die unterstützten elementaren Datentypen (wie z.B. `Integer` und `Real`-Typen),
- die durch die Maschinenbefehle ansprechbare logische Speicherorganisation,
- den Maschinenbefehlssatz des Prozessors sowie
- das logische Verhalten der Interruptorganisation.

2.1 Elementare Datentypen und deren Darstellung

Rechner stellen in der Regel gewisse elementare Datentypen zur Verfügung, wobei sie von einer standardisierten Informationsdarstellung (d.h.: Interpretation von Bitvektoren) ausgehen und Grund-Operationen (wie z.B. die Addition) anbieten. Diese Datentypen stellen im weitesten Sinne **abstrakte Datentypen** dar [Gil81], wie sie aus Grundvorlesungen bekannt sind.

Eine besondere Charakteristik des von-Neumann-Rechners ist es, dass auf gespeicherte Werte nicht nur mit den eigentlich dazugehörigen Operationen zugegriffen werden kann, sondern dass beliebige Operationen auf beliebigen Speicherwerten zulässig sind. So verhindert es die Hardware in der Regel nicht, dass Gleitkomma-Zahlen mit Integer-Operationen verarbeitet werden.

Wir werden in separaten Abschnitten Operationen auf Bitvektoren sowie die Operationen für natürliche, für ganze und für Gleitkomma-Zahlen besprechen. Rechnerarithmetik ist an unserer Universität bereits Gegenstand der Vorlesung "Rechnerstrukturen" vor dem Vordiplom. In der Vorlesung "Rechnerarchitektur" wollen wir ausgewählte ergänzende (z.Tl. auch wiederholende) Themen besprechen. Rechnerarithmetik ist bei vielen zwar ein unbeliebtes Thema. Gewisse Grundkenntnisse in diesem Bereich sind aber erforderlich, z.B. wenn Pakete zur mehrfachen Genauigkeit erstellt werden müssen, wenn die vorhandene Arithmetik zur Simulation anderer Datentypen genutzt werden soll und wenn die Genauigkeit von Gleitkomma-Rechnungen beurteilt werden soll. Eine detailliertere Beschreibung von arithmetischen Operationen enthält z.B. das Buch von Spaniol [Spa76].

2.1.1 Bitvektoren

Die meisten Befehlssätze unterstützen Operationen auf Bitvektoren begrenzter Länge (häufig: ein Wort). Wir wollen hier mit der Behandlung von Schiebeoperationen beginnen.

2.1.1.1 Schiebeoperationen

Bei den Schiebeoperationen ist zwischen den sog. logischen und den arithmetischen Schiebeoperationen zu unterscheiden. Die folgende Liste zeigt die Definition der vier von uns benutzten Schiebeoperationen *shift right logical, shift left logical, shift right arithmetical, shift left arithmetical* jeweils eine Stelle:

(2.1)	`srl(a)`	=	`'0' & a (a'left downto 1)`
(2.2)	`sll(a)`	=	`a (a'left-1 downto 0) & '0'`
(2.3)	`sra(a)`	=	`a (a'left) & a (a'left downto 1)`
(2.4)	`sla(a)`	=	`a (a'left) & a (a'left-2 downto 0) & '0'`

Schiebeoperationen um n können durch die n-malige Anwendung der Schiebeoperation um eine Stelle erklärt werden.

2.1.1.2 Weitere Operationen

Neben den Schiebe-Operationen unterstützen Rechner häufig auch Einzelbit-Operationen wie z.B. "Test Bit i", "Set Bit i", usw. Weiterhin können Bitvektoren häufig auch kopiert, nach Mustern durchsucht und gelöscht werden.

2.1.2 Natürliche Zahlen

Als ersten Zahlendatentyp wollen wir die natürlichen Zahlen behandeln.

Die folgende Funktion beschreibt die übliche Interpretation von Bitvektoren als natürliche Zahlen:

$$\texttt{nat(a)} = \sum_{i=0}^{a'left} a_i * 2^i \quad (2.5)$$

Dementsprechend stellt `"1000"` also beispielsweise eine 8 dar.

Übungsaufgaben:

- Zeigen Sie, dass die folgende Aussage gilt: `nat(srl(a))` $\rightarrow$ `nat(a) / 2` für alle Bitvektoren `a`. Dabei sei '/' die ganzzahlige Division.
- Unter welcher Vorraussetzung gilt `nat(sll(a))` = `nat(a) * 2`?

2.1.2.1 Addition

Diese Operation liefert die Summe natürlicher Zahlen, soweit dies aufgrund der üblicherweise festen Datenwortlänge möglich ist.

Im folgenden wollen wir annehmen, dass die Argumente der Grundrechenoperation `+` durch Bitvektoren $a = (a_{n-1}, .., a_0)$ und $b = (b_{n-1}, .., b_0)$ repräsentiert sind.

<u>Überläufe</u>

Wir wollen untersuchen, wie man erkennen kann, ob das Ergebnis einer Addition außerhalb des darstellbaren Zahlenbereichs liegt, also ob das Ergebnis der Operation mit einem Bitvektor $f = (f_{n-1}, .., f_0)$ der Länge n dargestellt werden kann.

Wir möchten gerne wissen: wann ist

$$nat(a) + nat(b) \quad \geq \quad 2^n \tag{2.6}$$

(wobei 2^n die erste nicht mehr durch f darstellbare Zahl ist) ? Wir möchten also wissen, ob die folgende Ungleichung gilt:

$$\sum_{i=0}^{n-1} a_i * 2^i + \sum_{i=0}^{n-1} b_i * 2^i \quad \geq \quad 2^n \tag{2.7}$$

Dies ist gerade dann der Fall, wenn bei der Addition ein Übertrag in die Stelle n hinein entsteht, also der Übertrag $c_n =' 1'$ ist. Bei der Addition **natürlicher** Zahlen ist der Übertrag c_n in die nächste Stelle also gleich dem Überlauf, den wir mit cf_+ bezeichen wollen.

Für c_n gilt:

$$\begin{aligned}
cf_+ = c_n &= (a_{n-1}b_{n-1}) \vee ((a_{n-1} \text{ xor } b_{n-1})c_{n-1}) \\
&= (a_{n-1}b_{n-1}) \vee ((a_{n-1} \vee b_{n-1})c_{n-1}) \\
&= (a_{n-1}b_{n-1}) \vee (a_{n-1}c_{n-1}) \vee (b_{n-1}c_{n-1})
\end{aligned}$$

Dabei haben wir davon Gebrauch gemacht, dass wir *xor* durch ein normales "oder" ersetzen können, weil der erste Term für $a_{n-1} = b_{n-1} = 1$ ohnehin eine '1' liefert.

Diese Gleichung ist jedoch in der Praxis vielfach nicht anwendbar, weil der interne Übertrag c_{n-1} meist nur innerhalb des Addierers verfügbar ist und auch auf Maschinen-Sprachebene und in höheren Programmiersprachen ist der Wert von c_{n-1} nicht abfragbar. Wir versuchen daher, ihn durch einen verfügbaren Wert zu ersetzen. Dazu analysieren wird die KV-Diagramme nach Abb. 2.2.

Daraus folgt, dass wir c_n auch darstellen können als:

$$c_n \quad = \quad (a_{n-1}b_{n-1}) \vee (a_{n-1}\overline{f_{n-1}}) \vee (b_{n-1}\overline{f_{n-1}})$$

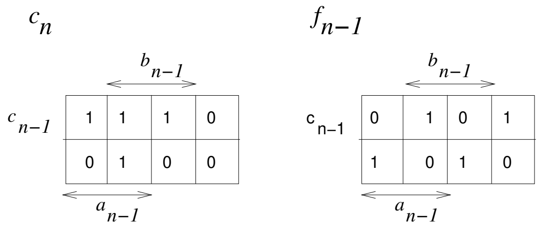


Abbildung 2.2: c_n und f_{n-1}

Der Wert von $\overline{f_{n-1}}$ kann meist abgefragt werden, sowohl in Schaltungen als auch in Programmiersprachen. Bei letzteren genügt ein Test "< 0" bei Interpretation des Ergebnisses als Zweierkomplementzahl.

Sättigungsarithmetik

Im Falle eines Überlaufs liefern Rechner meist ein Ergebnis nach der sog. *wrap around*-Arithmetik. Dies bedeutet, dass im Ergebnis nicht darstellbare Überträge einfach weggelassen werden. Eine Addition `a+b = "1000" + "1000"` liefert also den Wert `"0000"` (siehe Abb. 2.1).

	wrap-around-Arithmetik	Sättigungsarithmetik
`a`	`"1000"`	`"1000"`
`b`	`"1000"`	`"1000"`
`a+b`	`"0000"`	`"1111"`

Tabelle 2.1: Ergebnisse der Addition natürlicher Zahlen bei Bereichsüberlauf

Würden diese Vektoren z.B. Helligkeiten darstellen, so ergäbe die Addition halbheller Werte bei *wrap-around*-Arithmetik einen schwarzen Wert. Bei Audio- und Videoanwendungen ist es bei Bereichsüberschreitungen meist besser, den größten darstellbaren Wert als Ergebnis abzuliefern. In dem o.a. Fall würde also das Ergebnis `"1111"` abgeliefert werden, der Mittelwert wäre `"0111"`, also nicht 'ganz so falsch' wie `"0000"` . Gewisse Fehler werden bei der (verlustbehafteten) Datenreduktion von Audio- und Videodaten ohnehin zugelassen.

Entsprechend ist es besser, bei Bereichsunterschreitungen den kleinsten darstellbaren Wert abzuliefern, bei natürlichen Zahlen also `"0000"` . Eine Arithmetik mit diesem Verhalten heißt **Sättigungsarithmetik**. Diese Arithmetik verhält sich auch bei den übrigen Rechenoperationen entsprechend. Auf eine solche Arithmetik kann bei vielen Prozessoren für die digitale Signalverarbeitung (DSPs) umgeschaltet werden.

2.1.2.2 Subtraktion

Diese Operation liefert die Differenz natürlicher Zahlen, soweit dies aufgrund der üblicherweise festen Datenwortlänge möglich ist.

Überläufe

Bei der Subtraktion entsteht ein Überlauf cf_-, falls $a_{n-1} = 0$ ist und $b_{n-1} = 1$ oder $c_{n-1} = 1$ sind, sowie im Fall $a_{n-1} = 1$ und sowohl b_{n-1} als auch c_{n-1} sind gleich '1'. Das KV-Diagramm nach Abb. 2.3 (links) zeigt die Situation.

Gemäß linkem Teil der Tabelle gilt also für cf_-:

$$cf_- \; := \; (\overline{a_{n-1}} b_{n-1}) \vee (b_{n-1} c_{n-1}) \vee (\overline{a_{n-1}} c_{n-1})$$

Wieder entsteht das Problem, dass c_{n-1} meist nicht zugreifbar ist. Wir versuchen daher, statt c_{n-1} wieder f_{n-1} zu verwenden. f_{n-1} ist gleich '1', falls die drei beteiligten Variablen insgesamt eine ungerade Anzahl

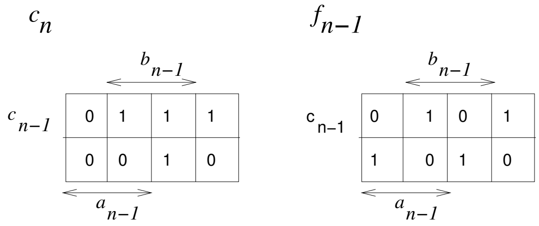


Abbildung 2.3: c_n und f_{n-1}

von Einsen besitzen (siehe rechten Teil des obigen KV-Diagramms). Unter Ausnutzung von f_{n-1} können wir cf_- schreiben als:

$$cf_- : \quad = \quad (\overline{a_{n-1}} b_{n-1}) \vee (b_{n-1} f_{n-1}) \vee (\overline{a_{n-1}} f_{n-1})$$

Damit ist das Ziel wieder erreicht.

2.1.2.3 Größenvergleich

Viele Rechner hinterlegen als Ergebnis einer Subtraktion im sog. Condition-Code Register eine Anzeige des Ergebnisses der Subtraktion. Insbesondere wird meist angezeigt, ob das Ergebnis Null und ob es negativ war. Dies leisten die Anzeigen *zf* und *sf*, die wie folgt definiert sind:

Def.: $zf =' 1' \iff \forall i \in [0..n-1] : f_i =' 0'$

Def.: $sf =' 1' \iff f_{n-1} =' 1'$

Für Vergleichoperationen mit natürlichen Zahlen ergeben die in Tabelle 2.2 gezeigten einfachen Umformungen die Formeln zur Berechnung von des Ergebnisses aus den Werten von zf und dem oben definierten cf_-.

$$\begin{array}{lclcl}
a < b & \iff & (a-b) < 0 & \iff & cf_- = 1 \\
a \geq b & \iff & \neg((a-b) < 0) & \iff & cf_- = 0 \\
a > b & \iff & ((a-b \geq 0) \wedge (a-b) \neq 0) & \iff & (cf_- = 0) \wedge (zf = 0) \\
a \leq b & \iff & \neg(a > b) & \iff & (cf_- = 1) \vee (zf = 1)
\end{array}$$

Tabelle 2.2: Berechnung der Vergleichsergebnisse aus den Condition-Codes

2.1.2.4 Multiplikation

Diese Operation liefert das Produkt natürlicher Zahlen, soweit dies aufgrund der üblicherweise festen Datenwortlänge möglich ist. Auf Maschinensprachebene kann das Produkt meist einen Bitvektor dargestellt werden, welcher die doppelte Länge der Bitvektoren der Argumente hat. Der entsprechend vergrößerte Zahlenbereich steht in höheren Programmiersprechen meist nicht zur Verfügung.

2.1.2.5 Natürliche Zahlen in VHDL

Falls der Zahlenbereich des VHDL-Datentyps `natural` ausreicht, kann `nat` mit der folgenden VHDL-Funktion berechnet werden :

```
function nat (a: bit_vector) return natural is
 variable s : natural;
```

```
begin
 s:= 0;
 for i in a'range loop -- absteigender
  s := s + s;          -- Schleifenindex
  if a(i) = '1' then s := s + 1;
  end if;
 end loop;
 return s;
end nat;
```

Diese Funktion kann mit Bitvektoren mit unterschiedlichem Indexbereich aufgerufen werden. Mit Hilfe des Attributs `'range` kann man im Rumpf der Funktion auf den Indexbereich des aktuellen Parameters Bezug nehmen, wie dies hier in der `for`-Schleife ausgenutzt wurde.

Die Funktion `nat` wird häufig benötigt, z.B. wenn Arrays indiziert werden. Beispiel:

`a (nat (b) )`

Dabei sei `a` ein beliebiges Array und `b` ein Bitvektor.

Um den Zugriff auf ein Array deutlich von einem Funktionsaufruf zu unterscheiden, werden wir gelegentlich eckige Klammern benutzen:

`a [nat(b)]`

Der Kürze halber werden wir gelegentlich den Aufruf der Funktion `nat` bei der Indizierung von Arrays fortlassen. Die Semantik kann jedoch stets als durch die ausführlichere Schreibweise definiert gelten.

Es hängt von dem jeweiligen Kontext ab, ob die Funktionen `nat` (und die weiter unten definierte Funktion `int`) explizit berechnet werden werden muss oder nicht:

- In einem VHDL-Simulations-System müssen die Funktionen aufgrund der Implementierung gemäß der strengen Typisierung in der Regel explizit berechnet werden.
- Die in VHDL beschriebene Hardware muss diese Funktionen nicht explizit "berechnen". Dies entspricht der Tatsache, dass derartige Funktionen auch als *cast*-Operatoren bezeichnet werden, die nicht in jedem Fall explizite Berechnungen auslösen, sondern nur in der Typüberprüfung benutzt werden.
- In der formalen Verifikation, z.B. von Compilern im Rahmen einer VDM-basierten Entwicklungsmethode [Sch83], sollte die explizite Angabe von Funktionen, die Bitvektoren interpretieren, hilfreich sein. Auch in diesem Fall werden die Funktionen nicht explizit berechnet.

Als Umkehrung der von `nat` wird auch eine Funktion benötigt, die Zahlen in Bitvektoren konvertiert. Die folgende Funktion leistet dies :

```
function bit_vect (arg, size : natural) return bit_vector is
 variable result : bit_vector (size-1 downto 0);
 variable i_val : natural := arg;
 begin
 if (size < 1) ... (Fehlerbehandlung) ;
 end if;
 for i in 0 to result'left loop
  if (i_val mod 2) = 0
   then result(i):= '0';
   else result(i) := '1';
  end if;
  i_val := i_val / 2;
 end loop;
 if not(i_val = 0) then warning ("vector truncated")
 end if;
```

```
 return result;
end bit_vect;
```

2.1.3 Ganze Zahlen

2.1.3.1 Interpretation der Bitvektoren

Bezüglich der ganzen Zahlen nehmen wir stets die Darstellung im **Zweierkomplement** an, weil dies heute in Rechnern praktisch ausschließlich verwendet wird. Bitvektoren $a = (a_n, ...a_0)$ werden bei dieser Darstellung gemäß der folgenden Formel als Zahlen interpretiert:

$$\texttt{int(a)} = \sum_{i=0}^{a'left-1} a_i * 2^i - 2^n * a_{a'left} \tag{2.8}$$

Dementsprechend stellt "1000" also beispielsweise eine -8 und "1001" eine -7 dar.

Übungsaufgaben:

- Zeigen Sie, dass die folgende Aussage gilt: `int(sra(a))` $\rightarrow$ `int(a) / 2` für alle Bitvektoren `a`. Dabei sei '/' die ganzzahlige Division.
- Unter welcher Vorraussetzung gilt `int(sla(a))` = `int(a) * 2`?

Sehr häufig benötigt man eine Operation, die einen Bitvektor mit n Elementen in einen solchen mit $m > n$ Elementen wandelt, wobei die dargestellte ganze Zahl dieselbe bleiben soll. Eine solche Operation nennt man Vorzeichenerweiterung (engl. `sign extend`). Diese Operation können wir definieren durch

```
function sign_ext(a: bit_vector,m: natural)return bit_vector is --Annahme:m>=a'left
variable s : bit_vector (m downto 0);
 begin
  s (a'left downto 0) := a;
  for i in m downto (a'left+1) loop
   s (i) := a(a'left);
  end loop;
 return s;
 end sign_ext;
```

Das Vorzeichenbit in `a` wird also repliziert.

Übungsaufgabe: Zeigen Sie, dass für alle Bitvektoren `a` und alle $m >$ `a'left` gilt:
`int(sign_ext(a,m)) = int(a)`.

Bei der Interpretation von Bitvektoren $a_{n-1}..a_0$ bzw. $b_{n-1}..b_0$ gemäß `int` als ganze Zahlen ist der in Abb. 2.4 gezeigte Zahlenbereich darstellbar.

2.1.3.2 Addition

Überläufe

Bei der Addition tritt ein Überlauf auf, wenn beide Operanden das gleiche Vorzeichen haben, das Ergebnis aber das entgegengesetzte Vorzeichen hat (siehe Tabelle 2.3).

Dies gilt auch für die Fälle $nat(a) = -2^{n-1}$ bzw. $nat(b) = -2^{n-1}$, die häufig eine Sonderbehandlung erfordern. Es gilt also (siehe auch: Bähring, Anhang.):

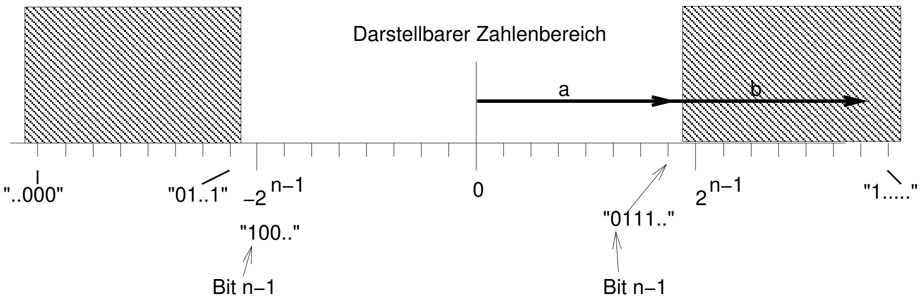


Abbildung 2.4: Darstellung ganzer Zahlen, Überlauf bei zwei positiven Zahlen

a_{n-1}	b_{n-1}	Überlauf unter der Bedingung
0	0	$f_{n-1} = 1$
0	1	nicht möglich
1	0	nicht möglich
1	1	$f_{n-1} = 0$

Tabelle 2.3: Bedingungen des Überlaufs bei der Addition von ganzen Zahlen

$$overflow\_add(a, b) = (a_{n-1} \equiv b_{n-1}) \wedge (a_{n-1} \text{ xor } f_{n-1})$$

Rechner speichern das Ergebnis dieser Berechnung nach Additionen meist in einem Bit des sog. Condition Code Registers. In einem folgenden Befehl kann dann mittels eines *branch if overflow*-Befehls bei einem Überlauf eine Verzweigung ausgelöst werden. Man beachte, dass der Rechner üblicherweise nicht 'weiß', ob er natürliche oder ganze Zahlen addiert und dass erst die 'richtige' Wahl des Verzweigungsbefehls eine bestimmte Interpretation zugrunde legt.

Etwas anders sind die Verhältnisse bei Sättigungsarithmetik: bei ganzen Zahlen sind die größten und die kleinsten Zahlen anders darzustellen als bei natürlichen Zahlen. Ist Sättigungsarithmetik sowohl für natürliche wie auch für ganze Zahlen realisiert, so werden separate Maschinenbefehle für die Grundoperationen beider Datentypen benötigt.

2.1.3.3 Subtraktion

Überläufe

Die Bedingungen bei der Subtraktion zeigt die Tabelle 2.4.

a_{n-1}	b_{n-1}	Überlauf unter der Bedingung
0	0	nicht möglich
0	1	$f_{n-1} = 1$
1	0	$f_{n-1} = 0$
1	1	nicht möglich

Tabelle 2.4: Bedingungen des Überlaufs bei der Subtraktion von ganzen Zahlen

Bei der Subtraktion tritt also ein Überlauf auf, wenn beide Operanden entgegengesetztes Vorzeichen haben und das Ergebnis $f_{n-1}..f_0$ ein anderes Vorzeichen als der erste Operand hat:

$$overflow\_sub(a, b) = (a_{n-1} \text{ xor } b_{n-1}) \wedge (a_{n-1} \text{ xor } f_{n-1})$$

2.1.3.4 Größenvergleich

Für Zahlen in Zweierkomplementdarstellung gilt: wegen

$overflow\_sub(a,b) = (a_{n-1} \text{ xor } b_{n-1}) \wedge (a_{n-1} \text{ xor } sf) = (\overline{a_{n-1}} \wedge b_{n-1} \wedge sf) \vee (a_{n-1} \wedge \overline{b_{n-1}} \wedge \overline{sf})$

folgt:

$$\begin{aligned} sf = 1 \quad &\Longleftrightarrow \quad overflow\_sub = \overline{a_{n-1}} \wedge b_{n-1} \\ sf = 0 \quad &\Longleftrightarrow \quad overflow\_sub = a_{n-1} \wedge \overline{b_{n-1}} \end{aligned}$$

Daraus ergibt sich die folgende Tabelle 2.5.

overflow_sub	sf	Kommentar	Ergebnis
0	0	kein Überlauf, F positiv, $0 \leq a - b \leq 2^{n-1} - 1$	$a \geq b$
0	1	kein Überlauf, F negativ, $-2^{n-1} \leq a - b < 0$	$a < b$
1	0	$(a_{n-1} = 1) \wedge (b_{n-1} = 0)$: a negativ, b positiv	$a < b$
1	1	$(a_{n-1} = 0) \wedge (b_{n-1} = 1)$: a positiv, b negativ	$a > b$

Tabelle 2.5: Bedingungen für Überläufe

Daraus ergeben sich folgende Beziehungen:

$$\begin{array}{llllll} a < b & \Longleftrightarrow & (overflow\_sub \text{ xor } sf) & & \\ a \leq b & \Longleftrightarrow & (a < b) \vee (a = b) & \Longleftrightarrow & (overflow\_sub \text{ xor } sf) \vee zf \\ a > b & \Longleftrightarrow & \neg(a \leq b) & \Longleftrightarrow & \neg((overflow\_sub \text{ xor } sf) \vee zf) \\ a \geq b & \Longleftrightarrow & \neg(a < b) & \Longleftrightarrow & overflow\_sub \equiv sf \end{array}$$

Diese Beziehungen werden vielfach benutzt, um anhand der Werte im Condition-Code Register Verzweigungsbedingungen abzutesten. Insbesondere ist zu beachten, dass die Größenvergleiche bei vorzeichenbehafteten Zahlen wegen der möglichen Überläufe nicht einfach durch Subtraktion und Vorzeichentest des Ergebnisses erfolgen dürfen!

2.1.3.5 Multiplikation

Diese Operation liefert das Produkt ganzer Zahlen, soweit dies aufgrund der üblicherweise festen Datenwortlänge möglich ist. Auf Maschinensprachebene kann das Produkt meist einen Bitvektor dargestellt werden, welcher die doppelte Länge der Bitvektoren der Argumente hat. Der entsprechend vergrößerte Zahlenbereich steht in höheren Programmiersprechen meist nicht zur Verfügung.

2.1.3.6 Integer in VHDL

Falls der Zahlenbereich des VHDL-Datentyps `integer` ausreicht, kann `int` mit der folgenden VHDL-Funktion berechnet werden :

```
function int (a: bit_vector) return integer is    -- Annahme: a'left > 0
constant t : natural := (2 ** (a'left));
 begin
  if a(a'left) = '0'     - - positive Zahl
  then return nat (a)
  else return (nat (a(a'left-1 downto 0)) -t)
  end if;
 end int;
```

In diesem Fall wurde mit dem Attribut `a'left` die "linke" Grenze des Indexbereichs des aktuellen Parameters abgefragt.

Man beachte, dass 2^n eventuell nicht mehr als VHDL-Zahlentyp darstellbar ist. Sofern 2^n gerade die erste nicht mehr darstellbare Zahl ist, kann es helfen, den Wert $-(2)^{n-1} + nat(a(a'left - 1 \text{ downto } 0)) - (2)^{n-1}$

zu berechnen. Im allgemeinen Fall wird man jedoch nicht jeden (langen) Bitstring in die Integer-Darstellung eines bestimmten VHDL-Simulationsystems konvertieren können.

2.1.4 Gleitkomma-Zahlen

2.1.4.1 Formate

Gleitkomma-Zahlen werden im wissenschaftlich-technischen Bereich insbesondere bei großen Zahlenbereichen vielfach benötigt. In Programmiersprachen stehen sie häufig als Datentyp *REAL* zur Verfügung. Diese Bezeichnung ist aber irreführend, denn die verfügbare Gleitkomma-Arithmetik erlaubt höchstens die näherungsweise Darstellung von reellen Zahlen. Zwecks besserer Einschätzung der Grenzen und Möglichkeiten üblicher Gleitkomma-Darstellungen wollen wir hier v.a. auf die Darstellung nach IEEE-Standards[1] eingehen[2].

Die Darstellung von Gleitkomma-Zahlen Z basiert auf der im technischen Bereich üblichen **halblogarithmischen** Darstellung, deren Mantisse und Exponent als Bitvektoren kodiert werden:

$$\begin{gathered} Z = (-1)^{VZ_M} * M * 2^{E'} \\ M \text{ : Betrag der Mantisse} \\ VZ_M\text{: Vorzeichen der Mantisse} \\ E' = (VZ_E, E)\text{: Exponent} \end{gathered}$$

Übliche Formate sind die folgenden:

a) | VZ_M | VZ_E | E | MANTISSENBETRAG M |
|---|---|---|---|

Mit: VZ_M = Vorzeichen zu M, VZ_E = Vorzeichen zu E,

b) | VZ_M | CHARAKTERISTIK C | MANTISSENBETRAG M |
|---|---|---|

Die **Charakteristik** wird berechnet, indem der Betrag des kleinsten gewünschten Exponenten auf alle Exponenten addiert und die resultierende Zahl in Bitvektordarstellung einer natürlichen Zahl gespeichert wird. Dies erlaubt praktisch ein Arbeiten mit ausschließlich positiven Exponenten. Es bewirkt eine Vereinfachung vieler Algorithmen. Erst bei Wandlung in die externe Darstellung wird die addierte Zahl wieder abgezogen.

Die Benutzung der Charakteristik hat weiter den angenehmen Effekt, dass betragmäßige Größenvergleichen für Gleitkommazahlen mit den entsprechenden Befehlen für natürliche oder 2er-Komplement-Zahlen ausgeführt werden können (siehe Übungsaufgabe).

Übungsaufgabe: Sei $abs(a)$ definiert als $abs(a)$ = `'0' & a(a'left-1 downto 0)`.
Zeigen Sie, dass $int(abs(s)) < int(abs(t)) \iff gk(abs(s)) < gk(abs(t))$ gilt.

Mantissen sind in der Regel normalisiert, z.B. auf den Bereich [1..2). Die Bitvektor-Darstellung vollständig gespeicherter Mantissen beginnt demnach stets mit einer Eins:

$M \in$ {"10...000", ..., "11..111"}

Das erste Vektorelement der Mantisse wird häufig nicht explizit gespeichert, da es immer ='1' ist (sog. *hidden bit*). Man kann somit die Genauigkeit um 1 Bit steigern.

Praktisch alle neueren Rechner realisieren die Formate nach den IEEE-Standards IEEE 754 und IEEE 854. Letzterer erlaubt im Gegensatz zum ersten auch die Basis 10. Wir behandeln hier nur den Standard IEEE 754.

Formate nach dem IEEE 754-Standard

<u>32-Bit-Format</u>

[1]Gesprochen: ei-tripple-i.
[2]Quelle: Goldberg [Gol91].

31	30 23	22 0
VZ_M	CHARAKTERISTIK C	MANTISSENBETRAG* M

$*$: **keine** Abspeicherung des verdeckten Bits.

64-Bit-Format

63	62 52	51 0
VZ_M	...CHARAKTERISTIK... C	...MANTISSENBETRAG*... M

$*$: **keine** Abspeicherung des verdeckten Bits.

80-Bit-Format

79	78 64	63 0
VZ_M	......CHARAKTERISTIK...... C	...MANTISSENBETRAG*...... M

$*$: **Abspeicherung** des verdeckten Bits.

Konventionen:

Die Bitvektoren werden gemäß folgender Funktionen als Werte interpretiert:

$gk_{32}(VZ, C, M) = sign(VZ) * 2^{nat(C)-127} * fract(`1`\&M)$
$gk_{64}(VZ, C, M) = sign(VZ) * 2^{nat(C)-1023} * fract(`1`\&M)$
$gk_{80}(VZ, C, M) = sign(VZ) * 2^{nat(C)-16383} * fract(M)$

Darin bedeuten:

gk_{xx}	Funktionen, die Bitvektoren im IEEE xx-Bit Format in Gleitkomma-Zahlen (*reals*) konvertieren (sog. Interpretationsfunktionen).
$sign$	$sign(X)$ = (if X='0' then return +1 else return -1)
nat	Die schon weiter vorn benutzte Funktion, die Bitvektoren in natürliche Zahlen konvertiert.
$fract$	Eine Funktion, die ihr Argument in eine reelle Zahl im Intervall [1..2) konvertiert: $frac(X) = \sum_{i=0}^{X'left} nat(X_i) * 2^{-X'left+i}$

Beispiel:

gk_{32} ('0',"10000000","1100...") $= +1 * 2^{128-127} * 1.75 = 3.5$

Die Definition der Funktionen gk enthält allerdings noch nicht die im folgenden beschriebenen Sonderfälle.

2.1.4.2 Sonderfälle

1. **Unendlich**

   $gk_{xx}(VZ, "11..11", "00...00") = \pm\infty$

   Motivation: Bei Überläufen ist die Rückgabe eines separaten Werts ∞ besser, als Rückgabe der größten darstellbaren Zahl. Beispiel:

   $\sqrt{x^2 + y^2}$ mit Überlauf bei x^2: Ergebnis $= \infty$ statt normaler Zahl.

Ausdruck	Ergebnis ($\infty \neq x > 0$)
$+x/0$	$+\infty$
$-x/0$	$-\infty$
$+x + (+\infty)$	$+\infty$
$+x - (+\infty)$	$-\infty$
$+x/(\pm\infty)$	0
$0/0$	NaN (s.u.)

2. **Null**

   0 ist nicht als normalisierte Zahl darstellbar, muss also gesondert behandelt werden. Es gilt die folgende Interpretation:

   $gk_{xx}(VZ, "00..00", "00..00") = 0$ für $xx \in \{32, 64, 80\}, VZ \in \{'0', '1'\}$

Die Unterscheidung zwischen +0 und -0 erreicht, dass $1/(1/x) = x$ auch für $x = \pm\infty$ gilt. Die Bitvektoren werden intern als $0 \vee (\epsilon > 0)$ bzw. $0 \vee (\epsilon < 0)$ behandelt.

Beispiele zur Behandlung der Null:

Ausdruck	Ergebnis
$+0 = -0$	true
$3 * (+0)$	+0
$+0/-3$	-0

3. **Nicht-normalisierte Zahlen**

Ein Beispiel der als normalisierte Zahlen darstellbaren reellen Zahlen liefert die Abbildung 2.5. In diesem Fall ist ein Exponentenbereich von $[-1..2]$ angenommen worden.

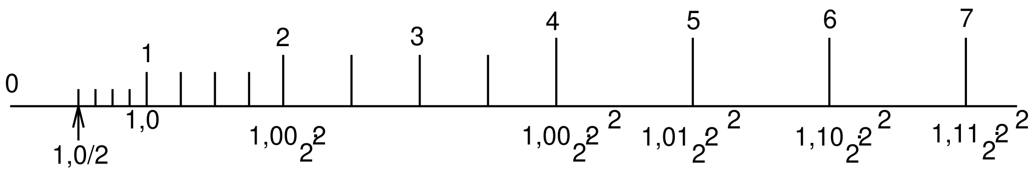


Abbildung 2.5: Normalisiert darstellbare Zahlen

Ein Problem liegt darin, dass der Abstand zwischen darstellbaren Zahlen in der Nähe der Null wieder zunimmt. Dies kann zu überraschendem Verhalten von Programmen führen. Beispielsweise kann die Anweisung

```
IF x ≠ y THEN z := 1/(x-y)
```

zur Division durch Null führen, denn für `x=1/2` und `y=5/8` ist `x ≠ y`, aber `x - y` aufgrund der Rundung 0.

Um dies zu vermeiden, erlaubt man wie in Abbildung 2.6 nicht normalisierte Zahlen. Für diese beginnt der Mantissenvektor auch bei expliziter Darstellung des verdeckten Bits nicht mit einer '1'.

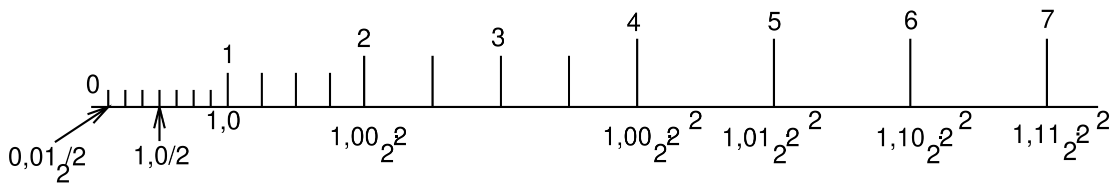


Abbildung 2.6: Nicht-normalisiert darstellbare Zahlen

Das Auftreten nicht-normalisierter Zahlen wird auch als *gradual underflow* bezeichnet. Nicht-normalisierte Zahlen werden nach IEEE-Norm gemäß der folgenden gesonderten Interpretation behandelt:

$gk_{xx}(VZ, "00..00", M) = sign(VZ) * 2^{-yy} * frac('0'\&M)$ für $M \neq$ "0...0"
für $(xx, yy) \in \{(32, 126), (64, 1022), (80, 16382)\}, VZ \in \{'0', '1'\}$

Das für die Charakteristik vorgesehene Feld besteht also aus lauter Nullen, das Mantissenfeld enthält die Mantisse und der Exponent ist fest.

Nicht-normalisierte Zahlen erfordern in den meisten Grundoperationen eine gesonderte Behandlung und damit Hardwareaufwand. Einige Prozessoren, die IEEE-Kompatibilität für sich in Anspruch nehmen, lösen für diese Zahlen daher einen Interrupt aus und erwarten eine Sonderfall-Behandlung in Software. Programme mit kleinen Zahlen laufen dann langsamer!

4. **Not-a-number**

Falls eine Rechenoperation für bestimmte Argumente nicht definiert ist, empfiehlt sich die Rückgabe eines speziellen Wertes. Wenn diese Rechenoperation ohnehin redundant war, wird so keine unnötige Fehlermeldung erzeugt. Redundante Rechenoperationen können durchaus vorkommen, z.B. infolge Pipelinings.

Der IEEE-Standard sieht hierfür den speziellen Wert *not-a-number* (NaN) vor. NaN wird als Spezialfall bei folgender Kodierung erzeugt:

$gk_{xx}(VZ, "11..11", M) = NaN$ für $M \neq$ "00..000"

NaN wird z.B. in folgenden Fällen erzeugt:

Operator	Ausdruck	Ergebnis
+	$(-\infty) + (+\infty)$	NaN
*	$0 * \infty$	NaN
/	$0/0, \infty/\infty$	NaN
REM (Rest)	x REM $0, \infty$ REM y	NaN
$\sqrt{x}$	$x < 0$	NaN

Beachte:
$\forall x : NaN \neq x$
$\rightarrow$: $x \neq x$ falls gk(x)=NaN!
$\rightarrow$: Die Zahlen sind nur noch partiell geordnet!
$\rightarrow$: $(x > y) \neq \neg(x \leq y)$

Ob NaN in einer konkreten Programmiersprache als Wert existiert, ist allerdings unklar.

2.1.4.3 Gleitkomma-Zahlen: System-Aspekte

Nachdem wir uns mit dem beschäftigt haben, was bei der Realisierung einzelner Gleitkomma-Maschinenbefehle zu beachten ist, wenden wir uns nun der Frage zu, wie die Gleitkomma-Arithmetik in höheren Programmiersprachen einzusetzen ist. Generell gibt es das Problem, dass der IEEE-Standard keinerlei Aussagen über die Unterstützung in höheren Programmiersprachen macht und auch die mögliche Unterstützung bei Verabschiedung des Standards nicht bedacht worden ist. Der Grund dafür: unter den Gleitkomma-Experten, die den Standard vorbereitet haben, gab es keinen Compiler-Spezialisten!

Wir betrachten hier vor allem betrachten wir die Frage, mit welcher Genauigkeit Zwischenrechnungen ausgeführt werden sollen. Entsprechende Überlegungen müssen v.a. beim Bau von Compilern bedacht werden.

Es gibt zunächst zwei intuitive Ansätze, die aber zu unakzeptablen Ergebnissen führen:

1. Man nehme für alle Zwischenrechnungen die maximal verfügbare Genauigkeit.

   Dies führt zu unerwarteten Ergebnissen. Beispiel:

   Sei `q` eine Variable einfacher Genauigkeit (32 Bit). Dann führt die Sequenz

   ```
   q :=3.0/7.0; print(q=(3.0/7.0))
   ```

   zum Ausdruck von `false`, denn bei der Zuweisung von `3.0/7.0` zu `q` gehen Mantissenstellen verloren. Wird der Vergleich (als "Zwischenrechnung") mit doppelter Genauigkeit ausgeführt, so müssen Mantissenstellen von `q` mit Nullen oder Einsen aufgefüllt werden.

2. Man nehme für alle Zwischenrechnungen das Maximum der Genauigkeiten der Argumente.

   Dies führt zu unnötigem Verlust von im Prinzip bekannter Information. Beispiel:

   Seien `x,y` Variablen einfacher und `dx` eine Variable doppelter Genauigkeit. Dann würde die Subtraktion in

   ```
   dx := x - y
   ```

   mit einfacher Genauigkeit erfolgen und `dx` könnten im Prinzip bekannte Mantissenstellen nicht zugewiesen werden.

Zur Lösung des Problems werden in den meisten Compilern zwei Durchläufe durch den Ausdrucksbaum benutzt. Die Aktionen sind dabei die folgenden:

1. Durchlauf von den Blättern zur Wurzel des Ausdrucksbaums. Für jede arithmetische Operation wird das Maximum der Genauigkeiten der Argumente gebildet. Die Genauigkeit bei einer Zuweisung ergibt sich aus der Genauigkeit der Zielvariablen. Für die Genauigkeit von Vergleichen empfiehlt sich in der Regel das Minimum der Genauigkeit der Argumente.

2. Aufgrund von Durchlauf 1 kann der Ausdrucksbaum inkonsistent sein. Im zweiten Durchlauf von der Wurzel zu den Blättern wird die Konsistenz hergestellt. Die Genauigkeit einer Operation wird reduziert, wenn ihr Ergebnis nicht in der bislang vorgesehenen Genauigkeit benötigt wird. Die Genauigkeit einer Operation wird erhöht, wenn ihr Ergebnis in größerer Genauigkeit als bislang vorgesehen benötigt wird.

Beispiel:

Abb. 2.7 zeigt, wie die Genauigkeit der Subtraktion an die der Zielvariablen angepaßt wird. `s` und `d` stehen dabei für die einfache und die doppelte Genauigkeit. Es werden zwei Konvertierungen von einfacher in doppelte Genauigkeit benötigt.

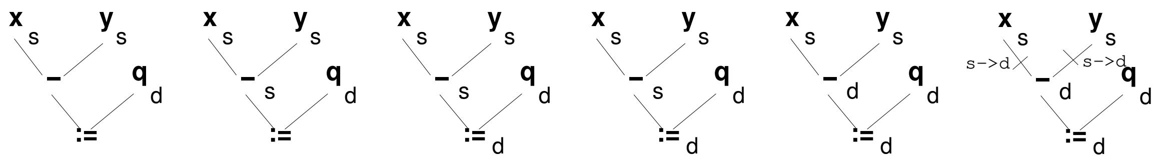


Abbildung 2.7: Erhöhung der Genauigkeit der Subtraktion

Folgende Probleme verbleiben auch bei dem angegebenen Verfahren:

- Die Genauigkeit ist nur im Kontext zu ermitteln. Ein Debugger, dem man die rechte Seite einer Anweisung übergibt, berechnet möglicherweise einen anderen Wert als der Compiler, der die Genauigkeit der Zielvariablen kennt.
- Der mögliche Fehler eines Ergebnisses ist unbekannt. Als Alternative wird von Kulisch (Univ. Karlsruhe) die Verwendung der **Intervallarithmetik** propagiert. Bei der Intervallarithmetik werden für alle Berechnungen die Intervallgrenzen der möglichen Werte betrachtet. Hierfür ist spezielle Hard- und Software entwickelt worden.

Unzulässige Optimierungen

Aufgrund der Ungenauigkeiten der Gleitkommaarithmetik sind viele zunächst korrekt erscheinende Compiler-"Optimierungen" falsch. Dazu einige Beispiele:

Ausdruck	unzulässige Optim.	Problem
$x/10.0$	$0.1 * x$	0.1 ¬ exakt darstellbar
$x * y - x * z$	$x * (y - z)$	falls $y \approx z$
$x + (y + z)$	$(x + y) + z$	Rundungsfehler
konstanter Ausdruck	result. Konstante	Flags werden ¬ gesetzt
gemeinsamer Ausdruck	Ref.auf 1.Berechn.	Rundungsmodus geändert ?
$eps := 1$; WHILE $eps > 0$ DO BEGIN $h := eps$; $eps := eps * 0.5$; END	$eps := 1$; WHILE $(eps + 1) > 1$ DO BEGIN $h := eps$; $eps := eps * 0.5$; END	

Zunächst scheint es zulässig zu sein, von beiden Argumenten des Vergleichs in der WHILE-Schleife jeweils 1 abzuziehen. Dennoch brechen die beiden Versionen der WHILE-Schleife an verschiedenen Stellen ab. In der linken Version wird *eps* bei 1 beginnend solange halbiert, bis es auf 0 abgerundet wird. Die gestrichelte Linie in Abb. 2.8 zeigt den Verlauf der Werte des Vergleichsoperanden *eps*. h speichert in diesem Fall den letzten von 0 verschiedenen Wert, 1/16. In der rechten Version wird *eps* solange halbiert, bis seine Addition zu 1 kein von 1 unterscheidbares Ergebnis liefert. Im Falle der Gleitkomma-Genauigkeit der Abb. 2.8 erhält man für *eps* die Folge 1, 1/2, 1/4, 1/8. Die durchgezogene Linie der Abbildung zeigt die Folge der Werte für den linken Vergleichsoperanden, $eps + 1$. Die Addition von 1/8 zu 1 verändert die 1 nicht mehr und h speichert den vorletzten Wert von *eps*, 1/4.

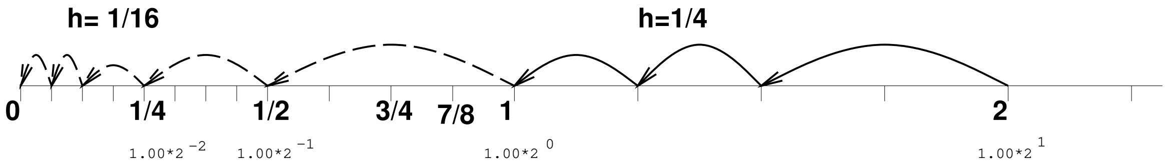


Abbildung 2.8: Werte der Vergleichsoperanden

Lediglich einige sehr einfache Optimierungen sind zulässig:

Ausdruck	optimierte Fassung
$x + y$	$y + x$
$2 * x$	$x + x$
$1 * x$	x
$x/2.0$	$x * 0.5$

2.2 Das Speichermodell

Wir betrachten wir als Nächstes das Speichermodell. Insgesamt realisieren Speicher den **Zustand** eines Rechensystems bzw. eines Programms.

2.2.1 Der Hauptspeicher

Das logische Verhalten des Hauptspeichers entspricht i.W. dem eines großen, adressierbaren Arrays. Überwiegend wird heute die Byte-Adressierung benutzt, d.h. jedes **Byte** des Hauptspeichers besitzt eine eigene, eindeutige Adresse. Größeren Datenstrukturen (wie z.B. Integern) sind dann mehrere Adressen zugeordnet, wobei zwischen der Zuordnung im *little-endian* und der Zuordnung im *big-endian*-System unterschieden werden kann (siehe unten).

Aus Performance-Gründen wird das Schreiben **einzelner** Bytes selbst bei Byte-Adressierung nicht immer unterstützt. Aus den gleichen Gründen wird häufig verlangt, dass größere Datenstrukturen an bestimmten Wortgrenzen ausgerichtet sind. So wird z.B. in der Regel verlangt, dass 32-Bit Integer an durch 4 teilbaren Adressen beginnen. Ähnliche *Alignment*-Beschränkungen gibt es meist auch für andere, durch die Maschine unterstützten Datentypen.

Ein weiterer, für die funktionale Sicht auf den Hauptspeicher wichtiger Parameter ist sicherlich seine maximale Größe. Historisch gesehen wurde von Rechnerarchitekten immer wieder die Zunahme der Anforderungen an die Größe des adressierbaren Speichers unterschätzt. Berühmte Beispiele sind die Beschränkungen auf 16-Bit Adressen bei der DEC PDP-11 und anderen Rechnern, auf 24-Bit Adressen bei IBM Großrechnern und die 640 kByte Grenze von MS-DOS. Insbesondere im Hinblick auf die Adressierung in großen Datenbanken findet gegenwärtig ein Übergang auf 64-Bit Adressen statt (Beispiele: DEC Alpha, SuperSPARC).

2.2.1.1 *Little endians* und *Big endians*

Die Numerierung der Bytes innerhalb von Worten wird in den verschiedenen Rechnern unterschiedlich gehandhabt. Es gibt zwei unterschiedliche Systeme:

1. *Little endian*: In diesem System erhält der am wenigsten signifikante Teil des Wortes die niedrigste Byteadresse.
2. *Big endian*: In diesem System erhält der signifikanteste Teil des Wortes die niedrigste Byteadresse.

Die Bezeichnung geht auf “Gulliver’s Reisen” und die darin beschriebene Frage, ob Eier mit dem schmalen oder mit dem dicken Ende zuerst zu essen sind, zurück (siehe [Coh81]).

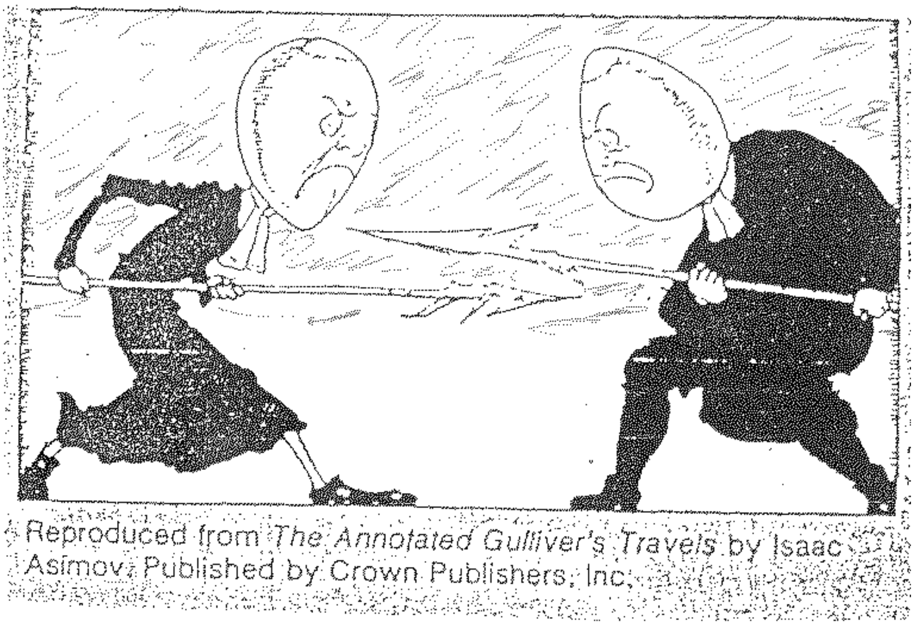


Abbildung 2.9: Streit zwischen *big endians* und *little endians* ©Crown Publishers, Inc.

Im folgenden betrachten wir die Auswirkungen dieser beiden Systeme in vier Fällen:

- **Bei der Speicherbelegung durch 32-Bit Integer**

  Die Zahl 258 = 256 + 2 würde in den beiden Systemen wie in Tabelle 2.6 abgelegt werden. Dabei entspricht "xx00" der Anfangsadresse der Zahl.

little endian		big endian	
Adresse	Wert	Adresse	Wert
"xx00"	2	"xx00"	0
"xx01"	1	"xx01"	0
"xx10"	0	"xx10"	1
"xx11"	0	"xx11"	2

Tabelle 2.6: Speicherbelegung durch eine 32-Bit Zahl

  Man erkennt, dass das Umschalten zwischen den beiden Systemen durch Invertieren der letzten beiden Adreßbits möglich ist.

- **Bei der Zeichenketten-Verarbeitung**

  Tabelle 2.7 zeigt, wie das Wort "Esel" bei den beiden Speicherbelegungsverfahren abgelegt wird.

little endian		big endian	
Adresse	Wert	Adresse	Wert
"xx00"	l	"xx00"	E
"xx01"	e	"xx01"	s
"xx10"	s	"xx10"	e
"xx11"	E	"xx11"	l

Tabelle 2.7: Speicherbelegung durch eine Zeichenkette

  Zu beachten sind bei der Konstruktion von Zeichenketten-Verarbeitungs-Routinen u.a. folgende Fragen:

  – Wie erfolgt das lexikalische Sortieren von Worten ?

  – Wie werden Zeichenketten an ihrem Ende erweitert ?
  – Wie werden Speicherdumps dargestellt, so dass Zeichenketten lesbar bleiben?
  – Wie wird jeweils sichergestellt, dass man nach Erhöhung eines Pointers auf einen String auf das nächste Zeichen zugreift ?

- **Bei der Numerierung von Bits in Einzelbit-Befehlen**

  Einzelne Bits innerhalb eines Wortes werden in den beiden System z.B. wie folgt bezeichnet:

  – Im little endian system:

MSB																															LSB
31																															0

  – Im big endian System:

MSB																															LSB
0																															31

- **Bei der seriellen Datenübertragung**

  Unabhängig von dem Speicherbelegungsverfahren spielt die Unterscheidung bei der seriellen Datenübertragung eine Rolle. Treffend wird hier die Unterscheidung zwischen *MSB travels first* und *LSB travels first* gebraucht (siehe Kap. 5).

Zur Verwirrung der Programmierer sind viele Maschinen inkonsistent entworfen: z.B. *little endian* für Integer und *big endian* für Gleitkommazahlen, u.s.w.

Die Wahl eines der beiden Systeme hat Auswirkungen v.a. beim Compilerbau, bei der Entwicklung von Bibliotheken sowie beim Datenaustausch zwischen Maschinen, die unterschiedliche Systeme benutzen. Im X-Window-System wird der Datenaustausch speziell geregelt: Keine Festlegung auf einen Standard, sondern Konvertierung nur dann, wenn eine Konvertierung beim Datenaustausch zwischen unterschiedlichen Maschinen notwendig ist.

Der Autor ist selbst ein *little endian*-Anhänger, da stets klar ist, was das am wenigsten signifikante Bit bzw. Byte ist. Beim *big endian*-System kommt man bei der Erweiterung von Adressen um Seitennummern oder Prozessidentifikatoren evtl. zu negativen Bit- oder Bytenummern. Im Fall von Zeichenketten muss das *little endian*-System allerdings durch geeignete abstrakte Zugriffsfunktionen gekapselt werden, um ein erwartetes Verhalten zu erreichen.

Wie in anderen Fällen, in denen eine Entscheidung zu treffen ist, umgeht man heute vielfach die Entscheidung und implementiert "sowohl als auch". Bei den RISC-Rechnern von MIPS kann man zwischen *little endian* und *big endian* umschalten.

Von der logischen Sicht auf den Hauptspeicher zu unterscheiden ist die Sicht auf die strukturelle Implementierung. Diese werden wir in Kapitel 4 vorstellen.

2.2.2 Registerspeicher

Neben den großen Hauptspeichern besitzen praktische alle Rechnerarchitekturen kleinere Registerspeicher. Diese Registerspeicher bieten Vorteile v.a. deshalb, weil sich Programme in gewissem Umfang **lokal** verhalten. Dies kann ausgenutzt werden, um in einem lokalen Auschnitt häufig genutzte Informationen in Registern zu halten. Es gelingt so sowohl ein schnellerer Zugriff auf die Informationen als auch eine Verkürzung der Adreßteile in den Maschinenbefehlen. Die in RISC-Architekturen häufig benutzte konstante Befehlswortlänge und die starke Fließbandverarbeitung werden so erst möglich.

Die Wahl der Größe derartiger Registerspeicher ist ein Kompromiß zwischen der Anzahl der für Register verfügbaren Befehlsbits und den gewünschten Programmausführungszeiten. Im Falle kleiner Registerspeicher können schnelle Architekturen nur durch eine besonders schnelle Speicherarchitektur realisiert werden. Aus diesem Grund entsteht für Architekturen mit kleinem Registersatz (z. B. den Intel x86-Architekturen) ein besonders großer Druck, einen schnellen Cache vorzusehen.

Weiter kann man noch zwischen Architekturen mit homogenem und solchen mit heterogenen Registersätzen unterscheiden. Bei homogenen Registersätzen besitzen alle Register dieselbe Funktionalität, sind also alle Register in den Befehle gleichermaßen verwendbar. Solche Registersätze erleichtern das Erstellen von Compilern, da diese dann nur Klassen von Registern betrachten müssen.

Heterogene Registersätze erlauben z.Tl. eine größere Effizienz der Architektur. Architekturen für die digitale Signalverarbeitung (engl. *digital signal processing*, DSP) erlauben häufig die schnelle Bearbeitung einer Multiplikations- und einer Additionsoperation durch einen *multiply/accumulate*-Befehl. Derartige Befehle beziehen sich häufig auf spezielle Register. Für die Benutzung allgemeiner Register fehlt z.Tl. der notwendige Platz im Befehl zur Angabe von Registernummern. Außerdem würde für einen solchen zeitkritischen Befehl möglicherweise auch die Taktperiode zu kurz sein. Aus Günden der Effizienz (Codedichte und Laufzeit) besitzen hier also heterogene Registersätze einen Vorteil. Aber selbst Architekturen, deren Registersätze weitgehend homogen sind, beinhalten Ausnahmen, wie z.B. die Verwendung eines der Register als Kellerzeiger (engl. *stack pointer*).

Beispiele:

1. Motorola 68000er-Prozessoren.

   Abb. 2.10 zeigt den Registersatz der Motorola 68000er-Prozessoren.

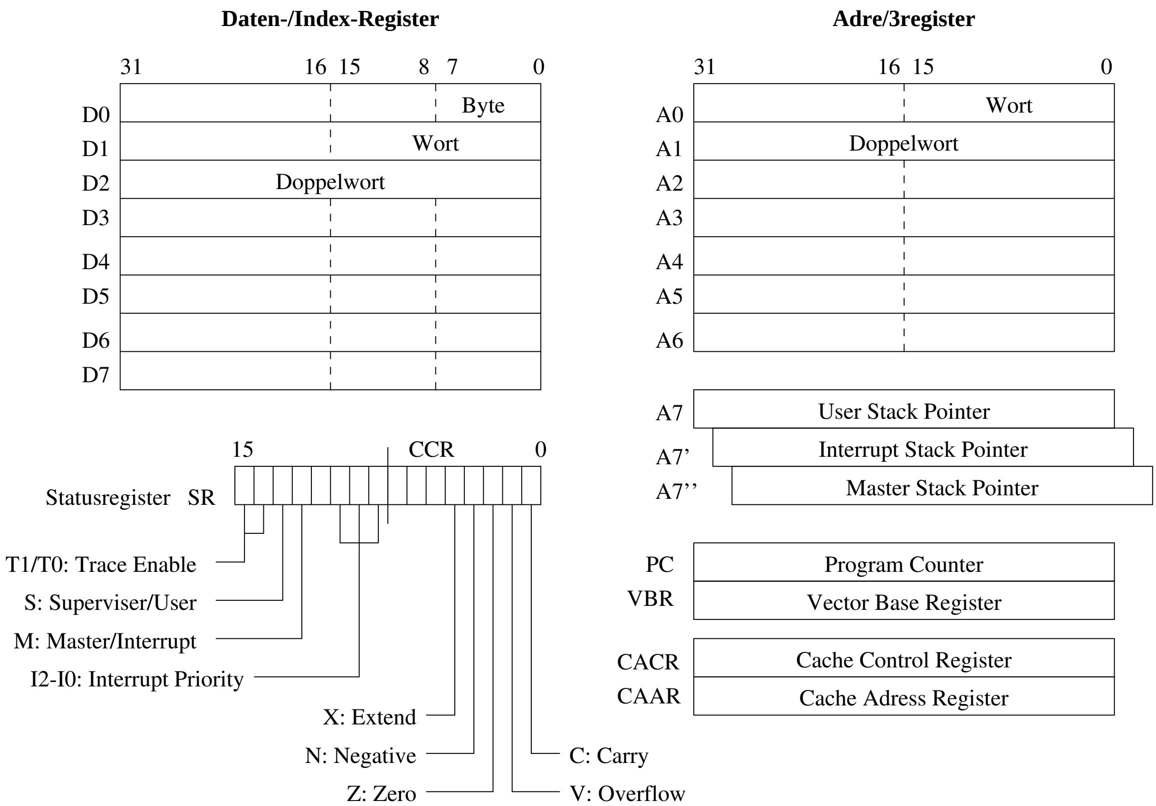


Abbildung 2.10: Registersatz der Motorola 680x0-Prozessoren

   Diese Prozessoren besitzen in jedem Prozessorzustand 16 Register, was für eine der ersten Mikroprozessorarchitekturen relativ viel ist. Die Register gelten als weitgehend homogen, wenngleich getrennte Adreß- und Datenregister vorhanden sind. Der Grund hierfür ist auch hier die Effizienz: in vielen Zusammenhängen ist so die Einsparung von jeweils einem Bit zur Adressierung der Register möglich, beispielsweise sind zur Angabe eines Adreßregister nur 3 Bit vorgesehen .

   Eine der Registernummern ist für ein Stackpointer-Register vorgesehen. Von diesen Stackpointer-Registern gibt es allerdings mehrere, von denen je nach Prozessorzustand eines ausgewählt wird. Auf diese Weise kann sichergestellt werden, dass Betriebssystem- und Interrupt-Routinen über dieses Register immer freien Speicher adressieren können (sofern solcher noch vorhanden ist).

   Zur Abspeicherung der in Abschnitt 2.1 berechneten Überlauf- und Übertragsbedingungen sind entsprechende Condition-Code-Register vorhanden. Von diesen Bedingungen abhängige Sprünge sind dadurch möglich.

2. MIPS R2000-8000

   Die MIPS-Architektur [HP96], die z.B. in Silicon Graphics Workstations eingesetzt wird, enthält 32 weitgehend homogen verwendbare Register. Als Ausnahme davon liefert ein Lesen unter der Registernummer 0 stets den Wert 0 und das Register 1 wird von Compilern für Pseudo-Assembler-Instruktionen als Hilfsregister freigehalten. Weiter gibt es zwei Register `HI` und `LO`, in denen Ergebnisse von Multiplikationen und Divisionen abgelegt werden. Es gibt keine Condition-Code-Register.

3. Intel 80x86

   Die 80x86-Prozessorfamilie besitzt einen relativ kleinen und inhomogenen Registersatz (siehe Abb. 2.11.

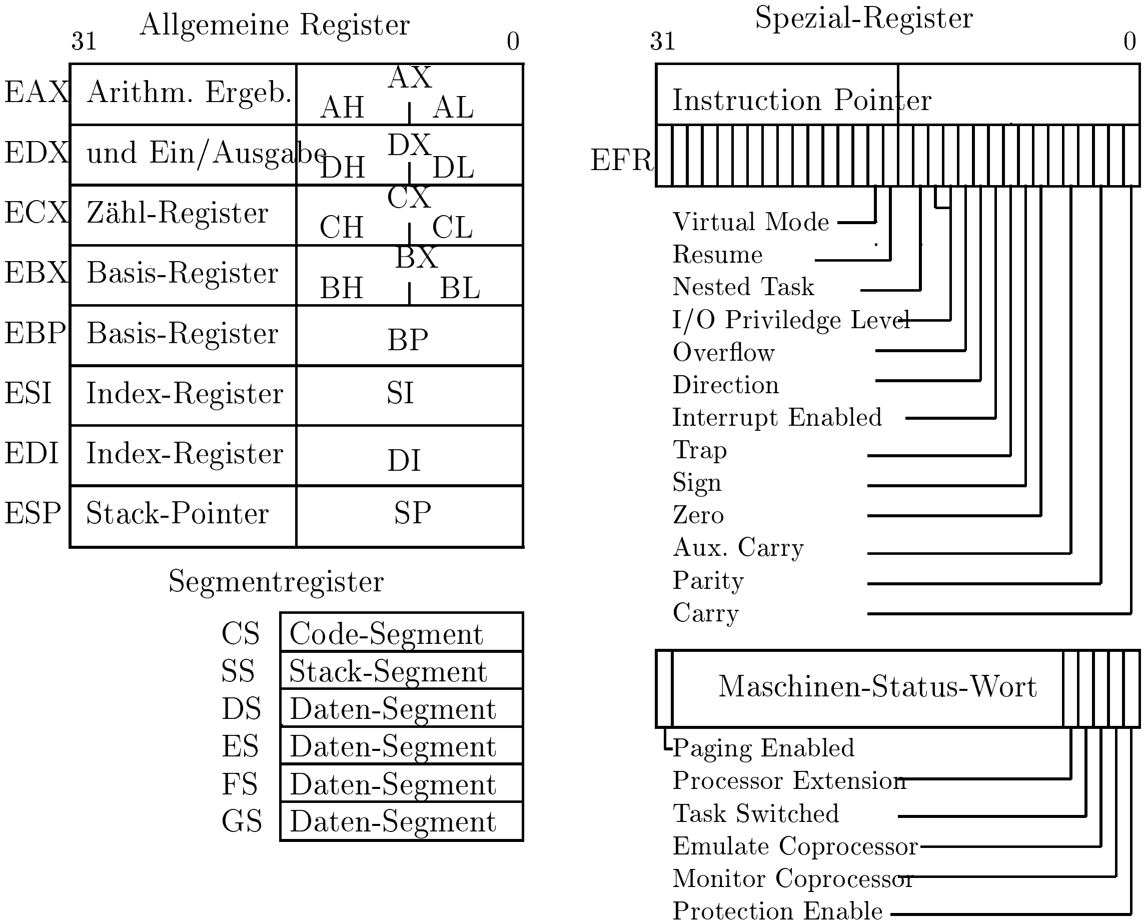


Abbildung 2.11: Registersatz der Intel-Prozessoren 80386 und 80486

2.3 Maschinenbefehle

2.3.1 Befehlsklassen

Zum externen Programmiermodell gehört neben den Speichern v.a. der Befehlssatz. Verfügbare Befehle können anhand der sog. **Befehlsgruppen** klassifiziert werden. Beispielsweise kann eine Einteilung in folgende Befehlsgruppen vorgenommen werden:

- Transferbefehle (LD, ST, MOVE, LEA)
- E/A-Befehle (IN, OUT)

- Arithmetische Befehle (ADD, SUB, MUL, DIV)
- Logische Befehle (AND, OR, NOT)
- Vergleichsbefehle (meist als Seiteneffekt von arithmetischen Befehlen auf das Condition-Code-Register)
- Bitfeld- und Flag-Befehle (→ Bähring Tab. 1.14-7)
- Schiebebefehle (Angabe von Richtung und Quelle für das Auffüllen)
- Sprungbefehle (→ Bähring Tab. 1.14-10, 1.14-11, 1.14-12)
- Kontrollbefehle (Disable Interrupt)
- Ununterbrechbare Befehle (z.B. Test-and-set (TAS))

2.3.2 Adressierungsarten

Speicherzellen werden wie bereits erwähnt über Speicher-**Adressen** angesprochen. Die Berechnung dieser Adressen kann durch die Maschinenbefehle unterschiedlich vorgeschrieben werden[3]. Bei der folgenden Darstellung der Möglichkeiten hierzu verwenden wir in Anlehnung an die Motorola 68000er Prozessorlinie folgende Bezeichnungen:

- AdrTeil : (Inhalt des) Adreßteils **im Befehl**
- Reg : (Inhalt des) Register-Feldes im Befehl
- IReg : (Inhalt des) Index-Register-Feldes im Befehl
- BReg : (Inhalt des) Basis-Register-Feldes im Befehl
- Speicher : Hauptspeicher, als Array modelliert
- D : Daten-Registerfile (oder allgemeines Registerfile), als Array modelliert
- Dx : Assemblernotation für "Datenregister x",
- A : Adreß-Registerfile, als Array modelliert
- Ax : Assemblernotation für "Adreßregister x",

Mit diesen Bezeichnungen kann die Adressierung eines einzelnen Operanden gemäß Abbildung 2.12 klassifiziert werden. Dabei werden Befehle nach der Anzahl der Zugriffe auf den Speicher eingeteilt.

Beispiele für die einzelnen Arten sind die folgenden[4]:

- Referenzstufe 0 (0 Zugriffe auf den Speicher)
  - Registeradressierung

Assembler-Notation	Semantik
CLR,3	D[3]:=0
LDPSW,3	D[3]:=PSW

  - unmittelbare Adressierung, immediate Adressierung
    Beispiel: LD-Befehl (# bedeutet **unmittelbare Adressierung**)

Assembler-Notation	Semantik
LD D3,#AdrTeil	D[3]:= AdrTeil

[3] vgl. Bähring Abschn. 1.15.

[4] Das Buch von Bähring enthält in den Abbildungen 1.15-3 bis 1.15-19 eine graphische Darstellung dieser Adressierungsarten.

- 0–stufige Speicheradressierung (Referenzstufe 0)
  - Register–Adressierung
  - unmittelbare Adressierung
- 1–stufige Speicheradressierung (Referenzstufe 1)
  - direkte Adressierung, absolute Adressierung
  - Register–indirekte Adressierung
  - indizierte Adressierung, Basisadressierung
  - Programmzaehler–relative Adressierung
- 2–stufige Speicheradressierung (Referenzstufe 2, indirekte Adressierung)
  - direkte Adressierung, absolute Adressierung
  - Register–indirekte Adressierung
  - indizierte Adressierung
  - Programmzaehler–relative Adressierung
- n–stufige Speicheradressierung (Referenzstufe n)

Abbildung 2.12: Adressierungsarten

- Referenzstufe 1, einstufige Adressierung
  - Direkte Adressierung, absolute Adressierung

Assembler-Notation	Semantik
LD D3,AdrTeil	D[3]:= Speicher[AdrTeil]

  - Register-Indirekte Adressierung

Assembler-Notation	Semantik
LD D3,(A4)	D[3]:= Speicher[A[4]]

    Varianten: Prä/Post- De/Increment zur Realisierung von PUSH und POP.
  - Relative Adressierung, Indizierte A., Basis-A.

Assembler-Notation	Semantik
LD D3,AdrTeil(IReg)	D[3]:= Speicher[AdrTeil+D[IReg]]

    Varianten:
    - ∗ Indizierte Adressierung
      AdrTeil kann aus dem vollen Adreßbereich sein, D ist evtl. kürzer. Bei ungleicher Länge von AdrTeil und D erfolgt eine Angleichung der Operandenlänge von "+", in der Regel ein Auffüllen mit Nullen (zero-extend), gelegentlich aber auch mit dem signifikantesten Bit (sign-extend).
    - ∗ Basisadressierung, Register-Relative Adressierung
      D[] kann aus dem vollen Adreßbereich sein, AdrTeil ist evtl. kürzer.
      Beispiel: IBM-370, 3090, ..: 32-Bit Register und 12-Bit AdrTeil.
    - ∗ Register-Relative Adressierung mit Index

Assembler-Notation	Semantik
LD D3,AdrTeil(IReg)(BReg)	D[3]:= Speicher[AdrTeil+D[IReg]+D[BReg]]

      Diese Adressierung ist grundlegend für die Adressierung innerhalb der gesamten IBM-370-Familie und deren Nachfolger (390 u.s.w.). Das Format der Maschinenbefehle, die zum größen Teil eine Länge von 32 Bit haben, ist das folgende:

Feld	Opcode	Reg	IReg	BReg	AdrTeil
Bits	31:24	23:20	19:16	15:12	11:0

* Programmzähler-Relative Adressierung

Assembler-Notation	Semantik
BRA $7FE	PC := PC + $7FE +i, mit i=0,1,2 oder 4

Die Konstante i ergibt sich dadurch, dass der Prozessor vor der Ausführung des Sprungs vorsorglich bereits den Programmzähler erhöht hat. Bemerkung: Vollständig PC-relative Programme sind im Speicher frei verschiebbar (relocatable). Dies wird gelegentlich für Stand-Alone-Programme (Programme ohne Betriebssystem-Unterstützung) ausgenutzt, z.B. für Speichertest-Programme, die vor dem Starten des Betriebssystems ausgeführt werden.

Konvention:

* relative Adressierung: vorzeichenbehaftete Distanz
* Basis-Adressierung: vorzeichenlose Distanz
* indizierte Adressierung: "kleiner" Wertebereich für das Register

- Referenzstufe 2, zweistufige Adressierung, indirekte Adressierung
  - Indirekte (absolute) Adressierung
    D[Reg] := Speicher[Speicher[AdrTeil]]
  - Indirekte Register-indirekte Adressierung
    D[Reg] := Speicher[Speicher[D[IReg]]]
  - Indirekt indizierte Adressierung, Vorindizierung
    Beispiel ohne BReg: D[Reg] := Speicher[Speicher[AdrTeil+D[IReg]]]
  - Indizierte indirekte Adressierung, Nachindizierung
    Beispiel (ohne BReg): D[Reg] := Speicher[Speicher[AdrTeil]+D[IReg]]
  - Indirekte Programmzähler-relative Adressierung

$$\begin{array}{lcl} D[Reg] & := & Speicher[Speicher[AdrTeil+PC]] \\ PC & := & Speicher[AdrTeil+PC+i] \end{array}$$

  Man beachte, dass bei der Adressierung von Befehlen über PC das spätere Holen des Befehls als zweiter Speicherzugriff zählt.

- Referenzstufen > 2

  Referenzstufen > 2 werden nur in Ausnahmefällen realisiert. Die fortgesetzte Interpretation des gelesenen Speicherwortes als Adresse des nächsten Speicherworts nennt man **Dereferenzieren**. Fortgesetztes Dereferenzieren ist u.a. zur Realisierung von PROLOG mittels der *Warren Abstract Machine* (WAM) wichtig [Kog91]. Bei der WAM wird die Anzahl des Dereferenzierens durch Kennzeichenbits im gelesenen Speicherwort bestimmt.

Die große Anzahl von Adressierungsarten wurde ursprünglich durch das Bemühen motiviert, möglichst kompakten Programmcode zu erzeugen. Mit dem Aufkommen der RISC-Architekturen wurde die Anzahl der Hardware-mäßig unterstützten Adressierungsarten deutlich reduziert. Bei RISC-Architekturen wurde damit der Möglichkeit der potentiell schnelleren Ausführung Priorität gegenüber der Kompaktheit des Programmcodes gegeben. Sofern Prozessoren Teil eines kompletten, auf einem Chip integrierten Systems sind (engl. *system-on-a-chip*), wird die für Speicher benötigte Chip-Fläche sehr wichtig. Für solche Systeme werden daher häufig keine RISC-Prozessoren verwendet.

2.3.3 n-Adreßmaschinen

Wir werden im folgenden mit kleinen Buchstaben die Adressen der entsprechenden, mit Großbuchstaben geschriebenen Variablen bezeichnen. Wenn `Speicher` der Name des Hauptspeichers ist, gilt also z.B. `Speicher[a] = A` usw.

Ein für Rechner benutztes Klassifikationsmerkmal besteht in der Angabe der Anzahl der Adressen bei 2-stelligen Operationen[5]. Es gibt die folgenden Fälle:

- 3-Adreßmaschinen, 3-Adreßbefehle

  3 vollständige Adressen, IBM-Bezeichnung: Befehle vom "Typ SSS"
  (d.h. Befehle der Form Speicher[a] := Speicher[b] $\circ$ Speicher[c])

  Beispiel:

  Wir betrachten die Zerlegung der Anweisung

  `D:= (A + B) * C;`

  Die Anweisung kann bei Verwendung von 3-Adreßbefehlen übersetzt werden in:

  ```
  ADD t1, a, b        - - Speicher[t1] := Speicher[a]  + Speicher[b]
  MULT d, t1, c       - - Speicher[d]  := Speicher[t1] * Speicher[c]
  ```

  Dabei ist `t1` die Adresse einer Hilfszelle und `ADD` und `MULT` sind Assemblerbefehle zur Addition bzw. Multiplikation. Wir nehmen an, dass der erste Parameter das Ziel des Ergebnisses beschreibt. Wir benötigen pro Befehl 3 Zugriffe auf den Speicher, insgesamt also 6 Speicherzugriffe.

  3-Adreßbefehle werden gelegentlich im Compilerbau als Zwischenschritt benutzt. Dabei werden alle arithmetischen Ausdrücke zunächst in 3-Adreß-Code zerlegt. Später werden dann die 3-Adreßbefehle auf die wirklich vorhandenen Befehle abgebildet.

  3-Adreßbefehle, in denen mit allen 3 Adressen der komplette Speicher adressiert werden kann, benötigen eine relativ große Befehlswortlänge.

  Beispiel:

  Unter der Annahme, dass 32 Bit zur Adressierung eines Operanden benötigt werden, dass die Befehlswortlänge ebenfalls 32 Bit beträgt und dass das erste Befehlswort in der Regel noch Platz bietet, Register (z.B. Indexregister) anzugeben, erhalten wir die Verhältnisse der folgenden Skizze. Insgesamt werden in diesem Fall 4∗32 = 128 Bit zur Kodierung eines einzigen 3-Adreßbefehls benötigt!

  $\leftarrow$ 32 Bit $\rightarrow$

Op-Code	weitere Informationen
a	
b	
c	

  Es gibt nun 3 Möglichkeiten der Reduktion der Befehlslänge:

  – die Überdeckung (Zieladresse = Quelladresse)
  – die Implizierung (Operand ist implizit, z.B. durch Opcode)
  – Kurzadressen (z.B. Basisregister und Displacement)

- 2-Adreßmaschinen, 2-Adreßbefehle

  Bei diesen wird die Überdeckung von Zieladresse und Quelladresse ausgenutzt. Es wird also stets ein Operand mit dem Ergebnis überschrieben. Die Befehle sind werden als "Typ SS" bezeichnet. Diese Befehle können in weniger Speicherworten kodiert werden.

  Beispiel:

  Unter den gleichen Verhältnissen wir oben benötigen wir nur noch 2 "Erweiterungsworte", also 96 Bit zur Kodierung eines Befehls.

  $\leftarrow$ 32 Bit $\rightarrow$

Op-Code	weitere Informationen
a	
b	

[5] Quelle: [Jes75].

Wir wollen diese Befehle benutzen, um die o.a. Anweisung zu übersetzen:

```
MOVE t1, a     - - Speicher[t1] := Speicher[a]
ADD  t1, b     - - Speicher[t1] := Speicher[t1] + Speicher[b]
MULT t1, c
MOVE d, t1
```

Wir benötigen pro Arithmetik-Befehl 3 und pro MOVE-Befehl 2 Zugriffe auf den Speicher, insgesamt also 10 Speicherzugriffe.

Unter Umständen könnte erkannt werden, dass man auch `d` als Adresse der Hilfszelle benutzen könnte (`t1 = d`). Dies erfordert aber eine Optimierung durch den Compiler, die sich im allgemeinen Fall als schwierig erweisen kann.

- 1 1/2-Adreßmaschinen, 1 1/2-Adreßbefehle

  Es hat sich gezeigt, dass man sowohl die Befehlswortbreite als auch die Zugriffsgeschwindigkeit auf Daten verbessern kann, wenn man Registerspeicher einführt. Die Adresse des Datums im Registerspeicher hat man als halbe Adresse bezeichnet. Man kommt damit zu 1 1/2 Adreßbefehlen. Diese bezeichnet man als "Typ RS"-Befehle. Die Registernummer kann in der Regel noch im ersten Befehlswort kodiert werden.

  Beispiel:

  Unter den gleichen Annahmen wie oben erhalten wir das in Abb. 2.13 dargestellte Format mit 64 Bit zur Kodierung eines Befehls.

  ← 32 Bit →

Op-Code	Reg.nr.	weitere Informationen
a		

  Abbildung 2.13: 1 1/2-Adreßbefehl

  Wir wollen diese Befehle benutzen, um die o.a. Anweisung zu übersetzen:

```
LOAD D1, a     - - D[1] := Speicher[a]
ADD D1, b     - - D[1] := D[1] + Speicher[b]
MULT D1, c     - - D[1] := D[1] * Speicher[c]
STORE d, D1     - - Speicher[d] := D[1]
```

  Diese Sequenz benötigt 4 Speicherzugriffe und weniger volle Adreßteile als die zuletzt angegebene.

- 1-Adreßmaschinen, 1-Adreßbefehle

  Früher enthielten Maschinen gelegentlich nur ein einzelnes, von Befehlen aus ansprechbares Register, den sog. Akkumulator. Damit kann die Registeradresse entfallen, und man kommt zu 1-Adreßbefehlen. Man verwendet hier Überdeckung und Implizierung.

  Wir wollen diese Befehle benutzen, um die o.a. Anweisung zu übersetzen:

```
LOAD  a    - - acc := Speicher[a]
ADD   b    - - acc := acc + Speicher[b]
MULT  c    - - acc := acc * Speicher[c]
STORE d    - - Speicher[d] := acc
```

  Im speziellen Fall wird die gleiche Sequenz wie im Fall davor erzeugt. Im allgemeinen Fall wird man nicht alle Zwischenergebnisse in einem einzigen Akkumulator halten können. Bei komplexeren Ausdrücken werden daher Umspeicherungen von Zwischenergebnissen im Akkumulator in den Speicher erforderlich sein (sog. *spilling*) .

- 0-Adreßmaschinen, 0-Adreßbefehle

  Kellermaschine: alle arithmetischen Operationen werden auf einem Keller (Stack) ohne explizite Angabe der Operanden durchgeführt. Es werden jeweils die beiden obersten Stackelemente verknüpft.

  Zusätzlich zum Keller ist natürlich noch ein normaler Speicher vorhanden. Aus diesem wird der Arithmetik-Keller mit PUSH-Befehlen in der richtigen Reihenfolge geladen und

in diesen werden die Ergebnisse mittels POP-Befehlen übertragen. Die PUSH- und POP-Befehle enthalten für diesen Zweck Adressenteile. Nur die Arithmetik-Befehle selbst enthalten keine Adressen!

Die Codeerzeugung für arithmetische Ausdrücke ist besonders einfach: es genügt die Übersetzung in Postfix–Form[6].

Wir wollen diese 0-Adreß-Befehle benutzen, um die o.a. Anweisung zu übersetzen:

```
PUSH a        - - a: Adresse im Hauptspeicher
PUSH b
ADD           - - 0-Adreßbefehl
PUSH c
MULT          - - 0-Adreßbefehl
POP d         - - speichere oberstes Kellerelement in Speicher[d]
```

Kellermaschinen bieten den Vorteil einer einfacher Übersetzung von arithmetischen Ausdrücken. Optimierungen brauchen nicht entwickelt zu werden, da keine Optimierungen möglich sind. Andererseits können auch häufig benötigte Informationen nicht in Registern gehalten werden und Kellermaschinen sind daher relativ langsam.

Vorkommen: u.a. bei Maschinen der Fa. Burroughs, bei Taschenrechnern der Fa. hp, bei verschiedenen abstrakten Maschinen (einschließlich der abstrakten Java-Maschine, s.u.) und beim Gleitkommaprozessor der Intel 80x86-Serie.

Die Befehlssätze moderner Rechner enthalten häufig Maschinenbefehle mit einer unterschiedlichen Anzahl von Operanden-Adressen. Besonders auffällig ist dies bei VAX-Rechnern, wo grundsätzlich 2- **und** 3-Adreßformen der Befehle zur Verfügung stehen.

2.4 Klassifikation von Befehlssätzen

2.4.1 CISC-Befehlssätze

Eine weitreichende Realisierung der verschiedenen Adressierungsarten bieten die Rechner, die bis in die Mitte der 80er Jahre hinein entworfen wurden. Diese Rechner zeichnen sich durch eine große Zahl von Befehlen sowie durch viele Adressierungsarten aus. Sie werden daher heute als CISC-Prozessoren (engl. *complex instruction set computers*) bezeichnet. Diese komplexen Befehlssätze waren v.a. durch zwei Gründe motiviert:

1. Der Entwurf erfolgte vor dem Hintergrund einer großen Diskrepanz zwischen der internen Verarbeitungsgeschwindigkeit (die sich durch die Halbleitertechnologie ergab) und der Geschwindigkeit der Speicher (die durch Magnetkernspeicher niedrig war). Es galt damit, mit jedem Holen eines Befehls möglichst komplexe Operationen (wie z.B. Additionen mit indirekt addressierten Operanden) anstoßen zu können. Vor diesem Hintergrund wurde auch untersucht, wie Befehlssätze beschaffen sein müssen, damit ein gegebenes Problem mit möglichst wenigen Zugriffen auf den Speicher ausgeführt werden kann.

   Inzwischen hat sich allerdings die Diskrepanz zwischen der internen Verarbeitungsgeschwindigkeit der Geschwindigkeit der Speichers wieder erhöht. Durch die Einführung von Caches, der Integration des Speicherzugriffs in die Fließbandorganisation usw. versucht man, diesen Effekt bei RISC-Architekturen zu mildern.

2. Die Diskrepanz zwischen Abstraktionsniveau von Assemblersprachen und höheren Programmiersprachen wurde allgemein beklagt und als **semantische Lücke** bezeichnet. Ein Ziel war, diese Lücke durch immer mächtigere Maschinenbefehle zu verkleinern. In einzelnen Fällen wurden sogar sog. *direct execution architectures* vorgeschlagen, die in höheren Programmiersprachen erstellte Programme weitgehend direkt ausführen sollten.

Beispiel: Die CISC-Prozessorfamilie Motorola MC 680x0

Maschinenbefehle der Motorola 68000er-Prozessorfamilie besitzen das folgende Format:

[6] siehe Vorlesung "PSÜ".

Opcode "00"	Größe (s.u.)	Z i e l Register |Modus	Q u e l l e Register |Modus
bis zu 4 Erweiterungsworte			

Größe: "01"=Byte, "11"=Wort, "10"=Doppelwort

Tabelle 2.8 zeigt die Adressierungsarten des einfachsten Mitgliedes dieser Familie, des MC 68000, am Beispiel des MOVE-Befehls.

Modus	Register-Feld	Erweit.-worte	Assembler-Notation	Adressierungstyp	effektiver Operand ggf. Seiteneffekt
"000"	n	0	Dn	Register-Adressierung	D[n]
"001"	n	0	An	(Adreß-)Register-Adress.	A[n]
"010"	n	0	(An)	(Adreß-)Register indir.	Speicher[A[n]]
"011"	n	0	(An)+	(Adreß-)Register indir. mit Postincrement;i=1,2,4	Speicher[A[n]]; A[n]:=A[n]+i
"100"	n	0	-(An)	(Adreß-)Register indir. mit Prädecrement	A[n]:=A[n]-i; Speicher[A[n]]
"101"	n	1	d(An)	Relative A. mit 16-Bit Distanz	Speicher[d+A[n]]
"110"	n	1	d(An,Xm)	Register-Relative Adr. mit Index; Xm kann Daten- oder Adreßregister sein	Speicher[d+A[n]+X[m]]
"111"	"000"	1	d	direkte Adressierung	Speicher[d]
"111"	"001"	2	d	direkte Adr. (32 Bit)	Speicher[d]
"111"	"010"	1	d(PC),d(*)	Programmzähler-relativ	Speicher[PC+d+2]
"111"	"011"	1	d(*,Xn)	Programmzähler-relativ mit Index	Speicher[PC+d+2+X[n]]
"111"	"100"	1-2	#zahl	unmittelbare Adressierung	zahl

Tabelle 2.8: MOVE-Befehle des 68000

Neuere Mitglieder der 680x0er-Prozessorfamilie besitzen noch komplexere Adressierungsarten.

2.4.2 RISC-Befehlssätze

Die Voraussetzung stark unterschiedlicher Geschwindigkeiten des Hauptspeichers und des Rechenwerks waren mit der Einführung der Halbleitertechnologie für beide Bereiche nicht mehr gegeben. Damit waren auch einfachere Maschinenbefehlssätze akzeptabel, ja sie boten sogar einen Vorteil: Während komplexe Befehle nur mit Mühen ohne Interpretation durch ein Mikroprogramm (s.u.) zu realisieren sind, können einfache Befehle direkt durch die Hardware verarbeitet werden. Da ein Mikroprogramm stets auch Zusatzaufwand für das Holen der Mikroprogramme bedeutet, entsteht durch den Fortfall der Mikrobefehle ein Zeitvorteil. Der Fortfall einer Interpretationsebene zieht allerdings Konsequenzen nach sich. Da alle Befehle jetzt direkt durch die Hardware interpretiert werden, darf kein Befehl mehr sehr komplex sein[7]. Komplexe Befehle sollten also aus dem Befehlssatz gestrichen werden. Man kommt damit zu den Rechnern mit reduziertem Befehlssatz, den RISC-Maschinen (engl. *reduced instruction set computer*).

Ein wichtiges Kriterium zur Unterscheidung zwischen RISC und CISC-Maschinen ist der sog. CPI-Wert, der wie folgt definiert wird:

Def.: Unter dem **CPI-Wert** (engl. *cycles per instruction*) einer Menge von Maschinenbefehlen versteht man die mittlere Anzahl interner Bus-Zyklen pro Maschinenbefehl.

Für mikroprogrammierbare Rechner liegt der CPI-Wert häufig über 2. Für ältere Prozessoren mit komplexeren Befehlssätze liegt der CPI-Wert noch höher. Für RISC-Rechner strebt man an, möglichst alle Befehle innerhalb eines internen Bus-Zyklus abschließen zu können. Zwar wird dies für einige Befehle, wie

[7]Moderne CISC-Prozessoren verzichten zumindest für häufig ausgeführte Befehle ebenfalls auf das Mikroprogramm. Eine Hardware-mäßige Ausführung komplexer Befehlssätze bleibt aber deutlich schwieriger als die Hardware-mäßige Ausführung einfacher Befehlssätze.

z.B. Gleitkomma-Befehle, nicht effizient möglich sein. Dennoch versucht man, zumindest einen CPI-Wert von knapp oberhalb von 1 zu erreichen. Damit möchte man dem eigentlichen Ziel, nämlich einer möglichst kurzen Programmlaufzeit, nahe kommen. Die Programmlaufzeit errechnet sich wie folgt:

$Laufzeit =$ Dauer eines Buszyklus $*$ Anzahl der auszuführenden Befehle $*$ CPI-Wert des Programms

Um einen kleinen CPI-Wert zu erreichen, gibt man folgende Ziele für den Entwurf des Befehlssatzes vor:

- Wenige, einfache Adressierungsarten
- LOAD/STORE–Architektur

  Arithmetische Operationen können nur auf Operanden in Registern ausgeführt werden. Zur Ausführung solcher Operationen sind die Operanden zunächst mit LOAD-Befehlen zu laden.
- Feste Befehlswortlängen

Sowohl die einfachen Adressierungsarten als auch die LOAD/STORE-Architektur führen dazu, dass Programme evtl. nicht ganz so kompakt dargestellt werden können, wie bei CISC-Prozessoren. Dies wurde als nicht mehr so gravierend betrachtet, da

1. immer größere Halbleiterspeicher zur Verfügung stehen und da
2. mit Hilfe von Pufferspeichern (s.u.) die etwas größere Speichergeschwindigkeit zur Verfügung gestellt werden konnte.

Man erkennt, dass der Übergang auf andere Maschinenbefehlssätze (d.h. auf eine andere externe Architektur) durch Änderungen in den Techniken der Realisierung bedingt ist. Das bedeutet aber auch, dass unter anderen technologischen Randbedingungen (z.B. wieder wachsende Geschwindigkeitsunterschiede zwischen Speicher und Rechenwerk) der umgekehrte Weg möglich ist.

Neben Änderungen in der Speichertechnologie war eine zweite wesentliche Änderung Vorraussetzung für die Einführung von RISC-Rechnern: die praktisch vollständige Abkehr von der Assemblerprogrammierung aufgrund besserer Compiler für höhere Programmiersprachen. Dies erlaubte die folgenden Änderungen gegenüber CISC-Rechnern:

- Das Ziel, die **semantische Lücke** zwischen Assemblersprachen und höheren Programmiersprachen durch möglichst leistungsfähige Assemblerbefehle zu reduzieren, wurde belanglos. Die große Zahl von Maschinenbefehlen war durch Compiler ohnehin immer nur zu einem kleinen Teil genutzt worden.
- Die Compiler wurden leistungsfähig genug, um Aufgaben zu übernehmen, die bisher mit Hardware-Aufwand realisert worden waren und dadurch häufig die Taktperiode verlängert hatten (Beispiel: Berücksichtigung des Hardware-Pipelinings im Compiler).

Beispiel eines RISC-Befehlssatzes:

Die Rechner der MIPS-Serie besitzen die in Tabelle 2.9 dargestellten Maschinenformate [HP95].

	Größe [bit]	6	5	5	5	5	6
Arithmetische Befehle	Format R	op	rs	rt	rd	shamt	funct
Immediate + LOAD/STORE	Format I	op	rs	rt	adr-teil		
Bedingte Sprungbefehle	Format J	op	Sprungadresse				

Tabelle 2.9: Befehlsformate der Rechner MIPS R2000-R8000

Alle arithmetischen Befehle der Prozessoren dieser RISC-Familie erwarten die Operanden in Registern. Quell- und Zielregister können getrennt voneinander angegeben werden. Es handelt sich also um eine auf Registeradressen beschränkte Variante von 3-Adreßbefehlen (Befehle vom "Typ RRR"). Die Wirkung der Befehle kann in Register-Transfer-Notation beschrieben werden durch:

`Reg[rd] := Reg[rs] • Reg[rt]`,

wobei `Reg` den Registersatz und • eine durch Operationscode `op`, Funktionserweiterung `funct` und *shift amount* `shamt` bestimmte Funktion bezeichnet.

LOAD/STORE-Befehle erlauben die Angabe eines Ziel bzw. Quell-Registers, eines Basisregisters und von einer 16-Bit Distanz. Der STORE-Befehl besitzt die Semantik

`Speicher[Reg[rs] + sign_ext(adr-teil,32) ] := Reg[rt]`.

Immediate-Befehle erlauben die Angabe von 16-Bit Konstanten. Der Immediate-Additions-Befehl `ADDI rd,rs,adr-teil` (add immediate) würde durch folgendes Statement beschrieben:

`Reg[rd] := Reg[rs] + sign_ext(adr-teil,32)`.

Das Ergebnis einer Operation wird in Assemblersyntax hierbei als erster Parameter angegeben.

Insgesamt können alle Befehle in 32 Bit kodiert werden.

Die Tabellen 2.10 bis 2.13 enthalten die wichtigsten Befehle der MIPS-Architektur [HP95].

Beispiel	Bedeutung	Kommentar
lw $1,100($2)	Reg[1] := Speicher[Reg[2]+100]	load word
sw $1,100($2)	Speicher[Reg[2]+100] := Reg[1]	store word
lui $1,100	Reg[1] := 100 << 16	Lade oberes Halbwort
mfhi $1	Reg[1] := Hi	Lade Multiplizierregister
mflo $1	Reg[1] := Lo	Lade Multiplizierregister

Tabelle 2.10: Datentransferbefehle

Beispiel	Bedeutung	Kommentar
add $1,$2,$3	Reg[1] := Reg[2] + Reg[3]	Addition, Ausnahmen (Überlauf) möglich
sub $1,$2,$3	Reg[1] := Reg[2] - Reg[3]	Subtraktion, Ausnahmen (Überlauf) möglich
addi $1,$2,100	Reg[1] := Reg[2] + 100	Ausnahmen (Überlauf) möglich
addu $1,$2,$3	Reg[1] := Reg[2] + Reg[3]	Addition natürl. Zahlen, keine Ausnahmen
subu $1,$2,$3	Reg[1] := Reg[2] - Reg[3]	Subtraktion natürl. Zahlen, keine Ausnahmen
addiu $1,$2,100	Reg[1] := Reg[2] + 100	Addition von Konstanten, keine Ausnahmen
mult $2,$3	Hi & Lo:= Reg[2] * Reg[3]	Multiplikation m. 64-Bit Ergebnis ganze Z.
multu $2,$3	Hi & Lo:= Reg[2] * Reg[3]	Multiplikation m. 64-Bit Ergebnis, natürl. Z.
mfc0 $1,$epc	Reg[1] := EPC	Kopieren des *exception program registers*
div $2,$3	Lo:= Reg[2] / Reg[3], Hi := Reg[2] mod Reg[3]; ganze Zahlen	
divu $2,$3	Lo:= Reg[2] / Reg[3], Hi := Reg[2] mod Reg[3]; natürl. Zahlen	

Tabelle 2.11: Arithmetische Befehle

Die Additions- und Subtraktionsfehle für ganze und für natürliche Zahlen der Tabelle 2.11 liefern dieselben Bitvektoren als Ergebnis. Die Versionen für natürliche Zahlen sind v.a. für Adressarithmetik gedacht. Für diese möchte man üblicherweise Überläufe ignorieren. Sie lösen daher keine Ausnahmen aus. C-Compiler generieren für den MIPS-Befehlssatz ausschließlich Additionsbefehle für natürliche Zahlen, da für C das Ignorieren von Überläufen vorgeschrieben ist. Vor diesem Hintergrund kann man verstehen, dass `addiu` die Konstante **vorzeichenerweitert** zum Registerinhalt addiert, was man von einer *unsigned*-Operation nicht erwarten würde.

Der auf einen Sprung statisch folgende Befehl wird stets noch ausgeführt (*delayed jump*), dies wird aber durch den Assembler ggf. durch Einführen eines NOPs versteckt.

Neben den angegebenen Befehlen kennt der Assembler noch eine ganze Reihe von Pseudo-Befehlen, für welche er Sequenzen von Maschinenbefehlen erzeugt.

Beispiel	Bedeutung	Kommentar
and \$1,\$2,\$3	Reg[1] := Reg[2] $\wedge$ Reg[3]	und
or \$1,\$2,\$3	Reg[1] := Reg[2] $\vee$ Reg[3]	oder
andi \$1,\$2,100	Reg[1] := Reg[2] $\wedge$ 100	and mit Konstanten
sll \$1,\$2,10	Reg[1] := Reg[2] << 10	schiebe nach links logisch
rll \$1,\$2,10	Reg[1] := Reg[2] >> 10	schiebe nach links logisch

Tabelle 2.12: Logische Befehle

Beispiel	Bedeutung	Kommentar
beq \$1,\$2,100	if (Reg[1]=Reg[2]) then PC := PC +4+100	*branch if equal*
bne \$1,\$2,100	if (Reg[1] $\neq$ Reg[2]) then PC := PC +4+100	*branch if not equal*
slt \$1,\$2,\$3	Reg[1] := if (Reg[2] < Reg[3]) then 1 else 0	*set on less than*, ganze Zahlen
slti \$1,\$2,100	Reg[1] := if (Reg[2] < 100) then 1 else 0	*set on less than*, ganze Zahlen
sltu \$1,\$2,\$3	Reg[1] := if (Reg[2] < Reg[3]) then 1 else 0	*set on less than*, natürl. Zahlen
sltui \$1,\$2,\$3	Reg[1] := if (Reg[2] < 100) then 1 else 0	*set on less than*, natürl. Zahlen
j 1000	PC := 1000	*jump*
jr \$31	PC := Reg[31]	*jump register*
jal 1000	Reg[31] := PC + 4; PC := 1000	*jump and link*

Tabelle 2.13: Sprünge und Tests

2.4.3 DSP-Befehlssätze

Prozessoren werden heute nicht nur in PCs und Workstations, sondern vielfach auch in sog. **eingebetteten Systemen** eingesetzt. Dies sind Systeme, in denen die Informationsverarbeitung in die (meist physikalische) Umgebung vollständig integriert ist und in denen (physikalische) Größen direkt von den Ergebnissen der Informationsverarbeitung aus beeinflußt werden. Diese Systeme kommen meist ohne Bildschirm und Tastatur aus. Das Benutzer-Interface läßt in der Regel nicht erkennen, dass Eingaben an Rechensysteme gemacht werden. Beispiele hierfür sind informationsverarbeitende Systeme in der Fahrzeug-Elektronik, in der Telekommunikation und im Audio/Video-Bereich. Charakteristisch für fast alle dieser Bereiche ist weiterhin die hohe Bedeutung der **Effizienz** der technischen Lösung, bei portablen Anwendungen v.a. hinsichtlich des Stromverbrauchs.

Eine wichtige Teilaufgabe von eingebetteten Systemen besteht in der digitalen Signalverarbeitung (DSP). Für diese Aufgaben sind spezielle, DSP-Aufgaben effizient verarbeitende Prozessoren entwickelt worden. DSP-Prozessoren besitzen die folgenden spezifischen Eigenschaften:

- *saturating arithmetic*
- **spezielle Adressierungsarten:**
  Aufgrund der Anwendung in üblichen DSP-Algorithmen sehen DSP-Prozessoren häufig spezielle Adressierungsarten vor. Die Modulo-Adressierung beispielsweise erlaubt die effiziente Realisierung von verzögerten Signalen mit Hilfe von Ringpuffern.
- **Eingeschränkte Parallelität:**
  Die Rechenwerke der meisten DSP-Prozessoren erlauben es, in einem Takt Zuweisungen zu mehreren Registern gleichzeitig auszuführen. Diese Prozessoren stellen diese Form der begrenzten Parallelität dann meist auch an der Befehlsschnittstelle zur Verfügung. Gängig sind z.B. *parallel moves* genannte Befehle, die gleichzeitig eine Arithmetik-Operation und einen Datentransport veranlassen.
- **Heterogene Registersätze**
- **multiply/accumulate-Befehle**

  Eine wichtige Aufgabe der digitalen Signalverarbeitung ist die digitale Filterung. Bei der Filterung werden die Elemente $y(j)$ einer Ausgangsfolge y mit der Formel

$$(2.9) \qquad y(j) = \sum_{k=0}^{m} a(k) * x(j-k)$$

aus den Elementen einer Eingangsfolge x berechnet. Der Vektor a stellt dabei einen Gewichtsvektor dar, der die Charakteristik des Filters bestimmt. m bestimmt die Ordnung des Filters. Die Summe kann iterativ aus Partialsummen $y_K(j)$ der Summation bis zu einem gewissen K berechnet werden:

$$(2.10) \qquad y_{-1}(j) = 0$$
$$(2.11) \qquad y_k(j) = y_{k-1}(j) + a(k) * x(j-k)$$
$$(2.12) \qquad y(j) = y_m(j)$$

Multiply/accumulate-Befehle erlauben es, mit einem einzigen Befehl einen Iterationsschritt der Gleichung 2.11 auszuführen. Dazu wird ein Akkumulatorregister `ACC` mit 0 initialisiert, zwei Datenregister `X` und `Y` mit $a(0)$ bzw. $x(j)$ geladen und und zwei Adressregister `A1` und `A2` zeigen auf das nächste relevante Element der Arrays a bzw. x. Anschließend wird mittels eines einzigen multiply/accumulate-Befehles in einer Schleife folgende Zuweisungen ausgeführt:

```
ACC := ACC + X * Y, X := Speicher[A1], Y := Speicher[A2], A1++, A2--
```

Mit jeder Befehlsausführung wird die nächste Partialsumme berechnet.

- **Realzeit-Fähigkeit:**
  Es ist wichtig, die maximale Laufzeit eines Programms möglichst exakt und nicht nur im Mittel angeben zu können. Deshalb wird z.B. vielfach auf Caches, die eine datenabhängige Laufzeit bewirken würden, verzichtet.

DSP-Prozessoren bieten v.a. für DSP-Applikationen mehr Rechenleistung pro Watt Leistungsverbrauch, weshalb sie v.a. in portablen Geräten vielfältig Einsatz finden.

2.4.4 EPIC- und VLIW-Befehlssätze

Die begrenzte Form der Parallelität von DSP-Prozessoren wird bei *very large instruction word* (VLIW) Architekturen weiter ausgebaut. Diese Architekturen besitzen eine größere Anzahl an funktionellen Einheiten sowie meist auch Speicher mit mehreren Ports. Das Befehlswort ist bei diesen Architekturen breit genug, um alle Einheiten gleichzeitig anzusteuern. VLIW-Architekturen beziehen ihre Leistungsfähigkeit aus der expliziten Parallelität in den langen VLIW-Befehlen, können sich daher den Aufwand zur Erkennung möglicher Parallelität in Hardware sparen.

Beispiel:

Abb. 2.14 zeigt skizzenhaft ein Beispiel einer einfachen, hypothetischen Architektur (Multiplexer und Bus-Treiber sind hierbei nicht gezeigt). In den ALUs können hier 3 Operationen gleichzeitig ausgeführt werden.

Prozessoren mit expliziter Parallelität werden neuerdings auch als *explicit parallelism instruction computers* (EPIC) bezeichnet. Sie beinhalten als spezielle Klasse die VLIW-Architekturen, können aber die explizite Parallelität evtl. auch anders als gemäß Abb. 2.14 ausnutzen. Kernidee der EPIC-Architekturen ist, die Erkennung der Parallelität in den Compiler zu verlagern und so mehr Parallelität bei gleichzeitiger Einsparung an Hardware zu gewinnen.

Weitere Beispiele:

1. Der TMS320C62xx Signalprozessor ist die erste "richtige" kommerzielle VLIW-Maschine, angekündigt 1997.

   Eigenschaften:

   - Zwei Rechenwerke mit je

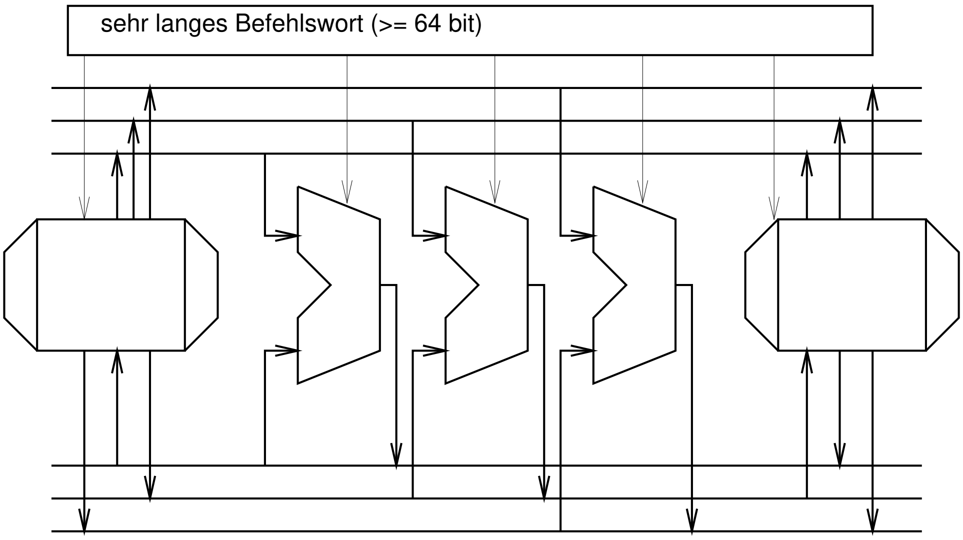


Abbildung 2.14: VLIW-Architektur (Beispiel)

  - 16 x 16 Bit Multiplizierer
  - 1 Addierer
  - 1 Addierer/Shifter
  - 1 Addierer/Shifter/Normalisierer
  - Registerspeicher mit 16 32-Bit-Registern
  - 1 Speicherlese- + 1 Speicherschreib-Pfad
  - 1 Pfad zum anderen Rechenwerk

- Vorgesehen, bei 200 MHz bis zu 8 32-Bit-Befehle pro Takt (5 ns) zu starten (max. 1.6 "GIPS")
- Jeder 32-Bit-Befehl spricht eine funktionelle Einheit an; der Compiler muss die Zuordnung zur Compile-Zeit vornehmen.

  Jeder 32-Bit-Befehl besitzt 1 Bit, welches entscheidet, ob der nächste Befehl noch im gleichen Takt ausgeführt wird. Dadurch Holen von max. 256 Befehlsbits (*instruction packet* genannt) / Zyklus.

1	Befehl-A	1	Befehl-B	1	Befehl-C	1	Befehl-D	0	Befehl-E	0	Befehl-F

Ausführung:	
Schritt 1	Befehle A, B, C, D, E
Schritt 2	Befehl F

Abbildung 2.15: Befehlspakete des TMS320C62xx

  De facto variable Befehlslänge von 32 bis 256 Bit. Falls in einem Zyklus nicht alle funktionellen Einheiten benutzt werden können, gehen dadurch keine Befehlsbits verloren.

  Man kann "mitten in ein Befehlspaket" hinein springen.

- Übliche DSP-Funktionalität: Sättigungsarithmetik (*saturating arithmetic*), Modulo-Adressierung (*modulo addressing*), Normalisierungsoperation, integrierte E/A-Leitungen, kein virtueller Speicher.

2. Der Philips TriMedia-Prozessor für Multimedia-Applikationen mit bis zu 5 Befehlen/Zyklus.

3. Der Intel i860 ist ein Prozessor mit einer Befehlswortbreite von 64 Bit. Davon spricht die eine Hälfte Gleitkomma-Einheiten und die andere Integer-Einheiten an. Dieser Prozessor enthält damit auch schon VLIW-Konzepte.

4. Die in der PG PRIPS entwickelte PROLOG-Maschine [ABM+93].
5. Die als Nachfolge der Pentium-Architektur u.a. von der Fa. Intel entwickelte IA-64 Architektur. Sein Befehlsformat gehört zur EPIC-Klasse (siehe Abb. 2.16).

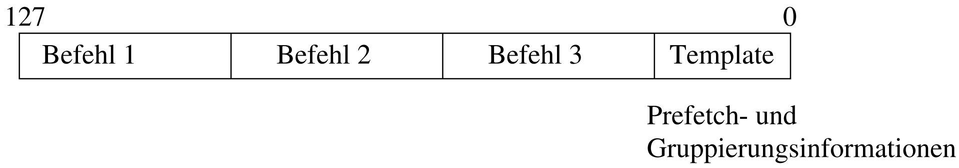


Abbildung 2.16: Befehlsformat des IA-64

Potentielle Probleme der VLIW-Architekturen liegen in folgenden Bereichen:

- Skalierbarkeitsprobleme

  Die explizite Parallelität in VLIW-Befehlen hat zur Folge, dass Architekturen mit mehr funktionellen Einheiten auch ein anderes Befehlsformat benötigen. Das heißt, selbst innerhalb einer Rechnerfamilie gibt es keine Binärcode-Kompatibilität.

  Einen Ausweg bietet hier die Benutzung anderer Formate zur Abspeicherung ausführbarer Programme: ausführbare Programme werden in Form von auszuführenden Operationen (abstrakte Opcodes) gespeichert, und die Erzeugung der Binärcodes erfolgt erst während des Ladevorgangs.

- Timing-Probleme

  VLIW-Architekturen sind aufgrund ihrer Parallelität auf Speicher angewiesen, die eine größere Anzahl von gleichzeitigen Zugriffen erlauben. Bei allen bekannten Realisierungsformen solcher Speicher hängt die Zugriffszeit davon ab, ob die Adressen dieselben Bereiche innerhalb der Speicher ansprechen. Da die Adressen nun vor der Ausführungszeit nicht bekannt sind, ergeben sich Probleme bei der Wahl der Zykluszeit. Bei einer festen Zykluszeit müßte man den schlechtesten Fall (alle Adressen beziehen sich auf denselben Speicherblock) annehmen und würde damit kaum über die Geschwindigkeit klassicher Befehle hinauskommen. Es müssen also Verfahren untersucht werden, mit denen die Zykluszeit von den tatsächlich vorkommenden Speicherzugriffskonflikten abhängig gemacht werden kann. Diese Techniken werden aber eigentlich auch bei modernen RISC-Maschinen benötigt.

- Kompatibiltätsprobleme

  Es ist nicht einfach, mit VLIW-Architekturen den Befehlssatz des 8086 zu verarbeiten. In PCs konnten sie daher bislang nicht eingesetzt werden. Für eingebettete Systeme, die nicht zum 8086 codekompatibel sein müssen, kommen sie aber durchaus in Frage.

  Der IA-64 soll allerdings x86-Code ausführen können.

Gelegentlich wird gegen diese Architekturen eingewandt, die Wortbreite wäre unsinnig groß. Dies trifft aber v.a. auf die ältere Implementierung von Fisher zu. Die oben angegebenen Maschinen sind durchaus effizient.

Bei gegenwärtigen RISC-Architekturen ist man bei derartig vielen Fließband-Stufen angelangt, dass die Hardware-Logik zur Behandlung von Datenabhängigkeiten bereits einen großen Teil der Logik überhaupt ausmacht. Gleichzeitig ist dieser Teil der Hardware sehr schwer korrekt zu entwerfen. Auch nach Beseitigung des bekannten Pentium-Gleitkomma-Fehlers enthalten gegenwärtige Pentium-Chips eine Vielzahl von Fehlern, die zum gutem Teil durch die Komplexität der Fließbandverarbeitung erklärbar sind. Selbst Hennessy und Patterson mussten zugeben, dass die erste Auflage ihres Buches [HP96] auch nach Erprobung an vielen amerikanischen Universitäten noch einen Fehler in der Fließbandkonzeption enthielt. Es sei in diesem Zusammenhang Bob Rau (hp Labs) zitiert: *the only ones in favor of superscalar (RISC) machines are those who haven't built one yet.*

Mit der Ankündigung des VLIW-Prozessors durch die Fa. Texas Instruments (1997) und des IA-64 scheint sich ein Trend zu solchen Prozessoren abzuzeichnen.

2.4.5 Multimedia-Befehle

Aufgrund der hohen Leistungsanforderungen im Multimediabereich, v.a. durch den Einsatz von DSP-Algorithmen zur Audio- und Videosignalverarbeitung, entstand der Bedarf zur Leistungssteigerung bei diesen Applikationen. Aus diesem Grund werden z.Z. (1997) viele RISC- und CISC-Prozessoren um DSP-spezifische Eigenschaften erweitert[PWW97]. So werden beispielsweise meist auch multiply/accumulate-Befehle angeboten. Dazu kommt die Ausnutzung der ohnehin vorhandenen parallelen Rechenwerke für die gleichzeitige Verarbeitung verschiedener Farbwerte oder verschiedener Bildpunkte.

Beispiele:

1. Die Hewlett-Packard *precision architecture* (hp PA) unterstützt den sog. Halbwort-Additionsbefehl `HADD`, der die Inhalte von 32-Bit-Registern als zwei in 16 Bit kodierte Zahlen interpretiert. Mit einem `HADD`-Befehl werden so zwei in 16 Bit kodierte Zahlen addiert (siehe Abb. 2.17).

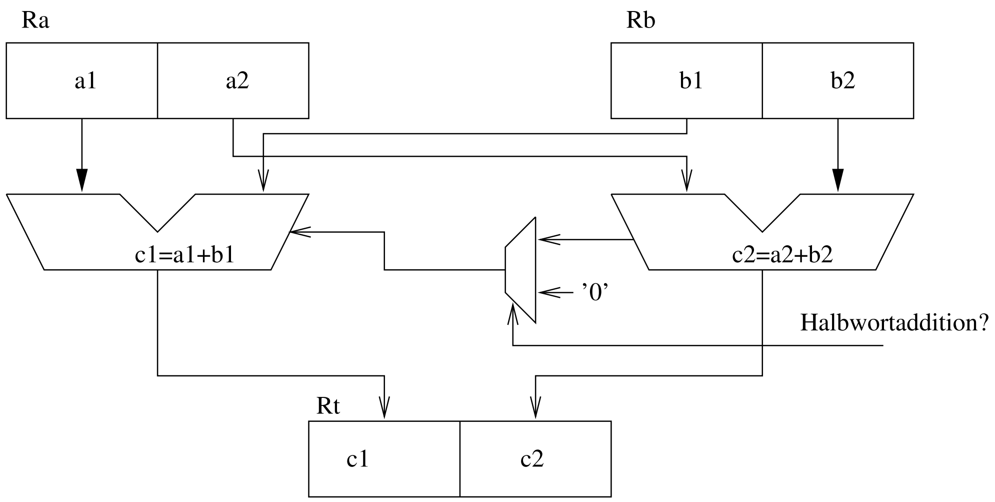


Abbildung 2.17: 16-Bit-Addition von Werten in 32-Bit-Registern Ra und Rb

Im 32-Bit-Addierer muss man dafür nur die internen Überträge an den 16-Bit-Grenzen unterbrechen.

Weiterhin können beide Additionen optional mit Sättigungsarithmetik erfolgen. Dies entspricht bei jeder Addition jeweils einem zusätzlichen Test auf Überlauf oder Unterlauf. Beachtet man, dass man in manchen RISC-Befehlssätzen die 16-Bit-Werte durch Schiebeoperationen aus 32-Bit-Worten extrahieren und ggf. zu solchen Worten wieder kombinieren muss, so ist verständlich, dass bis zu 10 Befehle durch einen `HADD`-Befehl ersetzt werden können.

2. Die Pentium MMX-Architektur benutzt für die ohnehin vorhandenen 64-Bit Datenwege einen ähnlichen Ansatz. 64-Bit-Vektoren können als 8 bytecodierte, 4 wortcodierte oder 2 doppelwortcodierte Zahlen betrachtet werden. Für die Arithmetik sind *wrap around- saturating*-Optionen wählbar (siehe Tabelle 2.14) [PWW97]. Es stehen separate Multimediaregister `mm0` bis `mm7` zur Verfügung, die aber stets konsistent mit den Gleitkommaregistern gehalten werden, damit sie beim Umschalten auf andere Prozesse nicht separat gerettet werden müssen, also den Kontext des laufenden Prozesses nicht vergrößern (siehe Ende diese Kapitels).

   Abb. 2.18 zeigt, wie die MMX-Befehle benutzt werden können, um zwischen zwei Bildern skaliert zu interpolieren. Die größte Beschleunigung ergibt sich, wenn die Bildinformation einer Farbe in benachbarten Bytes des Speichers abgelegt ist. Dann können, wie in dem Beispiel, die Werte einer Farbe für vier benachbarte Bildpunkte in einem Schritt bearbeitet werden und es ergibt sich im Idealfall eine Beschleunigung um den Faktor 4.

3. Der Ultra-SPARC Prozessor besitzt zur Multimedia-Untersützung den sog. *visual instruction set* (VIS). Dieser enthält einen Befehl zur Bewegungsschätzung, der bei 8-Bit Daten 8 Subtraktionen, 8 Absolutwerte und 8 Additionen in einem Zyklus ausführt. Auf diese Weise können bei einer Bewegungsschätzung 1.500 konventionelle Befehle durch 32 solcher Befehle ersetzt werden [BK95].

Befehl	Optionen	Beschreibung
Padd[b/w/d] PSub[b/w/d]	*wrap around,* *saturating*	Addition/Subtraktion von Bytes, Worten, Doppelworten
Pcmpeq[b/w/d] Pcmpgt[b/w/d]		Ergebnis = "111..11" falls wahr, "000..00" sonst Ergebnis = "111..11" falls wahr, "000..00" sonst
Pmullw Pmulhw		Multiplikation von 4*16-Bit, unteres Wort Multiplikation von 4*16-Bit, oberes Wort
Psra[w/d] Psll[w/d/q] Psrl[w/d/q]	Stellenzahl in Register oder in Befehl	Paralleles Schieben von Worten, Doppelworten und 64 Bit-Quad-Worten
Punpckl[bw/wd/dq] Punpckh[bw/wd/dq]		Paralleles Entpacken Paralleles Entpacken
Packss[wb/dw]	*saturating*	Paralleles Packen
Pand, Pandn Por, Pxor		Logische Operationen auf 64-Bit-Werten
Mov[d/q]		Datentransport Speicher/Register

Tabelle 2.14: Befehle der x86-MMX-Erweiterung

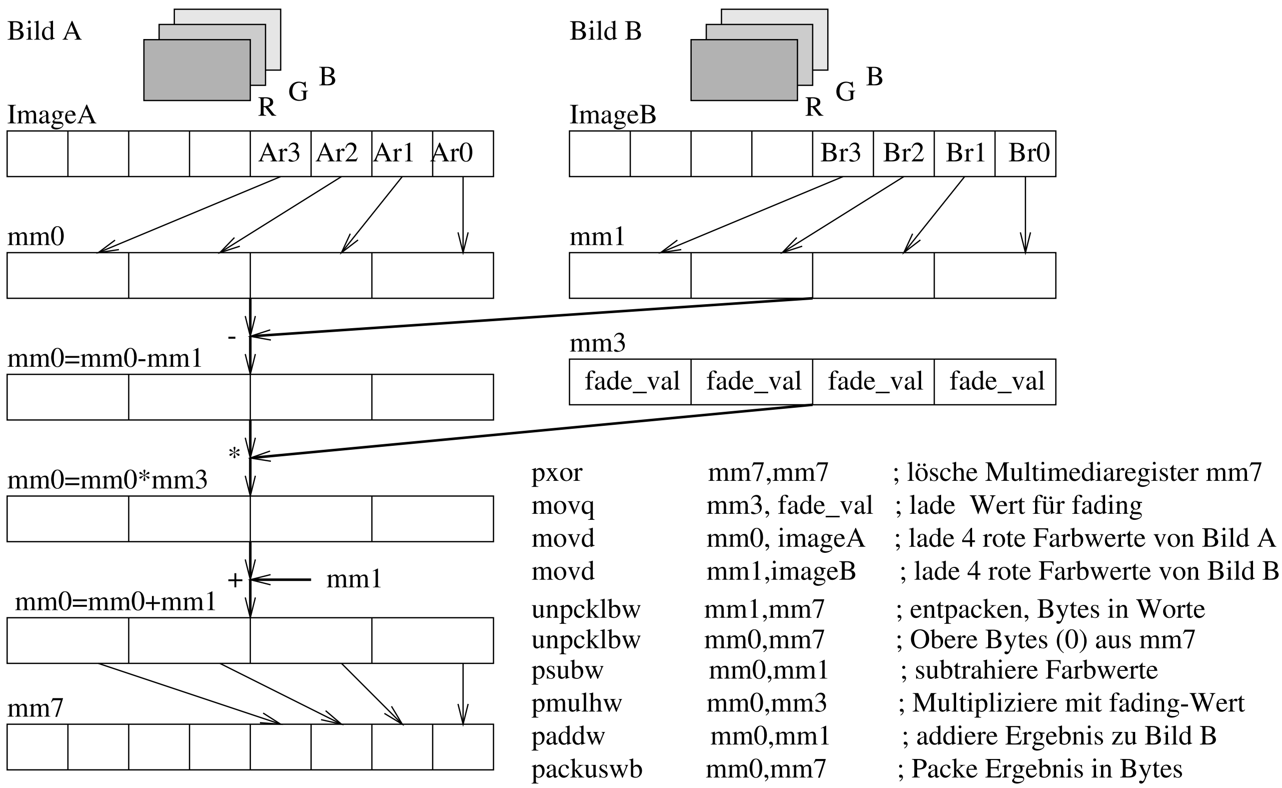


Abbildung 2.18: Skalierte Interpolation (fading) mit MMX-Befehlen

2.4.6 ASIPs

In dem Bestreben, für eingebettete Prozessoren effiziente Hardware anbieten zu können, ist die Verwendung von Applikations-spezifischen Prozessoren (ASIPs) vorgeschlagen worden [ANH+93]. Für diese wird der Befehlssatz aufgrund der Anwendung festgelegt.

2.4.7 Abstrakte Maschinen

Eine weitere Besonderheit stellen abstrakte Maschinen dar, die jeweils einen abstrakten Befehlssatz interpretieren. Solche abstrakten Befehlssätze sind insbesondere für die Programmiersprachen (UCSD-) Pascal, PROLOG, LISP, FORTH, Smalltalk und Java vorgeschlagen worden. Die Interpretation dieser abstrakten Befehlssätze vermeidet es, Programme jeweils in die Befehlssätze verschiedener Maschinen übersetzen zu müssen. Lediglich der Interpreter muss für verschiedene Maschinen jeweils neu erzeugt werden. Nachteilig ist allerdings die niedrigere Ausführungsgeschwindigkeit interpretierter Programme.

2.4.7.1 Java Abstract Machine

Sehr aktuell ist z.Z. (1997) die abstrakte Java-Maschine. Folgende Stichworte kennzeichnen die Sprache Java:

- Objektorientierte Programmiersprache
- Datentypen: byte, short, int, long, float, double, char
- Unterstützt Netzwerk-Programmierung
- Im Hinblick auf Sicherheit entworfen:
  - keine Pointer-Manipulation wie in C, C++.
  - beschränkte Möglichkeit, Informationen über die momentane Umgebung (wie z.B. Benutzernamen) zu erfahren
- Automatische Freispeicherverwaltung
- Multithreading

Java soll über Netze auf alle relevanten Maschinen geladen und dort ausgefürhrt werden können. Aus diesem Grund ist für Java eine abstrakte Maschine (JAM) definiert worden. Java-Programme können überall dort ausgeführt werden, wo diese abstrakte Maschine realisiert ist. Diese abstrakte Maschine besitzt die folgenden Eigenschaften [Dal97]:

- Die JAM ist eine Kellermaschine, die in einem Byte Operationscodes kodiert.
- Typische Befehle der JAM sind: lade integer auf den Keller, addiere die obersten Kellerelemente, starte nebenläufige Ausführung, synchronisiere nebenläufige Ausführung, Befehle zur Realisierung der Objektorientierung
- Der Byte-Code ist sehr kompakt (ca. 1/3 des Speicherbedarfs von RISC-Code). Dadurch ist sie besonders für **eingebettete Systeme** interessant, bei denen Programme zusammen mit den Prozessor auf einem Chip gespeichert werden müssen.
- Die JAM verzichtet weitgehend auf Alignment-Beschränkungen.

Es gibt drei Methoden der Realisierung von JAMs:

- Durch **Interpretation** von JAM-Befehlen in Software.
- Durch Übersetzung in den Maschinencode der aufrufenden Maschine unmittelbar vor der Ausführung *just-in-time compilation*.
- Durch Realisierung einer JAM als "echte" Maschine.

Zur Vermeidung der Performance-Verluste durch Interpretation des Byte-Codes ist die PicoJava-Hardware entwickelt worden. Die PicoJava-Maschine ist die Realisierung einer JAM als echte Maschine. Sie führt den Bytecode als hardwareunterstützten Befehlssatz ausführt. Komplexe Byte-Code werden dabei durch ein Mikroprogramm realisiert [Sun97].

PicoJava besitzt die folgenden Eigenschaften:

- Die obersten Kellerelemente werden im Prozessor in einem schnellen Speicher gehalten (64 Einträge bei der PicoJava I). Durch einen *dribbling*-Mechanismus wird versucht, diesen *stack cache* nie zu voll und nie zu leer laufen zu lassen.
- Die häufigsten Befehle werden in Hardware ausgeführt. Von diesen Befehlen kann pro Takt ein Befehl gestartet werden.
- Etwas komplexere Befehle werden mit einem Mikroprogramm ausgeführt.
- Ganz komplexe Befehle (z.B. Thread-Synchronisation) erzeugen einen Interrupt und werden dann in Software ausgeführt.
- Aufgrund der häufigen Folge eines Arithmetik-Befehls auf einen Lade-Befehl wird diese Kombination dynamisch in einen einzigen Befehl umgewandelt. Auf diese Weise soll die Kellermaschinen inhärente Ineffizienz abgemildert werden.
- Der Keller ist so organisiert, dass aufgerufene Methoden die übergebenen Parameter direkt auf dem Keller verarbeiten können und ein Kopieren in den Speicher nicht notwendig ist (Weiterentwicklung der überlappenden Registerfenster der SPARC).
- PicoJava I soll Sun Microelectronics zufolge fünfmal schneller sein als ein gleich schnell getakteter Pentium mit *just-in-time* Compiler.

2.4.8 Beurteilung von Befehlssätzen

Die folgenden Regeln (z. Tl. aus [HP96] entnommen) helfen bei der Beurteilung von Befehlssätzen:

- Eine variable Befehlswortlänge (wie bei CISC-Rechnern) bzw. kurze Befehlsworte (wie bei DSP-Prozessoren) sind vorteilhaft, wenn es auf eine geringe Codegröße ankommt.
- Eine feste Befehlswortlänge ist vorteilhaft, wenn es v.a. auf die Ausführungsgeschwindigkeit ankommt.
- Condition-Codes können die Codegröße reduzieren, erschweren aber die Fließbandverarbeitung.
- Bei mehreren Speicheroperanden pro Befehl ist es günstig, die Adressierungsart in einem separaten Feld zu kodieren (siehe Motorola 68000).
- Bei höchstens einem Speicheroperanden ist es günstig, die Adressierungsart im Befehlscode zu kodieren.
- Wenn die Effizienz der Implementierung bei hohen Leistungsanforderungen wichtiger ist als die Codekompatibilität (also bei anspruchsvollen eingebetteten Systemen), dann sind VLIW-Maschinen vorteilhaft.
- Wenn die Codekompatibilität höchstes Ziel ist, dann sind abstrakte Maschinen angesagt.

2.4.9 Nicht-Von-Neumann-Maschinen

Alle bislang besprochenen Befehlssätze basieren auf der Definition des von-Neumann-Rechners (siehe Kap. 1). Implizit wurde stets angenommen. dass Befehle sequentiell aufgrund des **Kontrollflusses** abgearbeitet werden. Eine allgemeinere Unterscheidung als die Unterscheidung von Maschinen anhand des Befehlssatzes ist die Unterscheidung anhand des sog. **Operationsprinzips** [Gil81]. Das Operationsprinzip legt fest, welcher Mechanismus der Auslösung der elementaren Operationen zugrunde liegt.

Auch moderne Prozessoren enthalten folgende wesentliche Elemente des von-Neumann-Modells:

1. Speicherzellen als **beliebig wiederverwendbare Container** für Werte. Variable in PASCAL sind letztlich auch nur eine etwas abstraktere Notation für Speicherzellen. Die Mathematik benötigt zur Beschreibung von Funktionen keine Speicherzellen.
2. Der explizite, durch Befehle bestimmte **Kontrollfluß** (diese Rechner heißen daher auch *control flow driven*).

3. Bei den meisten Sprachen kommt noch die explizite Sichtbarkeit der Adressen und ggf. eine explizite Speicherverwaltung hinzu. Diese werden zur Definition mathematischer Funktionen ebenfalls nicht benötigt.

Die folgenden Abschnitt beschreiben einige alternative Maschinenmodelle.

2.4.9.1 Reduktionsmaschinen

Bei Reduktionsmaschinen [Tea82, SK83, Veg84, Kog91] handelt es sich um eine Realisierung einer Maschine, welche die Auswertung funktionaler Programme direkt unterstützt. Reduktionsmaschinen akzeptieren geschachtelte Ausdrücke einer funktionalen Sprache (und keine Sequenzen von Maschinenbefehlen). Sie werten diese Ausdrücke mittels **Baumtransformationen** aus, welche die Ausdrücke reduzieren, bis zum Schluß ein **Wert** erhalten wird. Die Bedeutung eines Programms ist damit durch den abgelieferten Wert gegeben. Durch das Fehlen eines explizit sichtbaren Speichers ist die Verifikation von Programmen vereinfacht. Den theoretischen Hintergrund einer deratigen Programmierung bildet der λ-Kalkül.

Es gibt zwei unterschiedliche Ansätze der Speicherorganisation:

1. *string reduction*

   Jede Operation, die auf eine bestimmte Definition (auf einen bestimmten Teilbaum) zugreift, erhält und bearbeitet eine Kopie des Teilbaums.

   $\rightarrow$ erleichterte Realisierung, schnelle Bearbeitung skalarer Werte, ineffiziente Behandlung gemeinsamer Teilausdrücke (Beispiel: Der Aufwand für die Berechnung der Fibonacci-Zahlen steigt von linear auf exponentiell).

2. *graph reduction*

   Jede Operation, die auf eine bestimmte Definition zugreift, arbeitet über Verweise auf der Original-Definition.

   $\rightarrow$ schwieriger zu realisieren, falls parallel bearbeitet wird; effizient auch für strukturierte Werte und gemeinsame Teilausdrücke; komplizierte Haldenverwaltung (*garbage collection*).

Mögliche Ansätze hinsichtlich der Steuerung der Berechnungs-Reihenfolge:

1. Die *outermost*- oder *demand driven*-Strategie:

   Operationen werden selektiert, wenn der Wert, den sie produzieren, von einer bereits selektierten Operation benötigt wird (Aufrufe von der Wurzel zu den Blättern).

   Erlaubt *lazy evaluation*, d.h. redundante Argumente, z.B. bei bedingten Ausdrücken, brauchen nicht berechnet zu werden.

   Erlaubt konzeptuell unendliche Listen, solange nur endl. Teilmengen referenziert werden (vgl. Streams).

2. Die *innermost*- oder *data-driven*-Strategie:

   Operationen werden selektiert, wenn alle Argumente verfügbar sind.

Vorteile (im wesentlichen Vorteile der funktionalen Programmierung im allgemeinen):

- Programmierung auf höherer Ebene
- kompakte Programme
- keine Seiteneffekte
- kein Aliasing, keine unerwartete Speichermodifikation
- keine Unterscheidung zwischen call-by-name, call-by-value und call-by-reference (siehe PSÜ) notwendig
- einfachere Verifikation, da nur Funktionen benutzt werden

- das von-Neumann-Modell von Speicherzellen und Programmzählern ist überflüssig
- beliebige Berechnungsreihenfolge für Argumente (mit Ausnahme von Problemen bei nicht-terminierenden Berechnungen)
- vereinfachte Parallelverarbeitung
- Debugging einfach: Trace des aktuellen Ausdrucks (dies gilt für die "normale" funktionale Programmierung nicht)

2.4.9.2 PROLOG-Maschinen

Zur direkten Realisierung von logischen Programmiersprachen gibt es v.a. PROLOG-Maschinen.

Hierbei erfolgt meist Übersetzung von PROLOG in den Code der WAM (Warren Abstract Machine). Die Befehle der WAM werden sodann entweder von Maschinenprogrammen interpretiert, als Maschinenbefehle realisiert oder das komplette PROLOG-Programm wird in Maschinenbefehle übersetzt.

Man kann zwischen verschiedenen Klassen von PROLOG-Maschinen unterscheiden:

- Sequentielle Maschinen mit strenger Wahrung der PROLOG-Semantik
- Parallele Maschinen mit unterschiedlicher Semantik

Beispiele von Maschinen:

- Viele Entwürfe innerhalb des japanischen *5th Generation Projekts.*
- PRIPs = Entwurf der Projektgruppe PRIPs (Entwurf eines PROLOG-Chips) an der Uni Dortmund, Informatik XII. 1993 gefertigt und getestet.

2.4.9.3 Datenflussmaschinen

Bei diesen veranlaßt die Verfügbarkeit von Daten die Ausführung von Operationen [Tea82, Tre84, Den80].

Beispiel:

> z = (x + y) * (x - y); Darin sind x,y und z lediglich Benennungen für Werte. Siehe Abb. 2.19.

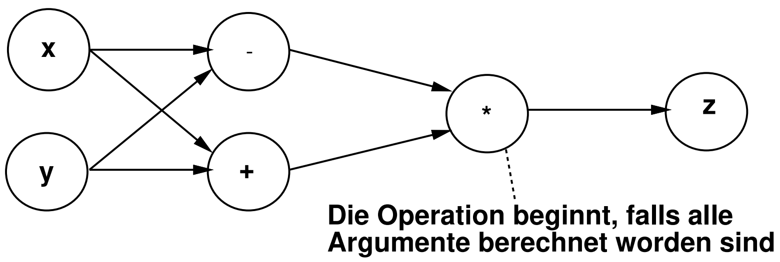


Abbildung 2.19: Datenflußgraph

Modellierung mit Marken:

> Gültige Daten werden als Marken dargestellt. Eine Operation kann ausgeführt werden, falls alle ihre Argumente markiert sind. Potenziell viele Operationen gleichzeitig!

Darstellung der Befehle in der Maschine (Dennis):

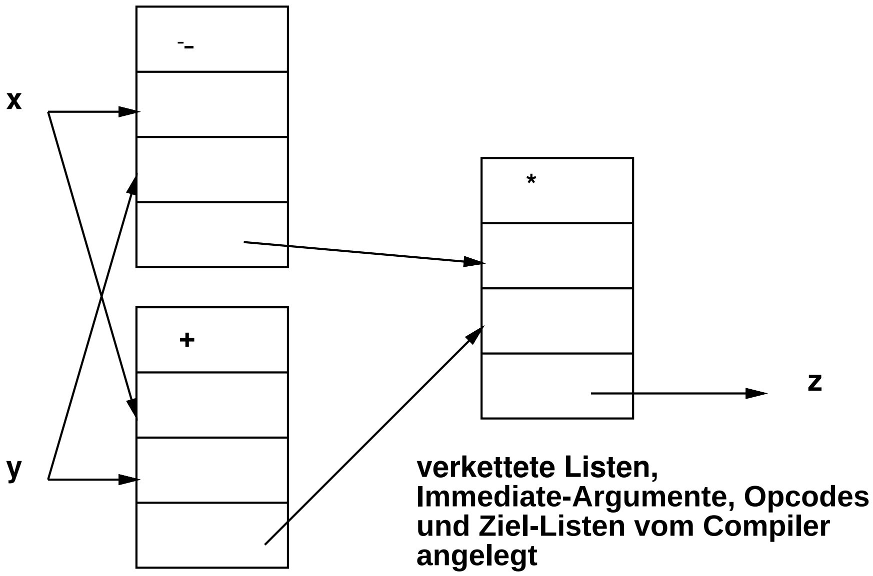


Abbildung 2.20: Implementierung nach Dennis für einfache Ausdrücke

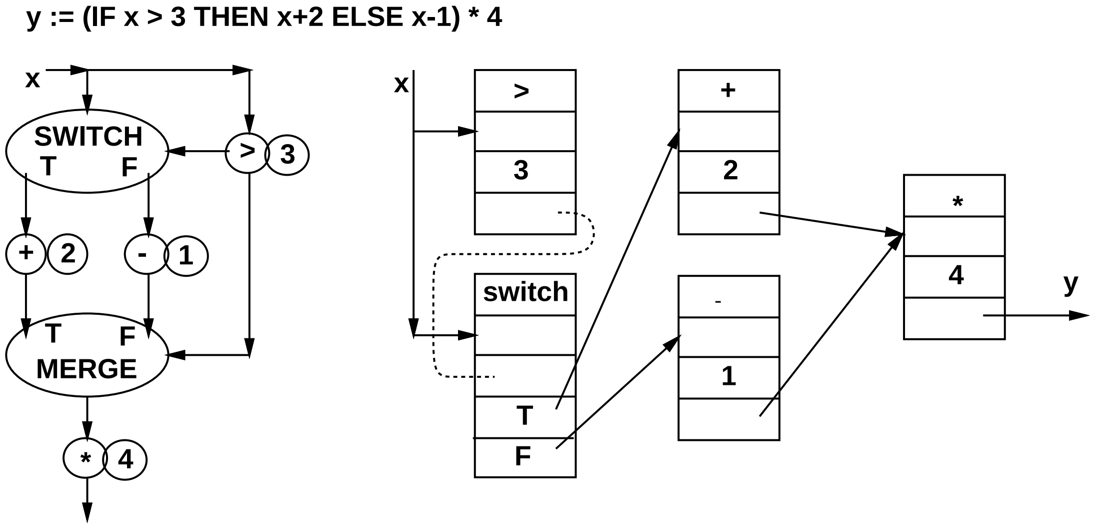


Abbildung 2.21: Implementierung nach Dennis für bedingte Ausdrücke

Als Tupel (Opcode, Plätze für Argumente, Ziel-Liste). Die Ziel-Liste ist die Liste der Tupel, die das Ergebnis als Argument benötigen, siehe Abb. 2.20 bis Abb. 2.21.

Vorteile der Datenflußrechner sind:

- eingebaute Parallelität
- eingebaute Synchronisation
- Adressen sind nach außen nicht sichtbar
- vorteilhaft bei gemeinsamen Teilausdrücken
- beliebige Reihenfolge der Abarbeitung ausführbereiter Befehle

Während vieler Jahre sah es so aus, als wäre die Entwicklung von Datenflussrechnern in einer Sackgasse gemündet. Ein Grund dafür war sicherlich der zusätzliche Aufwand durch die Verwaltung der Marken usw.

Inzwischen sind die Fließbänder normaler von-Neumann-Rechner wegen des Leistungsdrucks so komplex geworden, dass ein Aufwandsvergleich nicht mehr unbedingt zum Nachteil der Datenflussrechner ausfällt. Sie können relativ leicht asynchron realisiert werden und kommen dadurch mit wenig Strom aus. Aus diesem Grund ist offenbar geplant, eine Weiterentwicklung [] dieser Rechner in Videokameras der Fa. Sharp zur Datenreduktion einzusetzen.

2.4.9.4 Weitere Nicht-Von-Neumann-Maschinen

Einige in der Regel auf der Basis elektrischer Prinzipien realisierter Nicht-Von-Neumann-Maschinen, wie z.B. Neuronale Netze, können wir in dieser Vorlesung aus Zeitgründen nicht behandeln.

2.4.9.5 DNA/RNA- und Quantenrechner

DNA/RNA- und Quantenrechner basieren auf anderen Wirkungsprinzipien als die bislang erwähnten elektronischen Rechner. Beide Formen von Rechnern leisten eine Parallelverarbeitung weit jenseits der z.Z. parallel benutzten Prozessorzahlen.

Im Falle von RNA/DNA-Rechnern werden Informationen, beispielsweise mögliche Lösungen, in RNA- bzw. DNA-Strängen kodiert und mit mikrobiologischen Methoden gefiltert. Wegen der großen Zahl von gleichzeitig bearbeitbaren Strängen können riesige Lösungsmengen gleichzeitig auf zulässige Lösungen untersucht werden [Pet00]. Ein einfacher 10-Bit RNA-Rechner ist bereits in Betrieb [Joh00].

Im Falle von Quantenrechnern können sich Nullen und Einsen verschiedener Lösungen in einem Register überlagern. Durch die Form der Parallelarbeit gerät die klassische Komplexitätstheorie an ihre Grenzen. Es wurde nachgewiesen, dass die RSA-Kodierung mit Quantenrechnern effizient geknackt werden kann. Es wird erwartet, dass ein 10-Bit-Quantenrechner in zwei Jahren gebaut werden kann [SR00].

2.5 Unterbrechungen

Zur externen Architektur gehört auch noch die logische Sicht auf die Unterbrechungsverarbeitung (engl. *interrupt structure*).

Während der normalen Bearbeitung eines Programmes kann es in einem …-System immer wieder zu **Ausnahme-** *Situationen (engl. exceptions) kommen, die vom Prozessor eine vorübergehende Unterbrechung oder aber einen endgültigen Abbruch des Programms, also in jedem Fall eine Änderung des normalen Programmablaufs verlangen. Diese Ausnahme-Situationen können einerseits durch das aktuell ausgeführte Programm selber herbeigeführt werden, …, andererseits aber auch ohne jeglichen Bezug zum Programm sein. Ursachen sind im ersten Fall das Vorliegen von Fehlern im System oder im Prozessor selbst, im zweiten Fall der Wunsch externer Systemkomponenten, vom Prozessor "bedient" zu werden. Durch einen Abbruch reagiert das System stets auf schwere Fehler, ….* (Zitat Bähring).

Im ersten Fall spricht man von **synchronen Unterbrechungen**, im zweiten Fall von **asynchronen Unterbrechungen**.

Ablauf von Programmunterbrechungen:

- **Erzeugen von Unterbrechungsanforderungen**

  Aus Zeitgründen werden die Anforderungen meist nebenläufig zur Bearbeitung von Programmen erzeugt.

- **Abprüfen von Unterbrechungsanforderungen**
- **Beenden der gerade bearbeiteten Operation**, falls dies zulässig ist.
- **Retten des Prozessorzustandes** (PC, CC, PSW, A, D)
- **Verzweigen aufgrund der Unterbrechungsursache**

- ggf. **Senden einer Bestätigung** (interrupt acknowledge)
- **Ausführen der Interrupt-Routine**
- **Beenden der Interrupt-Routine** durch einen *return from interrupt-* Befehl
- **Restaurieren des Prozessorzustandes**

Tabelle 2.15 enthält Klassen von Unterbrechungen[8].

Klasse	synchron/ asynchron	Auslösung	maskierbar	Abbruch im Befehl	Fortsetzung
E/A-Gerät	asynchron	System	nein (2)	nein (1)	ja
BS-Aufruf	synchron	Prozess	nein	nein	ja
TRACE-Befehl	synchron	Prozess	ja	nein	ja
Überlauf	synchron	Prozess	ja	ja	nein
Timer	asynchron	System	ja (3)	nein	ja
Seitenfehler	synchron	System	nein	ja (!)	ja (!)
Schutz-Verletz.	synchron	System	nein	ja	nein
Unimplem.Befehl	synchron	System	nein	ja	nein
Hardware-Fehler	beides	beides	nein	ja	Wiederhol.
Netzausfall	asynchron	System	nein	?	nein

(1) Bei sehr langen Befehlen (move block) Abbruch und Fortsetzung.
(2) Kurzfristig maskierbar.
(3) Kann zu falschen Zeiten führen.

Tabelle 2.15: Unterbrechungsklassen (Auszug)

2.6 Realisierung mehrerer virtueller Maschinen per Prozessumschaltung

2.6.1 Aufgaben

Viele der heute von Rechensystemen zu lösenden Aufgaben lassen sich in Form von mehreren Prozessen beschreiben. Auf diese Weise können unabhängige oder lose gekoppelte Aufgaben jeweils getrennt beschrieben werden.

Da aus Effiziengründen nicht für jeden Prozess ein eigener Prozessor zur Verfügung gestellt werden kann, wird in den meisten Betriebssystemen durch Software ein Mehrprozeßbetrieb realisiert. Durch diese Software soll es für jeden Prozess so aussehen, als gehöre ihm allein die Maschine (virtuelle Maschine). Diese Software heißt *Dispatcher*. Der Dispatcher übernimmt v.a. die kurzfristige Zuteilung des Prozessors. Zu den Aufgaben des Dispatchers gehört u.a. der **Prozesswechsel** mit dem Umladen der Prozesszustandsinformation:

- Umladen der allgemeinen Register, des Programmzählers, des Condition-Code Registers
- Umladen der Adresse der Seitentabelle oder der Grenzregister
- ggf. Erklärung der Ungültigkeit des Caches
- Umladen von Timern

[8] Quelle: [HP96], Abb. 5.11.

2.6.2 Prozesszustände

Prozesse können Übergänge zwischen Zuständen gemäß der Abb. 2.22 ausführen[9].

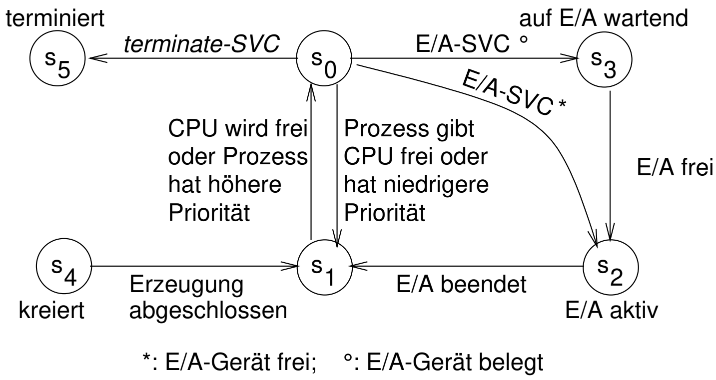


Abbildung 2.22: Übergänge zwischen Prozesszuständen

Die Übergänge sind ereignisgesteuert (und nicht etwa durch zyklische Abfragen!). Auslösung der Übergänge:

- E/A-Aufträge: Diese müssen über das BS abgewickelt werden. Dazu Ausführung eines Maschinen-Befehls, der einen Interrupt auslöst (SVC=supervisor call). Damit wird eine BS-Routine gestartet und damit wieder der Dispatcher. Direkte E/A-Maschinen-Befehle sind verboten (privilegiert)!
- Timer-Interrupts
- Ausführung eines *terminate*-Befehls (BS-Aufruf)
- Abschluß der Prozess-Initialisierung
- Abschluß eines E/A-Auftrags
- Freiwillige Freigabe des Prozessors

Grundsätzliche Unterscheidung zwischen zwei Klassen von Verfahren:

- Verfahren ohne Vorrang-Unterbrechung (engl. *non-preemptive scheduling*):

  Prozesse behalten die CPU, bis sie entweder einen E/A-Auftrag geben oder bis sie beendet sind.

  Problem: Prozess ohne E/A-Auftrag kann die übrigen Prozesse lange warten lassen, sie ggf. sogar vollständig blockieren.
- Verfahren mit Vorrang-Unterbrechung (engl. *preemptive scheduling*). [10]

2.6.3 Prinzip des Dispatchers

Die folgenden Flußdiagramme beschreiben die Arbeitsweise des Dispatchers für seine verschiedenen Operationen. Darin werden die folgenden Bezeichnungen benutzt:

- D_j : die Geräte eines Rechensystems
- Q_i : die Prozesse eines Rechensystems
- $D(Q)$: gegenwärtig von Q belegtes Gerät

[9] Quelle:Hayes

[10] Derartige Verfahren sind in Betriebssystemen schon einige Jahrzehnte üblich gewesen, bevor sie in *Windows 95* eingeführt wurden.

- $T(Q)$: der gegenwärtige Zustand des Prozesses Q (Zustände wie in Abb. 2.22).
- T_i: die Menge der Prozesse, die sich im Zustand i befinden.
- $p : \{Q_i\} \rightarrow \{0, ..., k\}$ die Abbildung, die jedem Prozess seine gegenwärtige Prorität zuordnet

Die Abbildungen 2.23 und 2.24 beschreiben die Realisierungen der verschiedenen Dispatcher-Operationen.

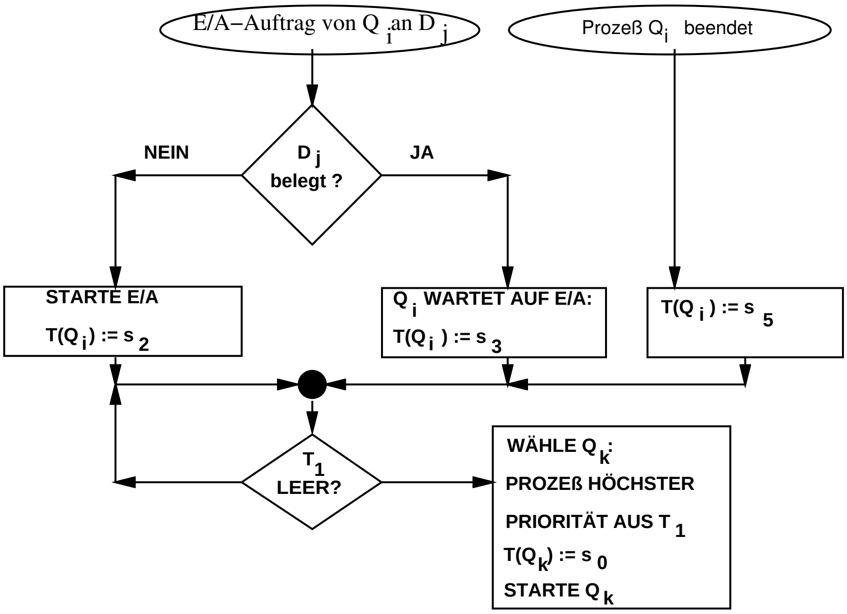


Abbildung 2.23: Behandlung von E/A-Aufträgen und Prozess-Terminierungen

Der Dispatcher und die Prozesse werden auf demselben Prozessor ausgeführt.

Der Dispatcher selbst ist in der Regel kein Prozess, er wird in der Regel im BS-Adreßraum ausgeführt. Beim Start müssen natürlich trotzdem Register gerettet und restauriert werden.

2.6.4 Prozesskontrollblock

Die Verwaltungsdaten für den Dispatcher sind jeweils im Prozess-Kontrollblock enthalten (PCB). Dieser wiederum enthält denjenigen Teil des Prozessorzustands, der bei einem Prozesswechsel gesichert werden muss. Diese Sicherung erfolgt bei Intel-Prozessoren in dem sog. *task state segment* (TSS). Abb. 2.25 ehält ein Beispiel für einen Prozessor der Intel-Familie. Die Worte mit den Adressen 2 bis 12 enthalten Stack-Pointer für unterschiedliche Privilegierungs-Stufen. Die Worte mit den Adressen 18 bis 32 enthalten gesicherten Werte der allgemeinen Register. Außerden müssen u.a. noch die Segment-Selektoren, die Condition-Codes (Flags) und der Zeiger auf die LDT gerettet werden.

Zum Zustand eines Prozesses gehört neben dem Inhalt der Hardware-Register auch weitere Verwaltungsinformation innerhalb des PCBs. Der PCB enthält z.B.:

- Den Prozesszustand (falls nicht über die Warteschlange bestimmt)
- Die Prozesskennung
- Den geretteten Inhalt des Condition-Code Registers
- Den geretteten Inhalt der allgemeinen Register
- Speicherverwaltungsdaten
- Abrechnungsdaten

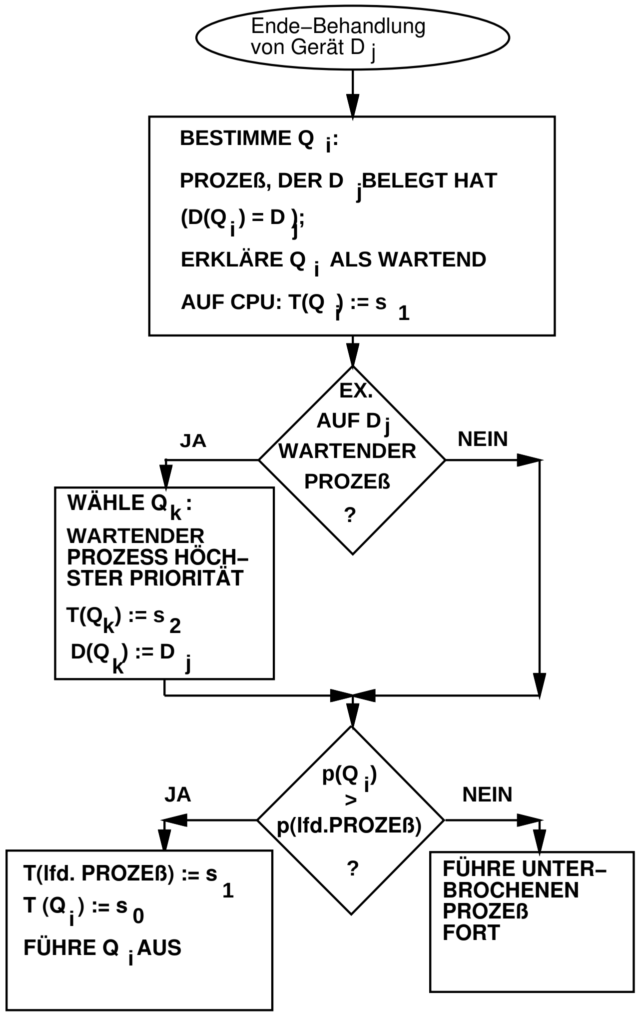


Abbildung 2.24: Ende-Behandlung von E/A-Aufträgen

- Priorität
- Prozessvariable
- E/A-Status
- Zeiger auf weitere PCBs.

Häufig ist außerhalb der Prozessverwaltung eine Datenübertragung einzelner Worte/Bytes für asynchron unterbrechende Geräte erforderlich. Danach muss zwischen zwei Sorten von E/A-Interrupts unterschieden werden:

1. Interrupt zur Datenübertragung (siehe Kapitel 5).
2. Blockende-Behandlung

   Ein auf Beendigung einer E/A-Operation wartender Prozess kann potentiell fortgeführt werden. Der entsprechende Interrupt führt zum Aufruf des Dispatchers, der eine Prozessumschaltung vornehmen kann.

Es muss einen Mechanismus geben, der eine Unterscheidung zwischen den beiden Sorten erlaubt.

	Wort
Task LDT Selektor	42
DS-Selektor (Datensegment)	40
SS-Selektor (Stack-Segment)	38
CS-Selektor (Code-Segment)	36
ES-Selektor (Extra-Daten-S.)	34
DI	32
SI	30
BP	28
SP	26
BX	24
DX	22
CX	20
AX	18
Statusregister (Flag-Register)	16
IP Instruction Pointer	14
SS Stackpointer	12
SP fuer CPL=2	10
SS Stackpointer	8
SP fuer CPL=1	6
SS Stackpointer	4
SP fuer CPL=0	2
BACK LINK SELECTOR to TSS	0

Bit 15 0

Abbildung 2.25: TSS in der Intel-Prozessorfamilie

Kapitel 3

Mikroarchitektur

Computers in the future may weigh no more than 1.5 tons

[Popular mechanics, 1949]

Als nächstes werden wir uns mit der Realisierung von Befehlssätzen mittels Mikroarchitekturen beschäftigen.

3.1 Symbole der internen Rechnerarchitektur

Zur Darstellung der internen Rechnerarchitektur werden wir die verschiedene Schaltungssymbole benutzen.

Abb. 3.1 zeigt die Darstellung von Leitungen.

Abbildung 3.1: Graphische Darstellung von Leitungen

Funktionseinheiten können definiert werden als durch Aufgabe oder Wirkung abgrenzbare Gebilde. Sofern nicht eines der übrigen Symbole Anwendung finden kann, werden sie durch Rechtecke (siehe Abb. 3.2) und die jeweils benötigten Leitungen dargestellt.

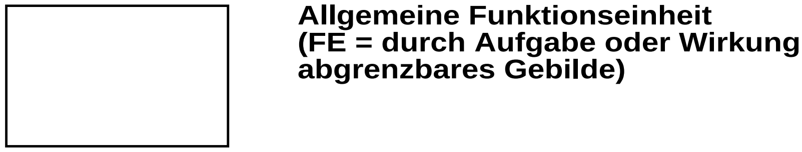


Abbildung 3.2: Graphische Darstellung von allgemeinen Funktionseinheiten

Abb. 3.3 zeigt die Darstellung der häufig benötigten Multiplexer, die der Auswahl von Daten dienen, sowie von Dekodern, welche Bitvektoren in andere Bitvektoren umkodieren.

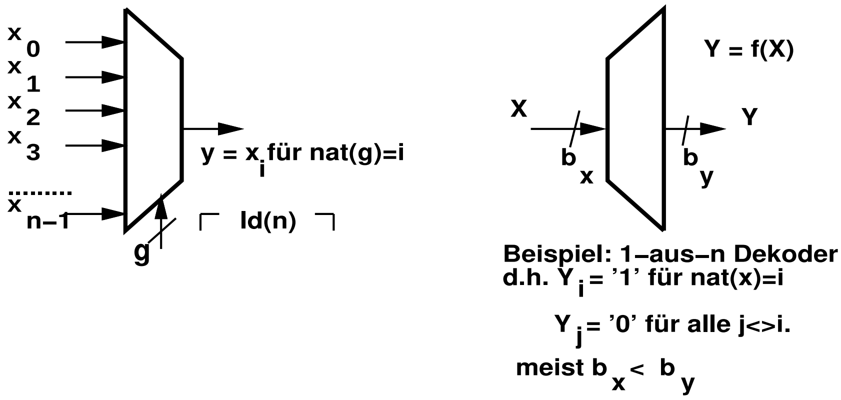


Abbildung 3.3: Multiplexer und Dekoder

Den speziellen Fall der Prioritätsencoder zeigt die Abb. 3.4.

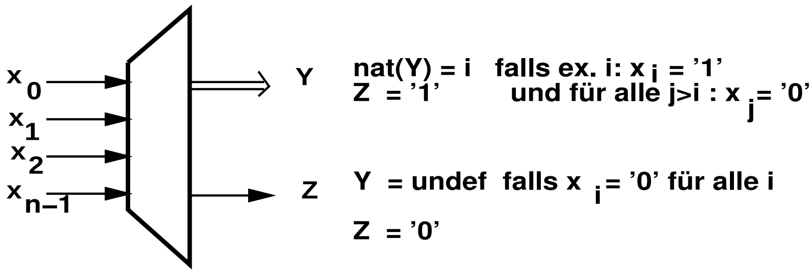


Abbildung 3.4: Prioritätsencoder

Die Abb. 3.5 zeigt links Bausteine, die zweistellige Operationen ausführen, also beispielsweise aus zwei Bitvektoren einen neuen Bitvektor berechnen, welcher die Summe darstellt. Ein Kontrolleingang erlaubt es, eine von mehreren Operationen auszuwählen. So kann mit diesem Eingang z.B. entschieden werden, ob addiert oder subtrahiert werden soll.

Der rechte Teil der Abbildung zeigt ein Register.

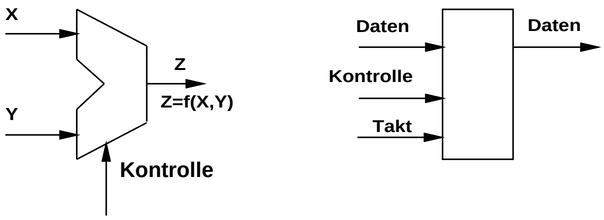


Abbildung 3.5: Dyadischer Operator und Register

Abb. 3.6 zeigt zwei Darstellungen von Speichern.

Im rechten Teil der Abbildung wird der Adressdekoder, der der Auswahl der einzelnen Speicherzellen dient, explizit repräsentiert. Den Adressdekoder von Speichern werden wir benutzen, um klarzustellen, dass gewisse interne Tabellen **adressiert** werden (siehe Kap. 4). Wir werden den Dekoder auch explizit darstellen, damit der Adresseingang von Speichern leichter erkannt werden kann. Dies ist in der Literatur kaum üblich. In den mündlichen Prüfungen hat sich aber herausgestellt, dass so manche Missverständnisse vermieden werden können.

Die Datenleitungen können in unterschiedlichen Variationen ausgeführt sein. Sowohl separate Ein- und Ausgänge als auch in der Richtung umschaltbare Datenleitungen sind in Gebrauch.

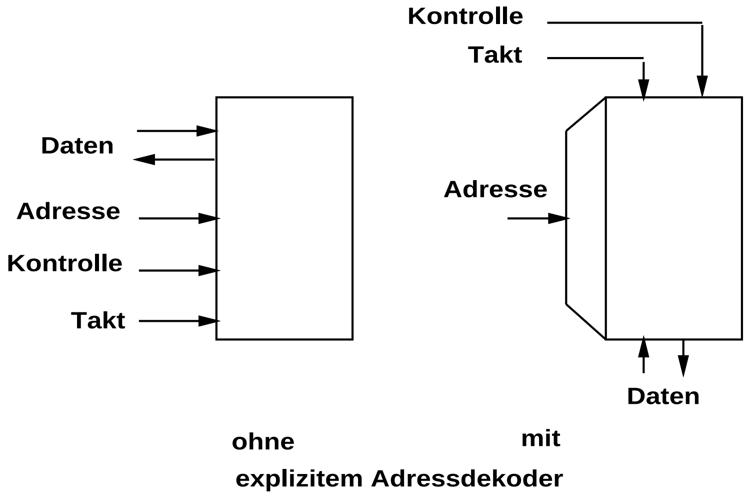


Abbildung 3.6: Speicher

3.2 Realisierung der Operationen elementarer Datentypen

3.2.1 Bitvektoren

Realisiert werden Schiebeoperationen auf Bitvektoren um n Stellen häufig durch **Barrelschifter**. Abbildung 3.7 zeigt den Elementarbaustein für das Schieben mit Barrelshiftern um 0-3 Bit bei 4 Ausgängen.

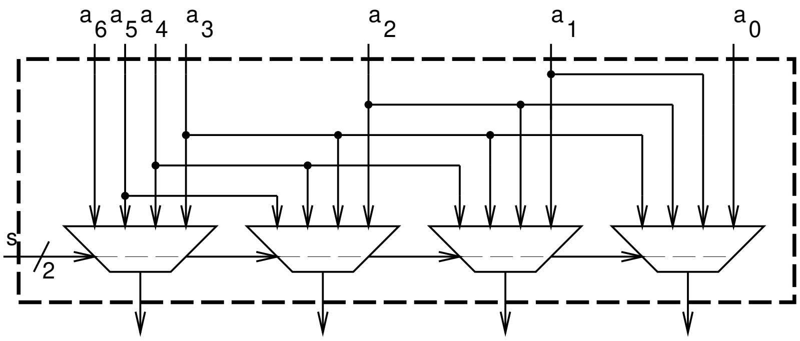


Abbildung 3.7: Elementarbausein, 4 Ausgänge, Schieben um 0-3 Stellen

Durch Kombination dieser Elementarbausteine kann man die Anzahl der Ausgänge erhöhen, siehe Abb. 3.8.

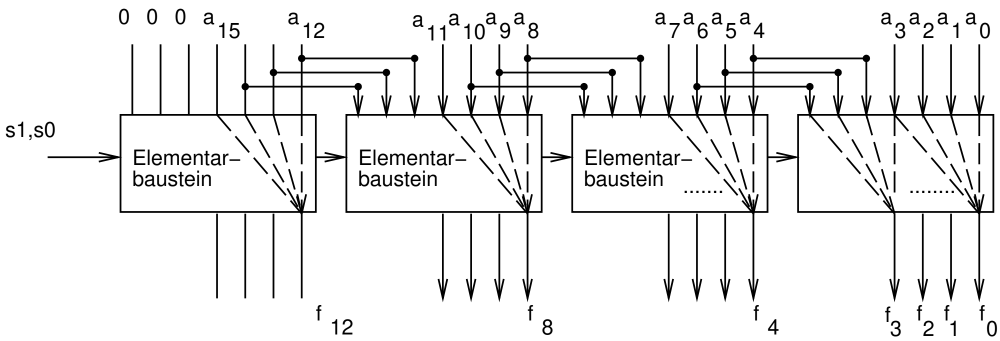


Abbildung 3.8: Barrelshifter mit 16 Ausgängen

Schließlich lässt sich durch geschickte Verschaltung auch die Anzahl der Stellen erhöhen, um die geschoben wird, siehe Abb. 3.9.

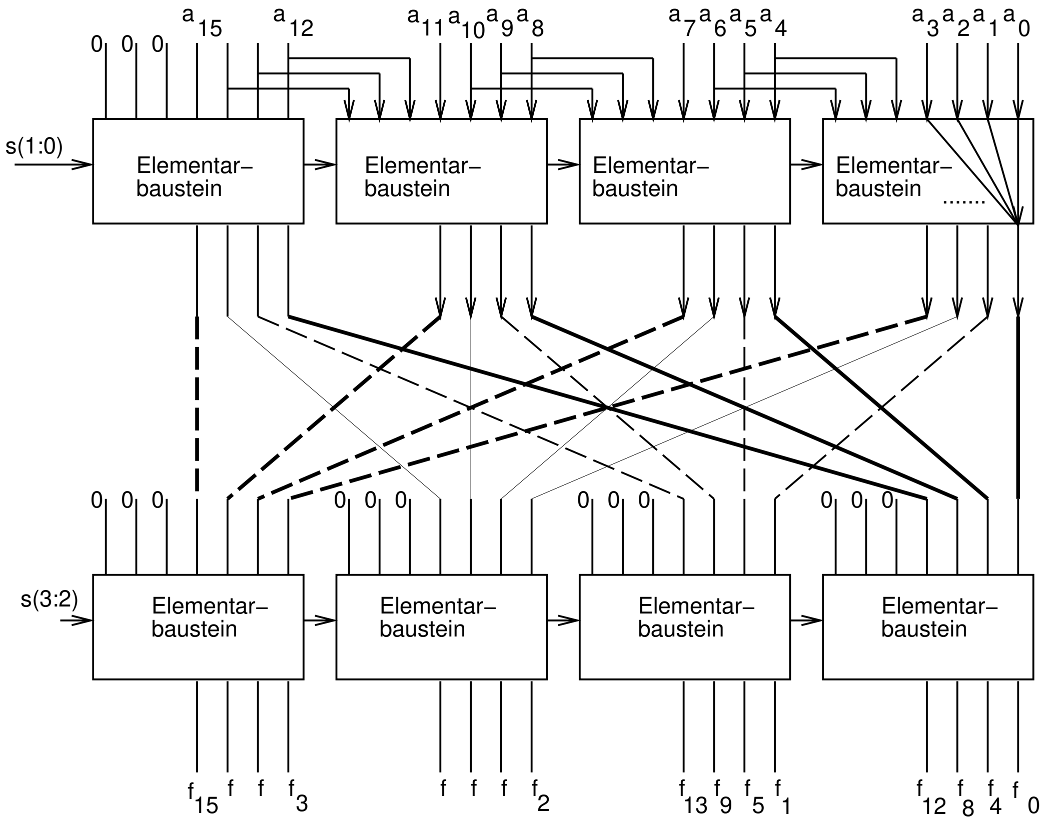


Abbildung 3.9: Barrelshifter für das Schieben um 0 bis 15 Stellen

Mit diesem kombinatorischen Schaltungselement kann ein Bitvektor um 0-15 Stellen verschoben werden.

Anhand dieser Schaltungen sollte deutlich werden, dass die Operation "Schieben um n Stellen" durch eine kombinatorische Schaltung realisiert werden kann, also nicht auch n Schritte einer sequentiellen Schaltung benötigt. Dies ist eigentlich selbstverständlich, wird aber aufgrund von Einschränkungen in manchen Maschinenbefehlssätzen leicht vergessen.

Barrel-Shifter sind (in Kombination mit Prioritätsencodern) sehr gut zur Beschleunigung des Booth-Algorithmus (s.u.) sowie zur Normalisierung von Gleitkommazahlen geeignet.

3.2.2 Natürliche Zahlen

3.2.2.1 Realisierungen der Addition

Als erste Operation des Datentyps betrachten wir die Realisierung der Addition. Das Basiselement ist der Volladdierer (siehe Abb. 3.10).

Die Addition von größeren Wortbreiten ist mit dem Ripple-Carry-Addierer möglich (siehe Abb. 3.11).

Das Weiterleiten der Überträge ist in der VLSI-Technik besonders einfach möglich, indem MOS-Transistoren wie Schalter im Signalweg eingesetzt werden. In diesem Fall leitet ein MOS-Transistor den Übertrag selektiv weiter. Die entsprechende Schaltung nach Abb. 3.12 heisst Manchester Carry-Chain-Addierer. In der Abbildung werden auch die Hilfssignale *kill, propagate* und *generate* definiert, die wir später noch benötigen werden.

Diese Schaltung wurde mit bipolaren TTL-Schaltungen nicht verwendet. Mit der stärkeren Verwendung von

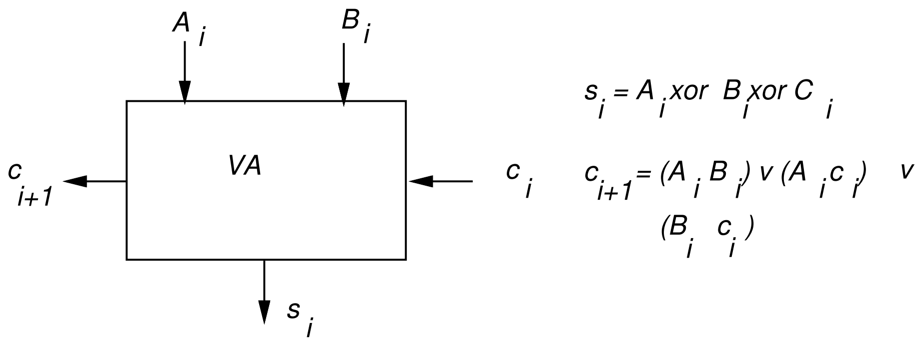


Abbildung 3.10: Volladdierer

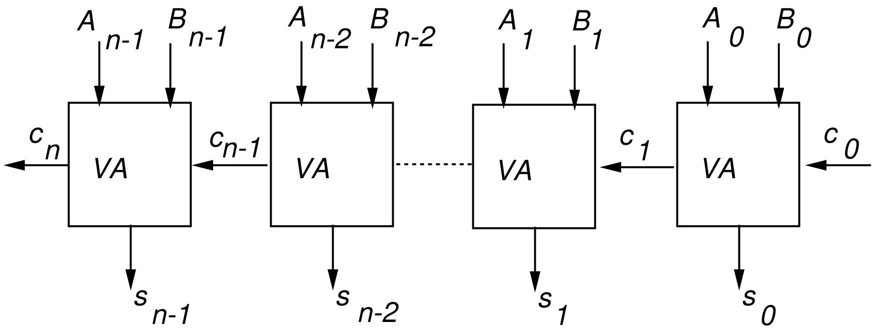


Abbildung 3.11: Ripple-Carry-Addierer ($n-1 = A'left$)

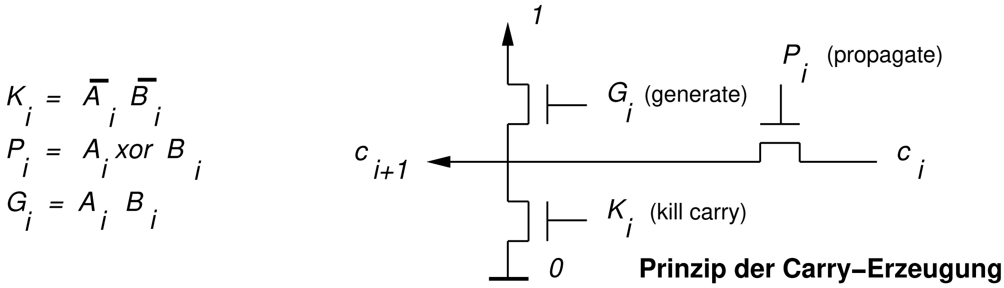


Abbildung 3.12: Manchester-Carry-Stufe

MOS-Transistoren Anfang der 80er-Jahre wurde sie aufgrund ihrer Einfachheit und ihres regulären Layouts populär (siehe z.B. [MC80]). Nachteilig ist, dass die Zeit zur Addition von Worten linear von der Wortlänge abhängt.

Schneller ist die Carry-Look-Ahead-Schaltung. Bei dieser erfolgt eine 2-stufige Berechnung des Übertrags für eine bestimmte Anzahl (häufig: 4) von Bits. Die folgenden Formeln zeigen, wie die Überträge am Bit 1, 2, 3 und 4 aus den "propagate"- und "generate"-Signalen der einzelnen Bitstellen berechnet werden können:

$$\begin{aligned}
c_1 &= G_0 \vee P_0 c_0 \\
c_2 &= G_1 \vee P_1 c_1 = G_1 \vee P_1 G_0 \vee P_1 P_0 c_0 \\
c_3 &= G_2 \vee P_2 c_2 = G_2 \vee P_2 G_1 \vee P_2 P_1 G_0 \vee P_2 P_1 P_0 c_0 \\
c_4 &= G_3 \vee P_3 c_3 = G_3 \vee P_3 G_2 \vee P_3 P_2 G_1 \vee P_3 P_2 P_1 G_0 \vee P_3 P_2 P_1 P_0 c_0
\end{aligned}$$

Diese Gleichungen werden z.B. zum Aufbau eines 4-Bit Addierers ausgenutzt (siehe Abb. 3.13). Die Rechtecke in der unteren Reihe realisieren gerade die soeben angegebenen Gleichungen. Man beachte, dass die

Signale P_i und G_i in zwei Gatterlaufzeiten aus A_i und B_i ausgerechnet werden können. Mit zwei weiteren Gatterlaufzeiten sind dann die c_i bekannt.

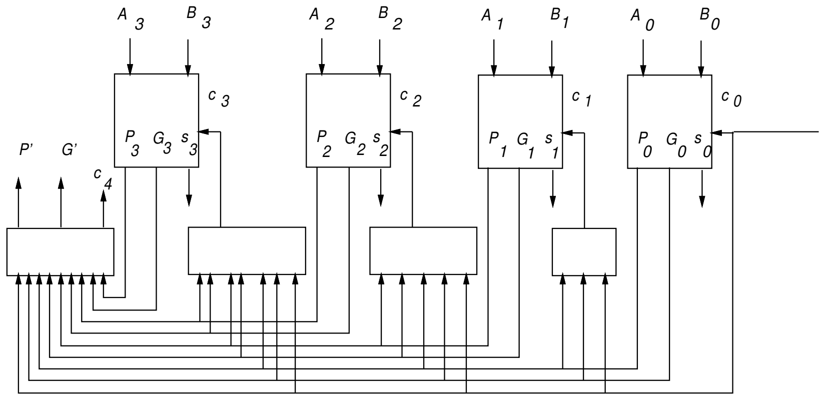


Abbildung 3.13: 4-Bit Carry-Look-Ahead-Addierer

Diese 4-Bit Bausteine kann man nun zu 16-Bit-Addierern zusammensetzen (siehe Abb. 3.14).

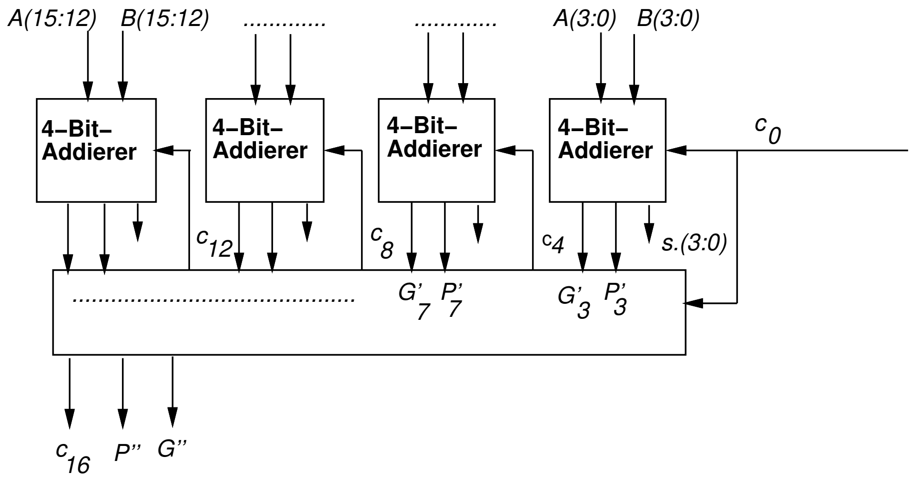


Abbildung 3.14: 16-Bit Carry-Look-Ahead-Addierer

Das untere Rechteck realisiert wieder die oben angegebenen Gleichungen für die Überträge. Der zeitliche Ablauf ist nun der folgende:

1. Alle 4-Bit-Addierer berechnen P_i' und G_i' ($\leq$ 4 Gatterlaufzeiten)
2. Der Carry-Look-Ahead (CLA)-Baustein (unteres Rechteck) berechnet alle c_i an seinem Ausgang ($\leq$ 2 Gatterlaufzeiten).
3. Alle 4-Bit-Addierer berechnen s_i. Diese Berechnung kann in 2 Gatterlaufzeiten abgeschlossen werden, wenn die Summenbits für die beiden Fälle (eingehender Übertrag = '0') und (eingehender Übertrag = '1') vorab berechnet und bei Eintreffen des Übertrags nur noch ausgewählt werden müssen.

Bei bis zu 64 Bit Wortbreite reicht eine dritte CLA-Stufe aus, bei bis zu 256 Bit reicht eine vierte CLA-Stufe usw.. Bei einer Vervierfachung der Wortbreite kommt jeweils eine CLA-Stufe hinzu, d.h. die Verzögerungszeit ist proportional zum **Logarithmus** der Wortbreite. Man kann zeigen, dass dies für bestimmte Einschränkungen beim Aufwand optimal ist (siehe [Weg89]).

3.2.2.2 Subtraktion

Für das Zweierkomplement gilt: $-B = \overline{B} + 1$, also $A - B = A + \overline{B} + 1$. Die Subtraktion ohne Borgebit kann also realisiert werden, indem B negiert und der Übertragseingang auf 1 gesetzt werden (siehe Abb. 3.15, vgl.

Bähring Abb. L 1.6-1).

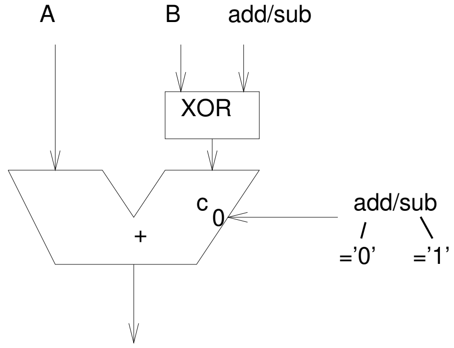


Abbildung 3.15: Realisierung der Subtraktion mit einem Addierer

Addition und Subtraktion werden in der Praxis in sog. arithmetisch/logischen Einheiten (ALUs) realisiert. Eine bekannte ALU ist die "74181", die in vielen Technologien realisiert wird. Mit Hilfe von Steuersignalen S3,S2,S1,S0 und M können die in Abb. 3.16 aufgeführten Funktionen ausgewählt werden.

Steuerkode				$M='1'$ (logische Op.)	$M='0'$ (arithm. Op.)	$M='0'$
S3	S2	S1	S0		$c_0='0'$	$c_0='1'$
'0'	'0'	'0'	'0'	$F:=\overline{A}$	$F:=A$	$F:=A+1$
'0'	'0'	'0'	'1'	$F:=\overline{A \vee B}$	$F:=A \vee B$	$F:=(A \vee B)+1$
'0'	'0'	'1'	'0'	$F:=\overline{A} \wedge B$	$F:=A \vee \overline{B}$	$F:=(A \vee \overline{B})+1$
'0'	'0'	'1'	'1'	$F:=0$	$F:=-1_{10}$	$F:=0$
.	.	.	.	...	$F:=...$	$F:=...$
'0'	'1'	'1'	'0'	$F:=A \neq B$	$F:=A-B-1$	$F:=A-B$
.	.	.	.	...	$F:=...$	$F:=...$
'1'	'0'	'0'	'1'	$F:=A=B$	$F:=A+B$	$F:=A+B+1$
.	.	.	.	...	$F:=...$	$F:=...$
'1'	'1'	'1'	'0'	$F:=A \vee B$	$F:=(A \vee \overline{B})+A$	$F:=(A \vee \overline{B})+A+1$
'1'	'1'	'1'	'1'	$F:=A$	$F:=A-1$	$F:=A$

Abbildung 3.16: Von der ALU 74181 angebotene Operationen

3.2.2.3 Multiplikation

Die Multiplikation kann für natürliche Zahlen nach der für die Basis 2 modifizierten Schulmethode erfolgen. Anhand eines Beispiels sei das Vorgehen erläutert:

10 *	1010	A
13	1101	B
	00000000	$P_0 = 0$
	1010	
	00001010	$P_1 = P_0 + B_0 * 2^0 * A$
	0000	
	00001010	$P_2 = P_1 + B_1 * 2^1 * A$
	1010	
	00110010	$P_3 = P_2 + B_2 * 2^2 * A$
	1010	
= 130	10000010	$P_4 = P_3 + B_3 * 2^3 * A$

Das Partialprodukt P_{i+1} kann also $\forall i = 0..(n-1)$ wie folgt gebildet werden:

$P_{i+1} := P_i + B_i * 2^i * A$ mit $P_0 = 0$.

Parallele Multiplizierer

Auch Multiplikationen können bei entsprechendem Hardware-Aufwand kombinatorisch ausgeführt werden. Man betrachte die folgende Formel für die Multiplikation natürlicher Zahlen:

$$P_N = \sum_{i=0}^{n-1} A_i * 2^i * B = \sum_{i=0}^{n-1} 2^i \sum_{j=0}^{n-1} A_i * B_j * 2^j$$

$\Rightarrow$: Es werden eine $n \times n$-Matrix von Und-Gattern für die 1-Bit-Multiplikation $A_i * B_j$ sowie Summierer für die Teilsummen benötigt.

Für große n kann eine mehrstufige Summation von Teilprodukten erfolgen. Genaueres: siehe Hayes [Hay79].

3.2.3 Ganze Zahlen

<u>Addition in VHDL</u>

In VHDL sind die Arithmetik-Operatoren wie `+`, `-`, $<$ usw. für Bitvektoren nicht vordefiniert. Diese Operatoren können aber selbst definiert werden. Ein Beispiel ist eine Funktion, die zwei Bitvektoren beliebiger (aber gleicher) Länge nach der für Volladdierer bekannten Formel addiert:

```
function "+" (a,b : bit_vector) return bit_vector is
 variable sum   : bit_vector (a'left   downto 0);
 variable carry : bit_vector (a'left+1 downto 0);
 begin
  carry(0) := '0';
  for i in 0 to a'left loop
  sum(i)     := ((a(i) XOR b(i)) XOR carry (i)) ; - - XOR nur in VHDL'92
  carry(i+1) := (a(i) AND (b(i) OR carry(i))) OR
                (b(i) AND carry(i));
  end loop;
  return sum;
 end "+";
```

Man könnte daran denken, die Addition von zwei Bitvektoren `a` und `b` alternativ zu dem oben angegebenen Verfahren durch `int(a)+int(b)` zu definieren. Dies ist wegen der begrenzten Größe von Integern in üblichen VHDL-Systemen nicht empfehlenswert.

Unter Ausnutzung dieses Konzeptes sind inzwischen umfangreiche Bibliothekspakete erstellt worden, in denen diese Operatoren auch für Bitvektoren und Mischungen von Bitvektoren und Zahlenkonstanten definiert sind. Wir werden annehmen, dass alle arithmetischen Operationen durch Import eines geeigneten Paketes auch für Bitvektoren zur Verfügung stehen.

3.2.3.1 Addition

Die bisherigen Addierer wurden für natürliche Zahlen entworfen. Im Buch von Hayes [Hay79] ist gezeigt, dass dieselben Addierer auch für ganze Zahlen geeignet sind, wobei aber der Übertrag aus der signifikantesten Stelle heraus nicht mehr die Bedeutung eines Überlaufs hat.

3.2.3.2 Subtraktion

Für die Subtraktion kann ähnlich wie für die Addition dieselbe Schaltung wie für natürliche Zahlen benutzt werden.

3.2.3.3 Multiplikation

Ganze Zahlen können analog zu dem Verfahren für natürliche Zahlen multipliziert werden, indem ggf. vor und nach der Multiplikation das Zweierkomplement gebildet wird. Diese (zeitraubende) Sonderbehandlung des Vorzeichens kann beim **Booth–Algorithmus** vermieden werden. Die Grundidee besteht darin, dass für eine Kette von Einsen nur 2 Additionen erforderlich sind. In dem folgenden Beispiel sei i die Position der am weitesten rechts und j die Position der am weitesten links stehenden 1:

$$A * \texttt{int}(\text{"0001111000"})$$

Ein Bitstring, der ab der Position j rechts nur noch Einsen enthielte, hätte den Wert $2^{j+1} - 1$. Durch die Nullen am rechten Rand wird der Wert um $2^i - 1$ kleiner. Es gilt also:

$$A * \texttt{int}(\text{"0001111000"}) = A * (2^{j+1} - 1 - 2^i + 1) = A * (2^{j+1} - 2^i)$$

Auf der stellengerechten Addition bzw. Subtraktion von A basiert der folgende `Booth`-Algorithmus:

```
FUNCTION Booth(A,B: IN bit_vector) RETURN bit_vector IS
  CONSTANT n : natural := A'LENGTH; -- Vor.: A'LENGTH = B'LENGTH
  VARIABLE P : bit_vector(n-1 DOWNTO 0) := (OTHERS => '0');
  VARIABLE Q : bit_vector(n DOWNTO 0)   := (OTHERS => '0');
BEGIN
 Q(n DOWNTO 1) := B;
 FOR i IN 0 TO n-1 LOOP
   CASE Q(1 DOWNTO 0) IS
     WHEN "10" => P := P - A;
     WHEN "01" => P := P + A;
     WHEN OTHERS =>          -- keine Aktion
   END CASE;
   -- arithmetisches Schieben (1 Bit nach rechts)
   P & Q := sra (P & Q);
 END LOOP;
 RETURN P(n-2 DOWNTO 0) & Q(n DOWNTO 1);
END Booth;
```

Beispiel:

Sei:

$$A = int(\text{"0011"}) = 3; B = int(\text{"0110"}) = 6$$

Es gilt also:

$$-A = int(\text{"1101"}); n = 4$$

Die Multiplikation mit 6 wird hier zur Multiplikation mit 8 und zur anschliessenden Subtraktion von $A * 2$:

$$A * 6 = A * int(\text{"0110"}) = A * (2^3 - 2^1)$$

Die Einzelschritte der Prozedur `Booth` sind (sra = Schiebeoperation):

```
Operation    P     Q     Kommentar
            0000  00000
            0000  01100  Q(1 DOWNTO 0) = ''00''
sra         0000  00110  Q(1 DOWNTO 0) = ''10''
-A          1101  00110  Subtraktion 1 Bit vom späteren LSB entfernt = -2* A
sra         1110  10011  Q(1 DOWNTO 0) = ''11''
sra         1111  01001
+A          0010  01001  ''1111''+''0011''=''0010'';3 Bits vom späteren LSB=+8*A
sra         0001  00100
                      ↑ LSB des Ergebnisses
               ↑ MSB des Ergebnisses
```

Das Ergebnis ist $int(\texttt{"0010010"}) = 18$.

Der "Trick" hinsichtlich der Behandlung des Vorzeichens von A besteht darin, dass P und Q zusammen eine Zahl speichern, die als Zweierkomplement-Zahl interpretiert wird, da beim Schieben das Vorzeichen erhalten bleibt. Wegen der Subtraktion von A muss ohnehin mit Vorzeichen gearbeitet werden. Allerdings funktioniert der Algorithmus nicht für die kleinste negative Zahl mit der Kodierung "100...00" (Übunsgaufgabe!), die auch sonst gelegentlich einen Sonderfall darstellt.

Wir wollen nun zeigen, dass der Booth-Algorithmus tatsächlich Zahlen gemäß der Integer-Interpretation der Bitvektoren A und B multipliziert. Wir zeigen dies für den oben verwendeten Fall von *B'length=4*. Die Verallgemeinerung auf Bitvektoren beliebiger Länge wird offensichtlich sein. Seien im folgenden die Elemente von `B` mit b_0 bis b_3 bezeichnet.

Diese Wirkung eines einzelnen Schrittes des Booth-Algorithmus besteht dann in einer stellengerechten Multiplikation von A mit dem Ausdruck $(b_{i-1} - b_i)$, der die Werte 0, -1 und +1 annimmt. Insgesamt berechnet der Booth-Algorithmus also den Ausdruck

$$\begin{array}{lcl} A & * & (b_{-1} - b_0) * 2^0 + \\ A & * & (b_0 - b_1) * 2^1 + \\ A & * & (b_1 - b_2) * 2^2 + \\ A & * & (b_2 - b_3) * 2^3 \end{array}$$

mit $b_{-1} = 0$.

Mit $-b_i * 2^{i-1} + b_i * 2^i = b_i * 2^{i-1}$ folgt, dass der folgende Ausdruck berechnet wird:

$$A * (-b_3 * 2^3 + b_2 * 2^2 + b_1 * 2^1 + b_0 * 2^0) \quad = \quad A * int(B)$$

Der Ausdruck in der Klammer entspricht genau der Integer-Interpretation des Bitvektors B.

Der Trick der korrekten Behandlung von B als Integer-Zahl besteht darin, dass "vergessen" wird, bei einer '1' im Vorzeichen von B die eigentlich übliche Behandlung am linken Rand einer Folge von Einsen vorzunehmen. Der "Fehler" bewirkt gerade, dass b_3 mit negativem Vorzeichen in den o.a. Ausdruck eingeht und B damit gemäß `int` und nicht gemäß `nat` interpretiert wird.

Verbesserungen des Booth-Algorithmus:

- Einzelne Nullen bzw. Einsen sollen wie beim Standardverfahren nur eine Operation erzeugen.

  $\Rightarrow$ Bei isolierter 1 im Bit i wird $2^{i+1} - 2^i = 2^i$ addiert.

  $\Rightarrow$ Bei isolierter 0 im Bit i wird $2^i - 2^{i+1} = -2^i$ addiert.

⇒ Übergang auf die Betrachtung eines Fensters von 3 Bit und Verschiebung um jeweils 2 Bit; abhängig vom Muster im Fenster Addition von $\pm 2 * A, \pm A$ (siehe Hayes)

- Ignorieren von Folgen gleicher Ziffern

  Statt jeweils um ein Bit zu schieben, kann man auch gleich in einem Schritt bis zum nächsten Rand einer Folge von Einsen schieben. Die bereits vorgestellten Prioritätsencoder sind geeignet, den Abstand bis zur nächsten '1' effizient zu finden. Mit zwei geeignet verschalteten Prioritätsencodern können sowohl linke als auch rechte Ränder der nächsten Folge von Einsen erkannt werden. In Kombination mit dem ebenfalls vorgestellten Barrelshifter kann so ein Schieben bis zum nächsten Rand einer Folge von Einsen vorgenommen werden.

  Sofern diese Kombination von Barrelshifter und Prioritätsencoder keinen Rand mehr finden kann, sofern also `B` nach einer gewissen Anzahl von Schritten **nur** noch Nullen oder Einsen enthält, kann das Verfahren abgebrochen werden. Wichtig ist dies für die häufig vorkommende Multiplikation von kleinen Zahlen. Bemerkenswert ist, dass dies auch für (betragsmäßig) kleine negative Zahlen funktioniert. So kann die Multiplikation von 32-Bit-Zahlen z.B. in 2 oder 3 Schritten abgeschlossen sein!

3.2.4 Gleitkomma-Zahlen

Die folgenden Algorithmen zur Ausführung der Grundoperationen enthalten keine Betrachtung der Sonderfälle NaN usw..

1. Addition

   Die Addition von zwei Gleitkomma-Zahlen kann nach folgendem Verfahren erfolgen:

   (a) Angleichen des kleineren Exponenten an den größeren (Bestimmung der Shift-Zahl mit Subtraktion; Schieben der Mantisse in einem Schritt mittels (mehrstufigem) Barrelshifter).
   (b) Addition der Mantissen
   (c) Normalisierung der Mantisse und Test auf Über/Unterlauf

   Man beachte, dass während der Rechnung auch die führende '1' der Mantisse explizit dargestellt wird. Erst beim Abspeichern in 32-Bit Worte wird diese wieder entfernt.

   Beispiel:

$6 + 2$	$= 1.5 * 2^2 + 1 * 2^1$	
	`= 0 1000 0001 1100..` $+_r$ `0 1000 0000 1000..`	(32-Bit Format,
	`VZ Charakt. *Mantisse`	'hidden bit' explizit)
	`= 0 1000 0001 1100..` $+_r$ `0 1000 0001 0100..`	nach Schritt 1
	`= 0 1000 0001` $_1$ `0000..`	mit $_1$: Übertrag
	`= 0 1000 0010 1000..`	nach Normalisierung
	$= 1.0 * 2^{(130-127)} = 1.0 * 2^3 = 8$	

2. Multiplikation

   Die Multiplikation kann aufgrund der Potenzgesetze wie folgt realisiert werden:

   (a) Addition der Exponenten (bei Verwendung der Charakteristik ist hier die bei der Erzeugung der Charakteristik benutzte Konstante abzuziehen.
   (b) Multiplikation der Mantissen
   (c) Normalisierung der Mantisse und Test auf Über/Unterlauf

3. Division

   Für die Division gilt aufgrund der Potenzgesetze folgendes Verfahren:

   (a) Subtraktion der Exponenten (bei Verwendung der Charakteristik ist hier $|min(Exp. - Bereich)|$ zu addieren).
   (b) Division der Mantissen
   (c) Normalisierung der Mantisse und Test auf Über/Unterlauf

3.3 Mikroprogrammierung

Wir wollen im folgenden im Detail kennenlernen, wir man einen Befehlssatz auf einer Hardware-Struktur realisieren kann. Wir versuchen dabei zunächst, mit möglichst wenigen Hardware-Blöcken auszukommen. Abb. 3.17 zeigt eine Hardware-Struktur, die so entworfen ist, dass sie den MIPS-Befehlssatz ausführen kann[1].

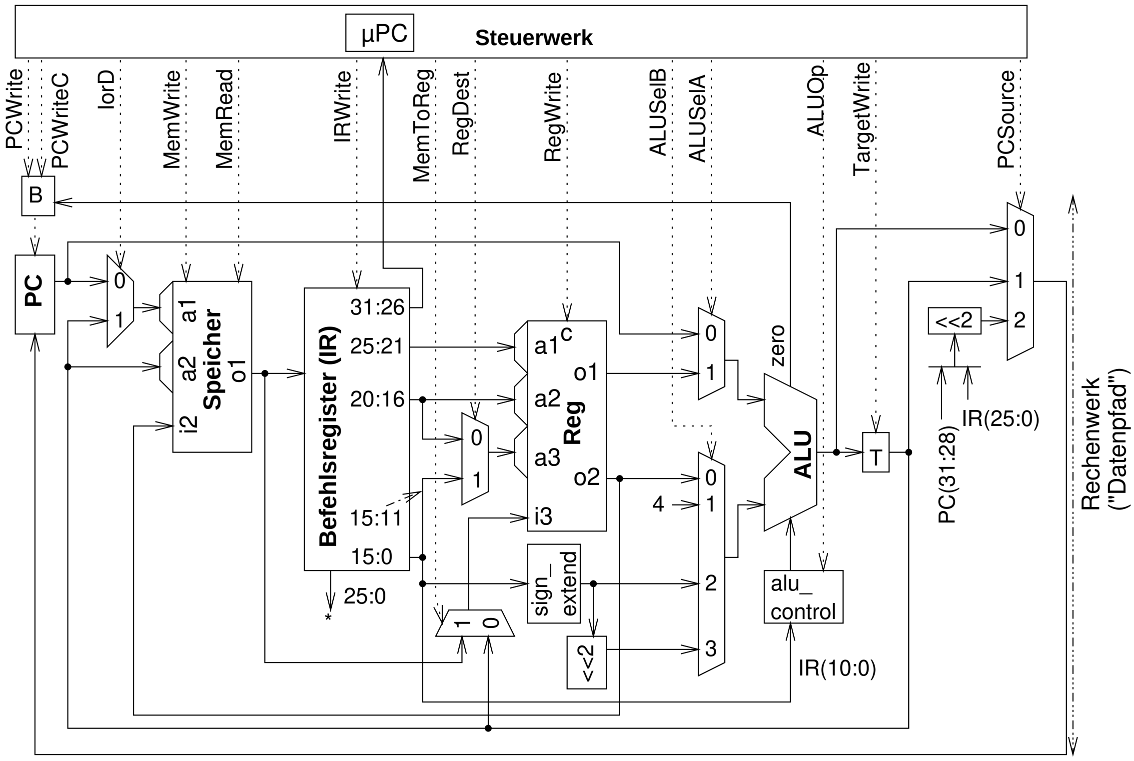


Abbildung 3.17: Hardware-Struktur zur Ausführung des MIPS-Befehlssatzes

Der Registerspeicher Reg hier als Speicher mit zwei Lese- und einem Schreibport ausgeführt. Zum Ausgangsport o1 gehört der Adresseneingang a1, zum Ausgangport o2 der Adresseneingang a2. a3 ist der Adresseneingang für den Schreibport. T ist ein Hilfsregister. Bedingte Sprünge werten das Signal zero der ALU aus. B ist ein einfache Gatterschaltung, welche das Schreiben in den Programmzähler PC steuert; die Schreibbedingung ist (PCWrite $\vee$ (PCWriteC $\wedge$ zero)). alu_control bestimmt die Funktion, die in der Recheneinheit **ALU** berechnet wird. Durch geeignete Kodierung auf der Steuerleitung ALUOp kann entweder die Addition gewählt oder die Auswahl der Funktion von den Bits (10:0) des aktuellen Befehls abhängig gemacht werden. Dies ist sinnvoll, da alle Register/Register-Befehle den Operationscode 0 haben und die ausgeführte Funktion durch die Bits (10:0) bestimmt wird. Der Speicher Speicher enthält sowohl die Befehle wie auch die Daten.

Die Beschränkung auf das Minimum an Hardware-Blöcken macht es notwendig, jeden Befehl in mehreren Schritten abzuarbeiten. Die dabei durchlaufenen Schritte kann man am Besten durch einen Automaten definieren, dessen Zustandsgraphen Abb. 3.18 beschreibt.

In der hier vorgestellten Form wird aus Gründen der Übersichtlichkeit allerdings nur ein Teil der Maschinenbefehle tatsächlich dekodiert.

Der Automat wird in der Hardware als Steuerwerk (engl. *controller*) realisiert. Die Eingaben in diesen (Moore-) Automaten werden durch die Bits (31:26), d.h. durch die Operationscodes des aktuellen Befehls, gebildet. Die gepunkteten Leitungen dienen der Übertragung der Ausgaben dieses Automaten an das Rechenwerk (den unteren Teil der Hardware-Struktur). Das Zustandsregister des Steuerwerks wollen wir μPC nennen. Für jeden Ausführungsschritt werden im Folgenden die Ausgaben und die Folgezustände dieses Automaten angegeben.

[1] Die Struktur basiert auf den Angaben bei Hennessy und Patterson ([HP95], Kapitel 5). Im Unterschied zu diesen Autoren nehmen wir hier aber nicht bereits Annahmen über die später betrachtete Realisierung mit einem Fließband vorweg.

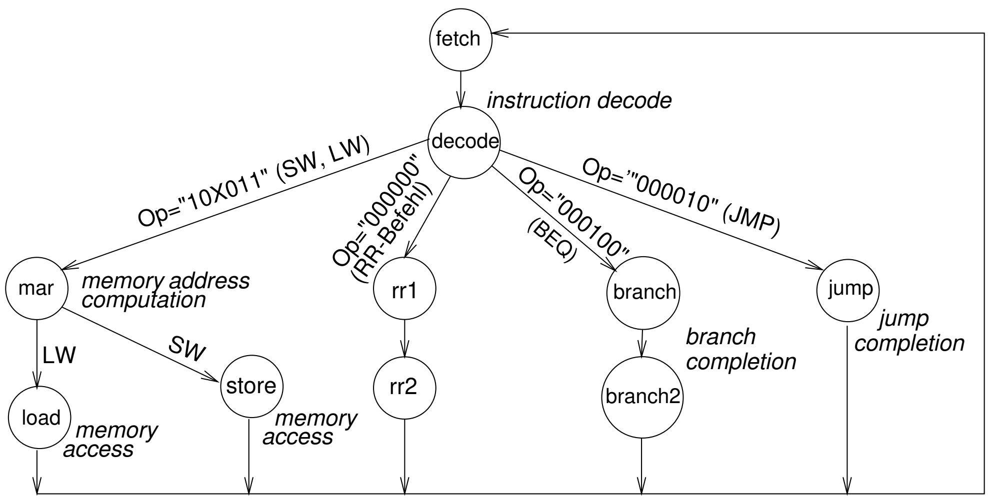


Abbildung 3.18: Zustandsgraph der Ausführung von MIPS-Befehlen

Im Zustand fetch wird zunächst einmal der nächste Befehl geholt. Gleichzeitig wird der Programmzähler um 4 erhöht (da Befehle immer 4 Bytes umfassen). Die in dieser Phase benutzten Datenwege sind in der Abb. 3.19 gestrichelt eingezeichnet.

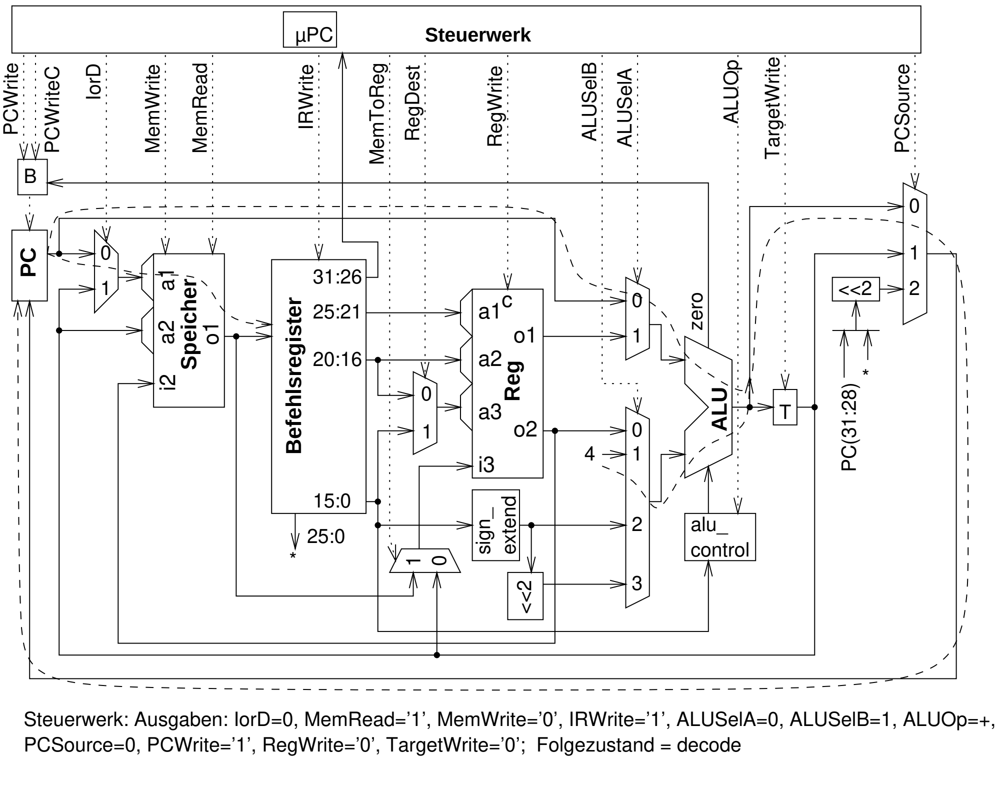


Abbildung 3.19: Benutzung der Struktur in der Befehlshol-Phase

Gleichzeitig sind auch die notwendigen Ausgaben des Steuerwerks und sein nächster Zustand (decode) angegeben. Wir nehmen an, dass mit einem Steuerkode von '0' für alle Register und Speicher das Schreiben unterdrückt und mit einem Steuercode von '1' durchgeführt wird. Für die Multiplexer geben wird die Nummer des auszuwählenden Eingangs an. Die Steuerleitungen müssen den binär codierten Wert dieser Nummer führen. Für ALUOp geben wir nur die ausgewählte Operation an; die entsprechende binäre Kodierung lassen wir offen. Alle Register, die in einem Schritt nicht beschrieben werden sollen, müssen in diesem Schritt den Steuerkode '0' erhalten. Für alle übrigen Bausteine ist der Steuerkode in Schritten, in denen sie nicht benötigt werden, redundant.

Im nächsten Schritt wird der aktuelle Befehl dekodiert (siehe Abb. 3.20).

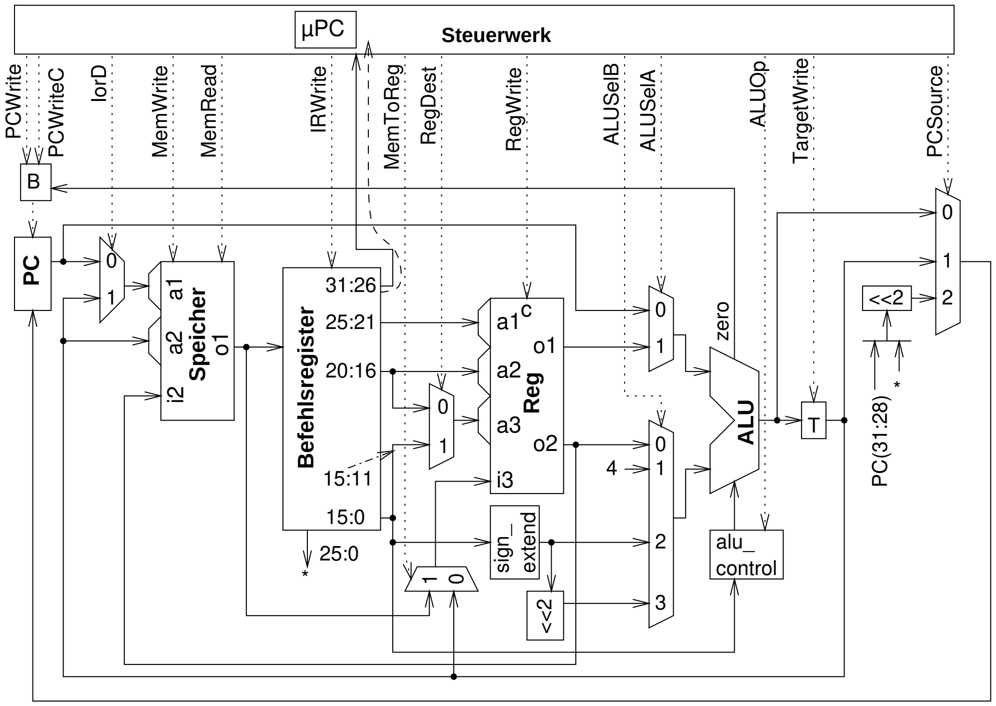


Abbildung 3.20: Benutzung der Struktur in der Dekodier-Phase

Zu diesem Zweck wird das Zustandsregister des Steuerwerks mit einem Wert geladen, der den Typ es auszuführenden Befehls eindeutig charakterisiert (ein beliebter Trick ist hier, den Operationscode des Befehls in das Zustandsregister zu laden und gültige Operationscodes als Kodierung von Zuständen des Controllers ansonsten zu vermeiden).

In diesem Schritt ist die ALU unbenutzt. Man könnte sie benutzen, um vorsorglich z.B. schon Adressen auszurechnen. Wir wollen dies unterlassen, da dies hier wenig einsichtig wäre.

Im Falle von Speicherbefehlen wird zunächst die effektive Adresse ausgerechnet. Diesem Zweck dient der Zustand mar. Die in diesem Zustand benutzten Datenwege zeigt die Abb. 3.21.

Die effektive Adresse wird um Register T abgelegt. Man beachte, dass alle Additionen von Konstanten vorzeichenerweitert erfolgen.

Im Falle eines *load*-Befehls folgt auf die Adressrechnung das Laden vom Speicher. Abb. 3.22 zeigt die benutzten Datenwege.

Im Falle eines *store*-Befehls folgt auf die Adressrechnung das Schreiben des Speichers. Abb. 3.23 zeigt die benutzten Datenwege.

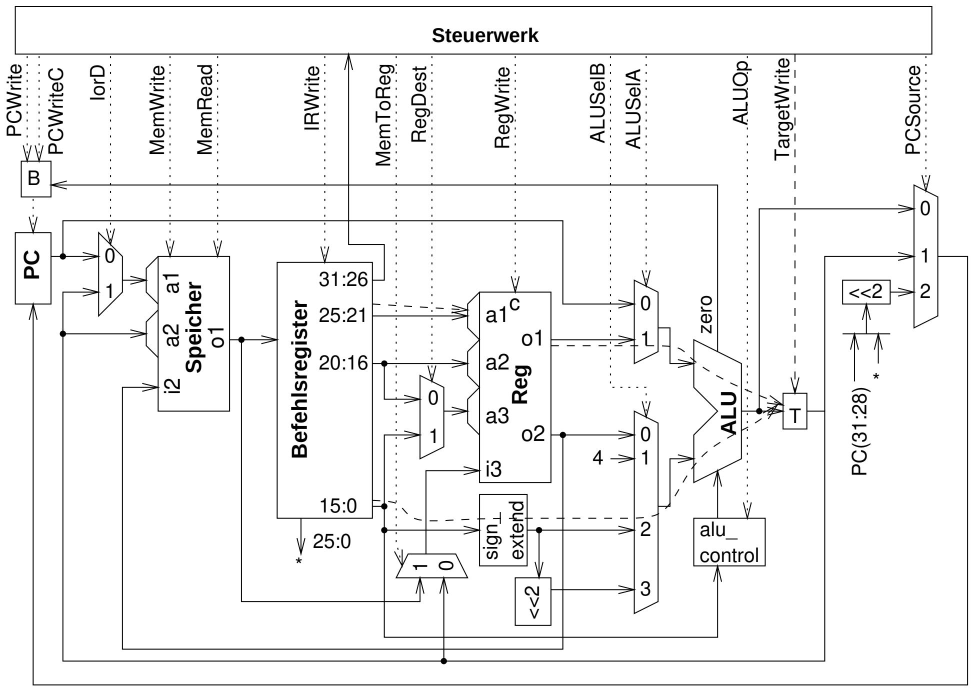
Bei Register/Register-Befehlen wird zunächst das Ergebnis ausgerechnet. Man könnte das Ergebnis unmittelbar in den Registersatz schreiben. Allerdings erfordert dies tatsächlich die Möglichkeit, auf drei voneinander unabhängige Adressen des Registerspeichers zuzugreifen. Dies ist nicht notwendig, wenn wir zunächst das Ergebnis im Hilfs-Register T halten. Diesen Vorgang zeigt Abb. 3.24.

Die Funktion der ALU ergibt sich bei Register/Register-Befehlen direkt aus den letzten 11 Bits des Befehls.

Das Kopieren aus dem Hilfsregister in den Registerspeicher zeigt dann Abb. 3.25.

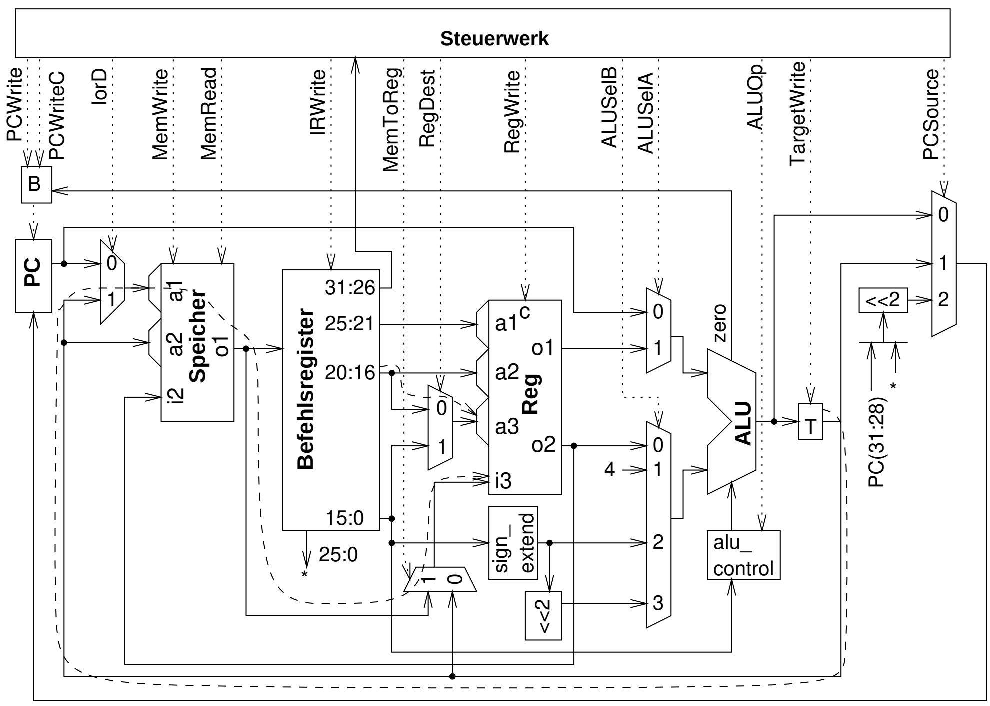
Wir könnten in dieser Phase eine der Leseadressen des Registerspeichers umschalten und so mit einem einfacheren Zweiport-Speicher auskommen.

Die Datenwege bei der Benutzung unbedingter Sprünge zeigt die Abb. 3.26.



Steuerwerk: Ausgaben: ALUSelA=1, ALUSelB=2, ALUOp=+, TargetWrite='1', andere Register: xxWrite='0'
Folgezustand = if Befehlsregister(31:26)='LW' then load else store,

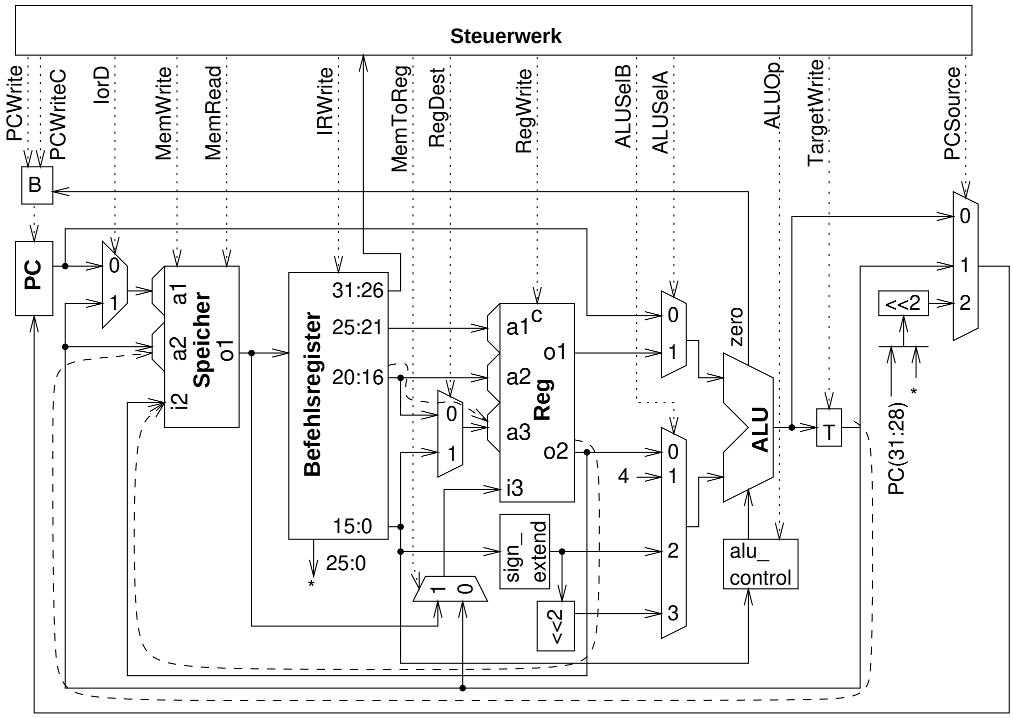
Abbildung 3.21: Benutzung der Struktur in der Adressrechnungs-Phase



Steuerwerk Ausgaben: IorD=1,MemRead='1', MemToReg=1, RegWrite='1', für andere Register: xxWrite='0';
Folgezustand = fetch

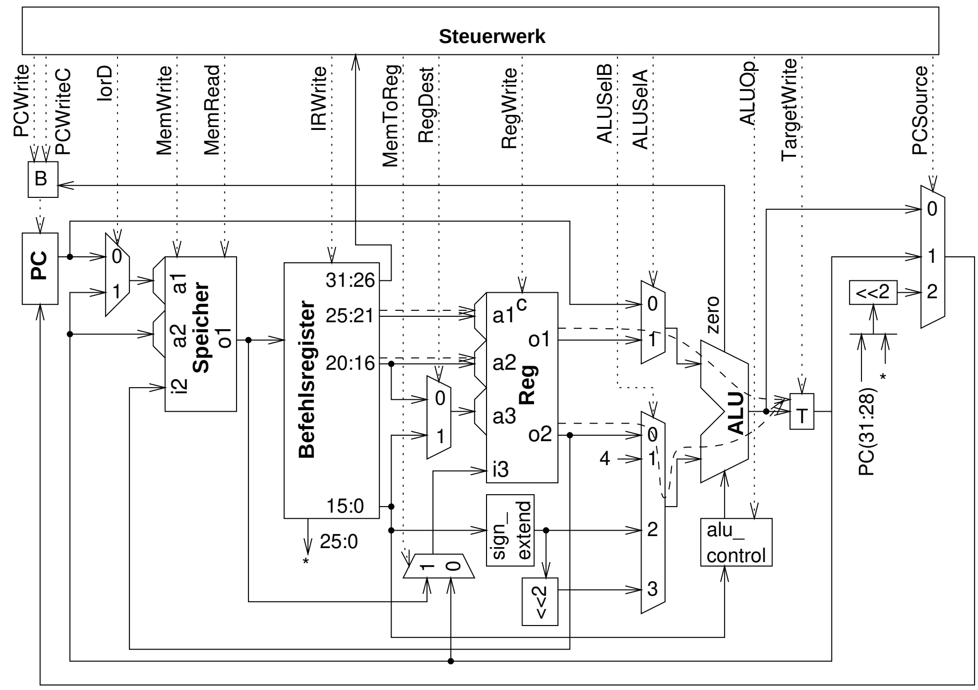
Abbildung 3.22: Benutzung der Struktur beim load-Befehl

Befehle beginnen stets an Wortgrenzen, die beiden hinteren Adressbits sind also stets = "00". Zur Vergrößerung der Sprungbereichs werden Sprungadressen daher stets um zwei Stellen nach links geschoben und mit Nullen aufgefüllt. Mit diesem Trick wird der Adressbereich von 2^{26} Bytes auf 2^{28} = 256 MB vergrößert. Die signifikantesten 4 Bits der Folgeadresse ergeben sich aus der aktuellen Programmadresse. Somit kann mit



Steuerwerk: Ausgaben: MemWrite='1', für alle anderen Register: xxWrite='0'; Folgezustand = fetch

Abbildung 3.23: Benutzung der Struktur beim store-Befehl



Steuerwerk: Ausgabe: ALUSelA=1, ALUSelB=0, ALUOp=Befehlsregister(10:0), TargetWrite='1', sonst: xxWrite='0'
Folgezustand = rr2

Abbildung 3.24: Benutzung der Struktur im Zustand rr1

jump-Befehlen nur innerhalb eines 256 MB-großen Bereichs gesprungen werden. Die entsprechende Hardware findet sich vor dem Dateneingang 2 des PC-Eingangsmultiplexers.

Als letzte Befehlsgruppe betrachten wir den BEQ-Befehl. Für diesen wird in einem ersten Zustand zunächst einmal das Verzweigungsziel ausgerechnet (Abb. 3.27).

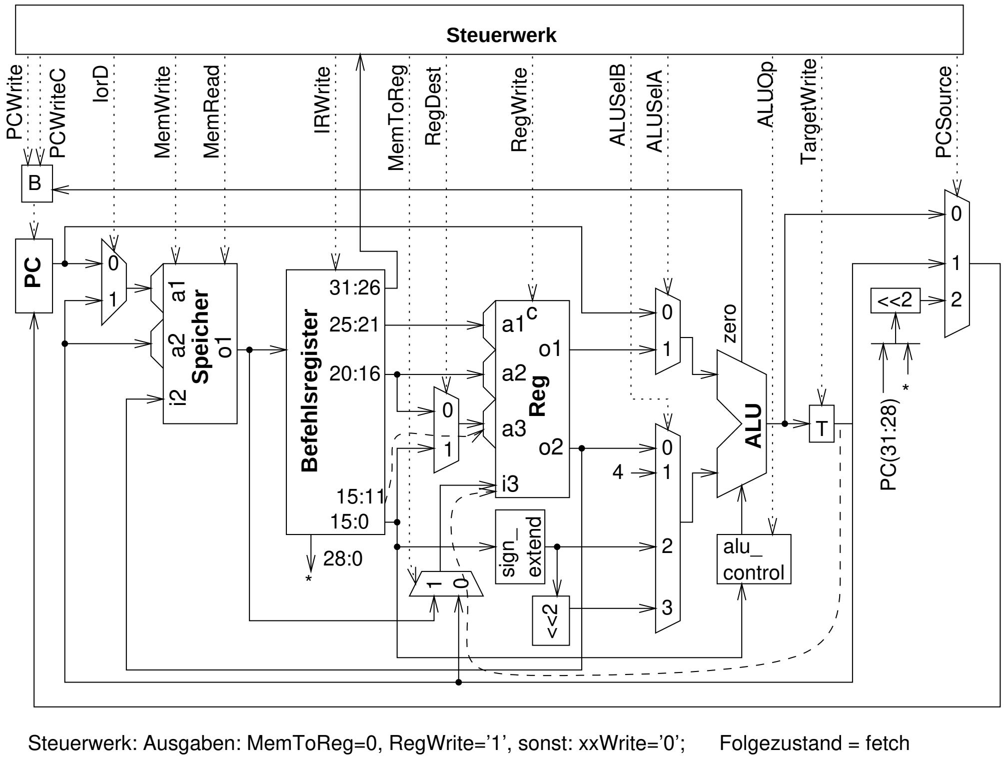


Abbildung 3.25: Benutzung der Struktur im Zustand rr2

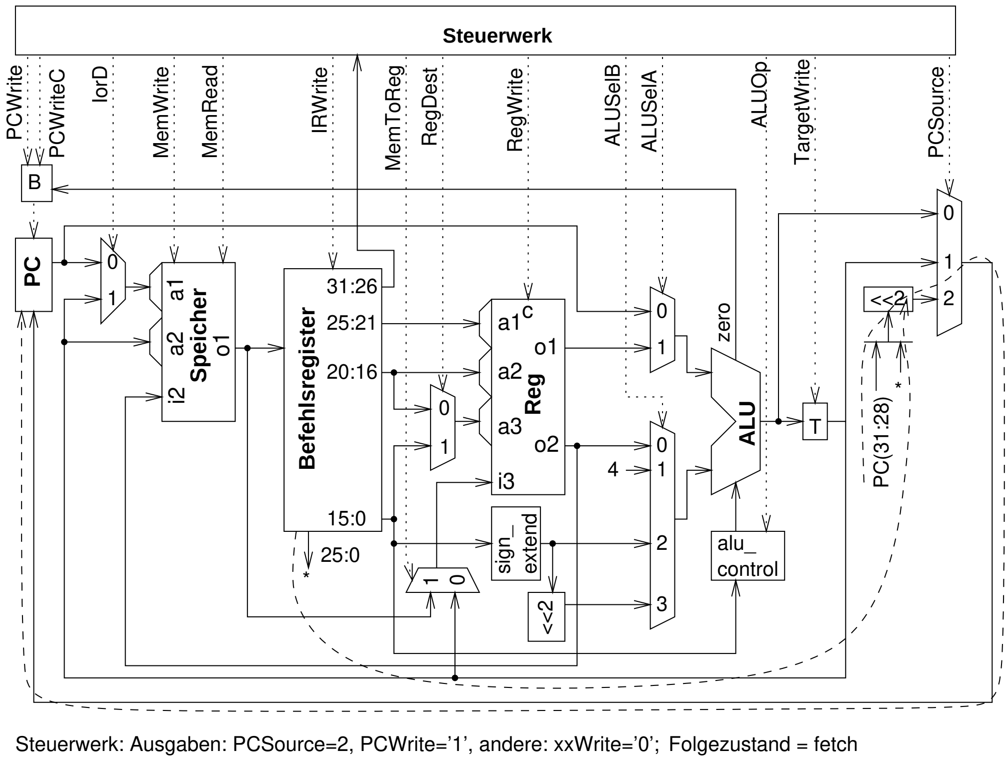
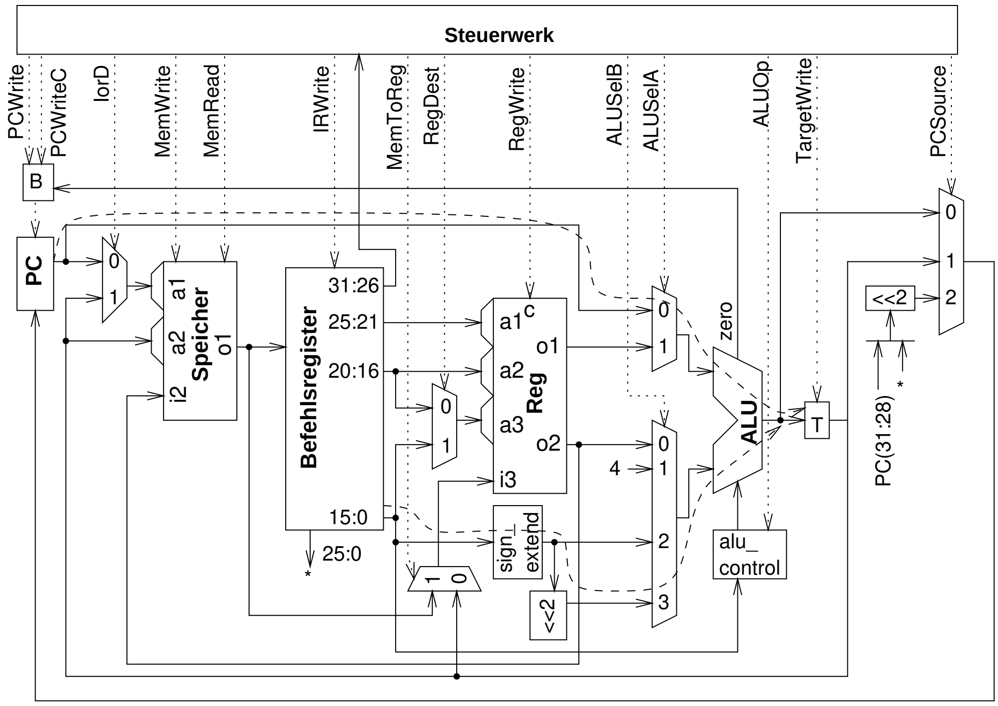


Abbildung 3.26: Benutzung der Struktur beim Sprüngen

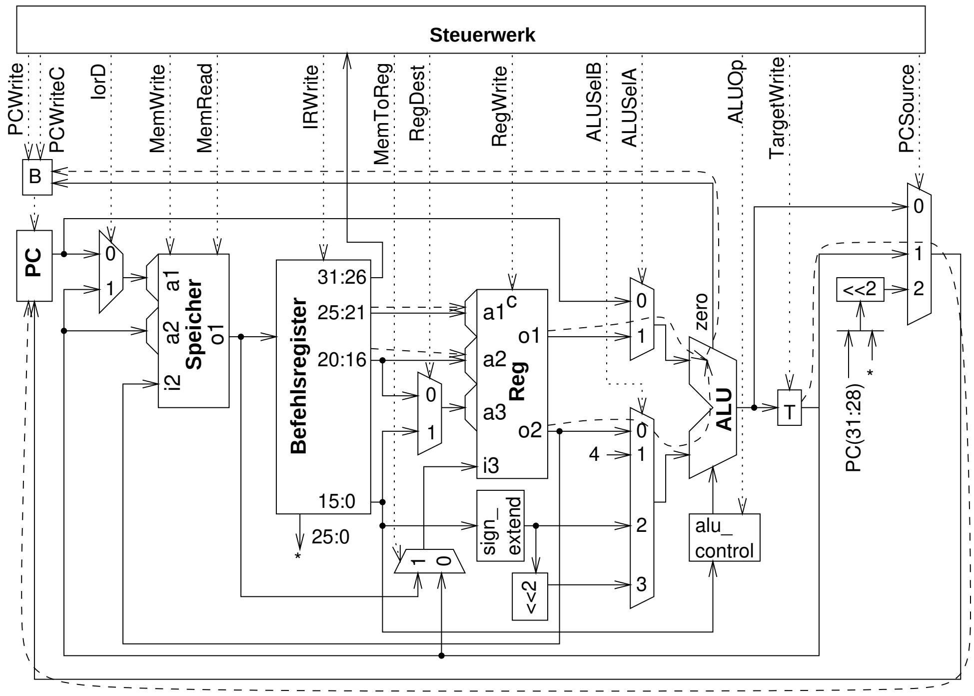
Das Verzweigungsziel ergibt sich aus der Addition einer vorzeichenerweiterten 16-Bit-Konstanten im Befehl zum aktuellen Wert des Programmzählers. Auch hier wird die Verschiebung um 2 Bit nach links genutzt.

In einem zweiten Zustand wird dann die Sprungbedingung berechnet und abhängig von dessen Ergebnis der Programmzähler geladen (siehe Abb 3.28).



Steuerwerk: Ausgabe: ALUSelA=0,ALUSelB=3, ALUOp=+, TargetWrite='1'; Folgezustand = branch2

Abbildung 3.27: Benutzung der Struktur bei Berechnungen des Verzweigungsziels



Steuerwerk: Ausgabe: ALUSelA=1, ALUSelB=0, ALUOp= = (oder = -) , PCSource=1, PCWrite='0', PCWriteC='1', xxWrite='0'; Folgezustand = fetch

Abbildung 3.28: Benutzung der Struktur bei Verzweigungen

Der Befehl BNE wurde hier nicht betrachtet. Für ihn wäre die Bedeutung der Leitung zero zu komplementieren.

Die so beschriebene Hardware realisiert allerdings nicht ganz den echten MIPS-Befehlssatz: Laut Spezifikation wird beim MIPS-Befehlssatz stets der auf den Sprung statisch folgende Befehl noch ausgeführt (*delayed* *branch*). Dieses Verhalten ergibt sich durch das Fließband der echten MIPS-Maschine. Ohne Fließband ist dieses Verhalten mühsam zu realisieren.

Aus den für die einzelnen Zustände angegebenen Ausgaben und Folgezuständen des Controllers kann jetzt die vollständige Funktion des Controllers bestimmt werden (siehe Tabelle 3.1).

Zustand	Folgezustand	PCWrite	PCWriteC	IorD	MemWrite	MemRead	IRWrite	MemToReg	RegDest	RegWrite	ALUSelB	ALUSelA	ALUOp	TargetWrite	PCSource
fetch	decode	'1'	'0'	'0'	'0'	'1'	'1'	'X'	'X'	'0'	"01"	'0'	+	'0'	"00"
decode	f(Opcode)	'0'	'0'	'X'	'0'	'0'	'0'	'X'	'X'	'0'	"XX"	'X'	X	'0'	"XX"
mar	load, store	'0'	'0'	'X'	'0'	'0'	'0'	'X'	'X'	'0'	"10"	'1'	+	'1'	"XX"
load	fetch	'0'	'0'	'1'	'0'	'1'	'0'	'1'	'0'	'1'	"XX"	'X'	X	'0'	"XX"
store	fetch	'0'	'0'	'X'	'1'	'0'	'0'	'X'	'X'	'0'	"XX"	'X'	X	'0'	"XX"
rr1	rr2	'0'	'0'	'X'	'0'	'0'	'0'	'X'	'X'	'0'	"00"	'1'	IR	'1'	"XX"
rr2	fetch	'0'	'0'	'X'	'0'	'0'	'0'	'0'	'1'	'1'	"XX"	'X'	X	'0'	"XX"
branch	branch2	'0'	'0'	'X'	'0'	'0'	'0'	'X'	'X'	'0'	"11"	'0'	+	'1'	"XX"
branch2	fetch	'0'	'1'	'X'	'0'	'0'	'0'	'X'	'X'	'0'	"00"	'1'	=	'0'	"01"
jump	fetch	'1'	'0'	'X'	'0'	'0'	'0'	'X'	'X'	'0'	"XX"	'X'	X	'0'	"10"

Tabelle 3.1: Automatentabelle des Steuerwerks

Wenn aus dem aktuellen Zustand (in μPC) mittels eines Speichers der Folgezustand und die Ausgabe bestimmt werden, dann nennen wir diesen Speicher **Mikroprogrammspeicher** und dessen Inhalt **Mikroprogramm**. Mikroprogrammspeicher können als reine Lesespeicher (ROM) oder wiederbeschreibbar (als RAM) ausgeführt werden.

Das Beispiel zeigt, wie Befehlssätze mit Mikrobefehlen realisiert werden können. Erweiterungen hinsichtlich des Abtestens von Interrupts usw. sind leicht zu ergänzen. Vorteile der Mikroprogrammierung sind vor allen Dingen die Flexibilität in Bezug auf späte Änderungen sowie die Möglichkeit, rechte komplexe Operationen ausführen zu können, ohne einen neuen Maschinenbefehl vom Hauptspeicher laden zu müssen. Komplexe Operationen, wie z.B. die Berechnung von Quadratwurzeln oder die Unifikation in PROLOG können kompakt in einem Maschinenbefehl kodiert und in vielen Mikroschritten mit entsprechend häufigen Zugriffen auf den Mikroprogramm-Speicher ausgeführt werden. Mikroprogrammierung ist damit besonders dann sinnvoll, wenn sich die Geschwindigkeit des Hauptspeichers und des Mikroprogramm-Speichers stark unterscheiden. Dies war bei Einführung der Mikroprogrammierung der Fall (große, langsame Ringkern-Speicher und kleine, schnelle Halbleiterspeicher).

Ein Nachteil der Mikroprogrammierung ist der zusätzliche Overhead. Die beiden ersten Schritte der Ausführung (Interpretation) von Maschinenprogrammen führen keine nützlichen Operationen auf den Daten aus. Würde man Anwendungsprogramme direkt in Mikrocode compilieren, so fielen diese beiden Schritte weg. Eine direkte Übersetzung auf die **unterste programmierbare Ebene** würde also Zeit sparen, vorausgesetzt, man könnte die Befehle schnell genug lesen.

3.4 Fließbandverarbeitung

3.4.1 Grundsätzliche Arbeitsweise

Bei der vorgestellten Realisierung des Maschinenbefehlssatzes per Mikroprogramm werden für jeden Maschinenbefehl mehrere Takte benötigt. Bei RISC-Rechnern möchte man die Anzahl der Takte (den CPI-Wert) aber zumindest in die Nähe von 1 bringen. Der wesentliche Trick ist die Fließbandverarbeitung.

Werden in einer Architektur mehrere Befehle analog zu einem Fertigungs-Fließband gleichzeitig bearbeitet, so sprechen wir von einer **Fließband-Architektur** (engl. *pipeline architecture*).

Als Beispiel werden wir die bislang benutzte Architektur so modifizieren (siehe Abb. 3.29), dass überwiegend mit jedem Takt die Bearbeitung eines Befehls beendet wird.

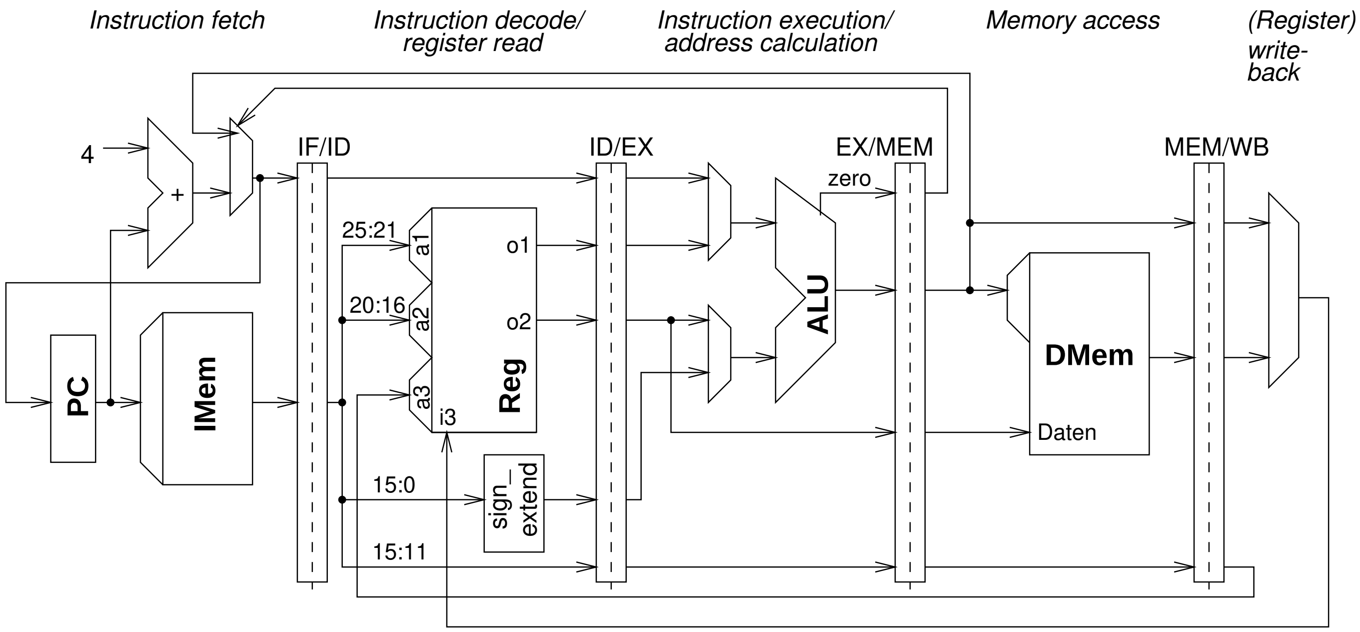


Abbildung 3.29: Fließband-Stufen

Die wesentlichen Veränderungen sind die folgenden :

- Für die Berechnung der Programm-Folgeadressen wird ein separater Addierer vorgesehen, damit sich diese und Rechnungen auf Daten nicht behindern. Die Multiplexer werden dadurch wieder einfacher.
- Der Speicher wird konzeptuell auf einen Daten- und einen Befehlsspeicher aufgeteilt, damit sich Zugriffe auf Befehle und Daten ebenfalls nicht behindern. Später werden getrennte Daten- und Programmcaches die Funktion getrennter Speicher realisieren.
- Das Rechenwerk wird in eine Reihe von Fließbandstufen aufgeteilt, die untereinander durch Pufferregister (fett gezeichnet) getrennt werden. Das bislang schon benutzte Register T wird ein Pufferregister. Ebenso wird das Befehlsregister eines dieser Pufferregister
- Wir verzichten auf einige Details der Generierung von Kontrollsignalen und der Verzweigungslogik.

Die einzelnen Stufen dieser Architektur realisieren jeweils eine Befehls-Verarbeitungsphase. Konkret betrachten wir in diesem Beispiel die folgenden fünf Phasen:

1. die Befehlshol-Phase
2. die Dekodier- und Register-Lese-Phase
3. die Ausführungs- und Adressberechnungs-Phase

4. die Speicherzugriffs-Phase
5. die Abspeicherungssphase (engl. *result writeback phase*)

Die Architektur ist wieder (unter Berücksichtigung von noch notwendigen Verfeinerungen) im Wesentlichen in der Lage, den MIPS-Maschinenbefehlssatz zu bearbeiten.

Im einzelnen übernehmen die Phasen bzw. die Stufen die folgenden Aufgaben:

1. die Befehlshol-Phase:

   Lesen des aktuellen Befehls aus dem Befehlsspeicher IMem. Der Einfachheit halber nehmen wir vorläufig an, der Befehlsspeicher sei von dem Datenspeicher unabhängig. Wir werden später Techniken kennenlernen, durch Verwendung von Pufferspeichern de facto getrennte Speicher auch dann zu erreichen, wenn physikalisch ein und derselbe Hauptspeicher sowohl Daten als auch Befehle aufnimmt.

   Def.: Architekturen, bei denen für die Verarbeitung von Daten und von Befehlen separate Speicher und Busse vorhanden sind, bezeichnet man als **Harvard-Architekturen**.

   Die meisten der RISC-Architekturen sind heute Harvard-Architekturen.

2. die Dekodier- und Register-Lese-Phase

   In dieser Phase wird der auszuführende Befehl dekodiert. Da die Registernummern feste Plätze im Befehlsformat haben, können diese Nummern parallel zur Dekodierung schon zum Lesen der Register benutzt werden.

3. die Ausführungs- und Adressberechnungs-Phase

   In dieser Phase wird bei arithmetischen Befehlen die arithmetische Funktion berechnet. Bei Lade-, Speicher- und Sprungbefehlen wird die effektive Adresse berechnet.

4. die Speicherzugriffs-Phase

   Bei Lade- und Speicherbefehlen wird in dieser Phase der Speicherzugriff ausgeführt. Bei den übrigen Befehlen wird diese Phase im Prinzip nicht benötigt, der einfacheren Struktur des Fließbands wegen aber beibehalten.

5. die Abspeicherungssphase (engl. *result writeback phase*)

   In dieser Phase erfolgt das Abspeichern im Registersatz, und zwar sowohl bei Lade- als auch bei Arithmetik-Befehlen. Der Eingang des Registersatzes, obwohl hier in der Dekodierstufe eingezeichnet, wird regelmäßig erst in der *writeback*-Phase eines Befehls benutzt.

Im Prinzip wäre es möglich, dieselben Hardware-Einheiten wie z.B. Addierer durch verschiedene Stufen/Phasen gemeinsam nutzen zu lassen. Beispielsweise könnte man evtl. einen getrennten Inkrementierer für PC-Werte durch Nutzung der ALU sparen. Auch könnte man dafür sorgen, dass Lade- und Speicherbefehle die *writeback*-Phase nicht erst durchlaufen. Des einfacheren Übergangs auf eine funktionierende Fließbandverarbeitung wegen werden wir diese Möglichkeiten nicht ausnutzen.

Den idealen Durchlauf von Befehlen durch das Fließband zeigt die Tabelle 3.30.

Zyklus	Befehl holen	Dekodieren	Ausführen	Speicherzugriff	Abspeichern
1	Befehl 1	-	-	-	-
2	Befehl 2	Befehl 1	-	-	-
3	Befehl 3	Befehl 2	Befehl 1	-	-
4	Befehl 4	Befehl 3	Befehl 2	Befehl 1	-
5	Befehl 5	Befehl 4	Befehl 3	Befehl 2	Befehl 1
6	Befehl 6	Befehl 5	Befehl 4	Befehl 3	Befehl 2

Abbildung 3.30: Arbeitsweise eines idealen Fließbands

Aus einer Reihe von Gründen kann die Arbeit des Fließbands beeinträchtigt werden, da es möglicherweise gegenseitige Abhängigkeiten zwischen Befehlen gibt, die gleichzeitig im Fließband bearbeitet werden oder bearbeitet werden sollen.

3.4.2 Strukturelle Abhängigkeiten

Die ideale Arbeitsweise des Fließbands setzt voraus, dass soviele Hardwareressourcen zur Verfügung gestellt werden, dass die gleichzeitige Verarbeitung in den verschiedenen Stufen nie an fehlenden Hardwareressourcen scheitert.

In einigen Fällen kann ein derartiger Hardwareaufwand nicht effizient realisiert werden. Beispielsweise kann es sein, dass der Datenzugriff von Register-Ladebefehlen nicht gleichzeitig mit dem Holen eines Befehls ausgeführt werden kann, da sonst der Hauptspeicher unter zwei unabhängigen Adressen gleichzeitig lesbar sein müsste, was nur teuer zu realisieren ist. Caches reduzieren die Häufigkeit derartiger gleichzeitiger Zugriffe; ganz auszuschließen sind sie aber nicht. Ein anderes, häufiges Beispiel stellen Gleitkommaeinheiten dar. Sie sind meist intern nicht mit so vielen Fließbandstufen aufgebaut, wie es erforderlich wäre, um mit jedem Takt eine neue Gleitkomma-Operation starten zu können.

In solchen Fällen gehen evtl. Zyklen durch die Sequentialisierung des Zugriffs auf Ressourcen verloren. Man nennt eine Situation, in der aufgrund von strukturellen Abhängigkeiten zwischen Befehlen die maximale Arbeitsgeschwindigkeit des Fließbands beeinträchtigt werden könnte, eine **strukturelle Gefährdung** bzw. auf Englisch *structural hazard*.

3.4.3 Datenabhängigkeiten

Sofern ein Befehl i Daten bereitstellt, die von einem folgenden Befehl j (mit $j > i$) benötigt werden, sprechen wir von einer **Datenabhängigkeit** [Mal78]. Datenabhängigkeiten sind zweistellige **Relationen** über der Menge der Befehle. Hennessy und Patterson bezeichnen diese Art der Abhängigkeit, in der j Daten liest, die ein vorangehender Befehl erzeugt, als RAW (*read after write*)-Abhängigkeit.

Als Beispiel betrachten wir die folgende Befehlssequenz:

```
ADD $1,$2,$3
SUB $4,$5,$1
AND $6,$1,$7
OR  $8,$1,$9
XOR $10,$1,$11
```

Die Befehle `SUB, AND, OR` und `XOR` sind wegen der in `$1` bereitgestellten Daten vom `ADD`-Befehl datenabhängig.

Neben der Datenabhängigkeit gibt es noch die **Antidatenabhängigkeit** (engl. *antidependence* oder *antidependency*). Ein Befehl i heisst antidatenabhängig von einem nachfolgenden Befehl j, falls j eine Speicherzelle beschreibt, deren ungeänderter Inhalt von i noch gelesen werden müsste. Hennessy und Patterson bezeichnen diese Art der Abhängigkeit als WAR (*write after read*)-Abhängigkeit (Schreiben muss nach dem Lesen erfolgen). Wenn der `OR`-Befehl in der obigen Sequenz das Register `R1` beschreiben würde, gäbe es eine Antidatenabhängigkeit zwischen ihm und den `SUB`- und `AND`-Befehlen.

Schließlich gibt es noch die Ausgabeabhängigkeit (engl. *output dependency*). Zwei Befehle i und j heißen ausgabeabhängig, falls beide dieselbe Speicherzelle beschreiben. Hennessy und Patterson bezeichnen diese Art der Abhängigkeit als WAW (*write after write*)-Abhängigkeit. Nach der eben erwähnten Änderung wären beispielsweise der `ADD`- und der `OR`-Befehl voneinander ausgabeabhängig.

Antidatenabhängigkeit und Ausgabeabhängigkeit entstehen durch das Wiederverwenden von Namen für Speicherzellen. Ohne diese Wiederverwendung (und in Sprachen, die Wiederverwendung vom Konzept her ausschließen) kämen sie nicht vor. Daraus folgt auch, dass man sie im Prinzip durch Umbennung vermeiden kann. Für Register sind derartige Techniken als *register renaming* bekannt.

Den Ablauf der oben beschriebenen Befehle im Fließband und die Wirkung der Datenabhängigkeit zeigt die Abb. 3.31.

Sofern keine besonderen Vorkehrungen getroffen werden, wird der `SUB`-Befehl in `$1` nicht das Ergebnis des `ADD`-Befehls finden, da letzterer erst im Zyklus 5 abspeichert. Auch der `AND`-Befehl würde in Zyklus 4 einen

Zyklus	Befehl holen	Dekodieren	Ausführen	Speicherzugriff	Abspeichern
1	ADD \$1,\$2,\$3	-	-	-	-
2	SUB \$4,\$5,\$1	ADD \$1,\$2,\$3	-	-	-
3	AND \$6,\$1,\$7	SUB \$4,\$5,\$1	ADD \$1,\$2,\$3	-	-
4	OR \$8,\$1,\$9	AND \$6,\$1,\$7	SUB \$4,\$5,\$1	ADD \$1,\$2,\$3	-
5	XOR \$10,\$1,\$11	OR \$8,\$1,\$9	AND \$6,\$1,\$7	SUB \$4,\$5,\$1	ADD \$1,\$2,\$3
6	...	XOR \$10,\$1,\$11	OR \$8,\$1,\$9	AND \$6,\$1,\$7	SUB \$4,\$5,\$1

Abbildung 3.31: Datenabhängigkeiten

alten Inhalt (engl. *stale data*) von `$1` lesen. Man nennt die "Gefährdung" des einwandfreien Fließbandverhaltens durch Datenabhängigkeiten wie die für `$1` *data hazard*.

In den folgenden Beispielen wollen wir wie Hennessy und Patterson annehmen, dass das Schreiben von `DMem` und von `Reg` in der Mitte eines Zyklus erfolgt. Dies kann man z.B. dadurch realisieren, dass die Pipeline-Register mit der positiven Flanke des Taktes und die beiden erwähnten Speicher mit der negativen Flanke desselben Taktes ihre Daten übernehmen.

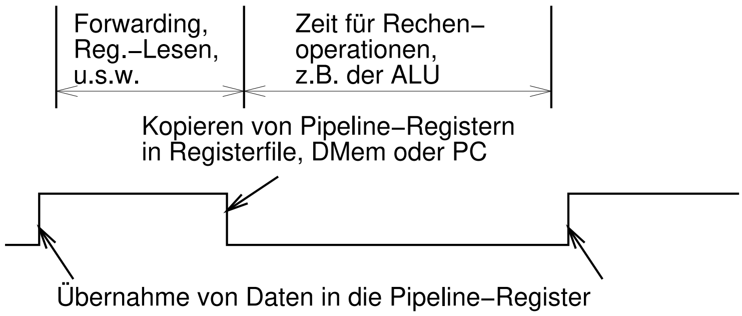


Abbildung 3.32: Taktung des Fließbands

Für die korrekte Arbeitsweise der ersten drei Stufen reicht es aus, wenn die in den Pipeline-Registern zu speichernden Informationen vor dem Ende des Zyklus bekannt sind.

Wir nehmen weiter an, dass `Reg` so ausgeführt ist, dass geschriebene Daten durch *bypassing* innerhalb des Speichers auch sofort wieder zur Verfügung stehen, wenn Schreib- und Leseadresse identisch sind. Unter dieser Annahme wird der `OR`-Befehl korrekt ausgeführt, da der Inhalt des Registers `R1` in Zyklus 5 zunächst abgespeichert und danach in das ID/EX-Fließbandregister übernommen wird.

Bypässe, *forwarding*

Um für den `SUB`- und den `AND`-Befehl den richtigen Wert zu verwenden, können wir ausnutzen, dass dieser am Ende des Zyklus 3 auch schon bekannt ist. Wir müssen nur dafür sorgen, dass die Datenabhängigkeit durch die Hardware erkannt wird (durch eine Buchführung über die jeweils benötigten Register) und zu einem Kurzschließen der ansonsten für diesen Wert noch zu durchlaufenden Datenwege führt (engl. *forwarding, bypassing* oder *short-circuiting*).

Das Beispiel zeigt bereits, dass Bypässe von verschiedenen Fließbandstufen zu den ALU-Eingängen führen müssen. Weiter ist klar, dass für beide ALU-Eingänge separate Multiplexer zur Auswahl zwischen den verschiedenen Bypässen bestehen müssen (siehe Abb. 3.33, nach Hennessy und Patterson, Abb. 3.20).

Für die Ansteuerung der Multiplexer muss jeweils die betroffene Registernummer bekannt sein. Zu beachten ist, dass bei einem Interrupt zwischen den datenabhängigen Befehlen kein Bypass benötigt wird, da das Fließband vor der Bearbeitung des Interrupts geleert wird (engl. *pipeline flushing*).

Das nächste Beispiel eines Programmsegments zeigt, dass Bypässe nicht im jedem Fall eine korrekte Lösung bei Datenabhängigkeiten darstellen:

```
LW  $1,0($2)   -- load word
SUB $4,$1,$5
```

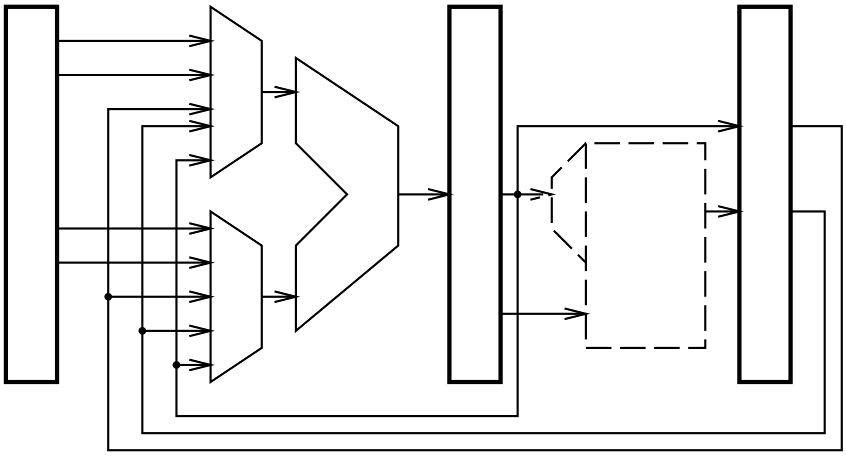


Abbildung 3.33: Bypass-Logik an den Eingängen der ALU

```
AND $6,$1,$7
OR  $8,$1,$9
```

Abb. 3.34 zeigt den zeitlichen Ablauf. `R1` ist erst im Zyklus 4 bekannt und kann damit keine Operanden mehr für die `SUB`-Operation liefern, bei dem oben angenommenen Timing aber per Bypass noch für die `AND`- und die `OR`-Operation.

Zyklus	Befehl holen	Dekodieren	Ausführen	Speicherzugriff	Abspeichern
1	LW \$1,0(\$2)	...	...	...	...
2	SUB \$4,\$1,\$5	LW \$1,0(\$2)	...	...	...
3	AND \$6,\$1,\$7	SUB \$4,\$1,\$5	LW \$1,0(\$2)	...	...
4	OR \$8,\$1,\$9	AND \$6,\$1,\$7	SUB \$4,\$1,\$5	LW \$1,0(\$2)	...
5	...	OR \$8,\$1,\$9	AND \$6,\$1,\$7	SUB \$4,\$1,\$5	LW \$1,0(\$2)

Abbildung 3.34: Unzulänglichkeit der Bypass-Lösung

Eine mögliche Lösung besteht im Anhalten der Pipeline (engl. *pipeline stall*, auch *hardware interlocking* genannt). Bis zum korrekten Laden des Registers führen die übrigen Stufen des Fließbands *no operation* Befehle aus. Man nennt diese Befehle auch *bubbles* [HP96] im Fließband. Die Einträge *idle* in der Abb. 3.35 sollen andeuten, dass die betreffenden Stufen in den Zyklen 5 bzw. 6 unbeschäftigt sind. Der `SUB`-Befehl erhält den korrekten Inhalt von `$1` im Zyklus 5 über *forwarding*.

Zyklus	Befehl holen	Dekodieren	Ausführen	Speicherzugriff	Abspeichern
1	LW \$1,0(\$2)	...	...	...	...
2	SUB \$4,\$1,\$5	LW \$1,0(\$2)	...	...	...
3	AND \$6,\$1,\$7	SUB \$4,\$1,\$5	LW \$1,0(\$2)	...	...
4	NOOP	NOOP	NOOP	LW \$1,0(\$2)	...
5	OR \$8,\$1,\$9	AND \$6,\$1,\$7	SUB \$4,\$1,\$5	NOOP	LW \$1,0(\$2)
6	...	OR \$8,\$1,\$9	AND \$6,\$1,\$7	SUB \$4,\$1,\$5	NOOP
7	...	...	OR \$8,\$1,\$9	AND \$6,\$1,\$7	SUB \$4,\$1,\$5

Abbildung 3.35: Anhalten des Fließbands

Da das Anhalten des Fließbands zu einem Leistungsverlust führt, versuchen moderne Compiler, den Code so zu umzusortieren, dass es selten zu einem solchen Anhalten kommt. Man spricht in diesem Zusammenhang vom *instruction scheduling*.

3.4.4 Kontrollfluss-Abhängigkeiten

Besondere Schwierigkeiten ergeben sich durch bedingte Sprünge. Als Beispiel betrachten wir das Programm:

```
     BEQ  $1,$2 t --branch to t if equal
     SUB  ...
     ....
t:   ADD
```

Abb. 3.36 zeigt das zeitliche Verhalten für dieses Programm im schlimmsten Fall. Dies ist der Fall, in dem bis zur Berechnung der Sprungbedingung der nächste Befehl noch nicht gestartet wird. Es zeigt sich, dass in diesem Fall drei Zyklen durch den bedingten Sprung verloren gehen würden. Man spricht in diesem Zusammenhang von einem *branch delay* von zwei Zyklen.

Zyklus	Befehl holen	Dekodieren	Ausführen	Speicherzugriff	Abspeichern
1	BEQ	...	...	...	...
2	SUB	BEQ	...	...	...
3	NOOP	NOOP	BEQ	...	...
4	SUB oder ADD	NOOP	NOOP	BEQ	...
5	...	SUB oder ADD	NOOP	NOOP	BEQ
6	...	...	SUB oder ADD	NOOP	NOOP
7	...	...	...	SUB oder ADD	NOOP
8	...	...	...	...	SUB oder ADD

Abbildung 3.36: Bedingter Sprung, schlimmster Fall

Eine Beschleunigung der Implementierung ist möglich, wenn sowohl die möglichen Sprungadressen als auch die Sprungbedingung in einer früheren Stufe des Fließbands berechnet werden. `BEQ` (*branch if equal*) und `BNE` (*branch if not equal*) sind die einzigen bedingten Sprungbefehle im MIPS-Befehlssatz. Diese einfachen Bedingungen können bei geringem Hardware-Aufwand schon in einer früheren Stufe der Pipeline berechnet werden. Allerdings wird ein separater Addierer zur Berechnung der Folgeadresse bei relativen Sprüngen benötigt, da die ALU in der Regel wegen der Ausführung eines anderen Befehles nicht zur Verfügung steht. Abb. 3.37 zeigt die zusätzliche Hardware, die wir in der Dekodierstufe benötigen, um sowohl den Test auf Gleichheit wie auch das Sprungziel schon in dieser Stufe zu berechnen.

Das geänderte Fließband realisiert Sprünge mit einem *branch delay* von einem Zyklus, in dem im Falle der MIPS-Spezifikation ein sinnvoller Befehl ausgeführt werden kann (siehe Abb. 3.38).

Der Befehl `SUB` wird also immer ausgeführt (getreu dem Motto *it's not a bug, it's a feature*). Die einfachste Möglichkeit, *branch delay slots* korrekt mit Befehlen zu füllen, besteht in dem Einfügen von `NOOP`s. Weniger Zyklen gehen verloren, wenn der Compiler die *slots* mit Befehlen füllt, die sinnvolle Berechnungen ausführen. Dafür kommen v.a. Befehle in Frage, die sonst **vor** vor dem Sprungbefehl im Speicher stehen würden. Bei anderen Befehlen muss sorgfältig darauf geachtet werden, dass die Semantik des Programms erhalten bleibt. Verfahren, welche solche anderen Befehle ausfindig machen, heißen globale **Scheduling-Verfahren** (engl. *global scheduling*). Eines der ersten solcher Verfahren ist das *trace scheduling* von Fisher [Fis81]. Verbesserungen davon sind das *percolation scheduling* und das *mutation scheduling*.

Der Nachteil von *delayed branches* ist es, dass eine Eigenheit der internen Architektur in der externen Architektur sichtbar gemacht wird. Dies ist ein mögliches Problem, da sich die interne Architektur im Laufe der Zeit ändern wird.

Die beschriebene Reduktion der verlorenen Zyklen ist bei Maschinen mit komplexen Sprungbefehlen schwieriger zu realisieren.

Im Falle anderer Spezifikationen muss nach dem bedingten Sprungbefehl noch für einen Zyklus angehalten werden (siehe Abb. 3.39).

Nach der beschriebenen Möglichkeit, bedingte Sprünge mittels Anhaltens des Fließbands zu realisieren, erwähnen wir im Folgenden weitere Möglichkeiten, Zyklen einzusparen.

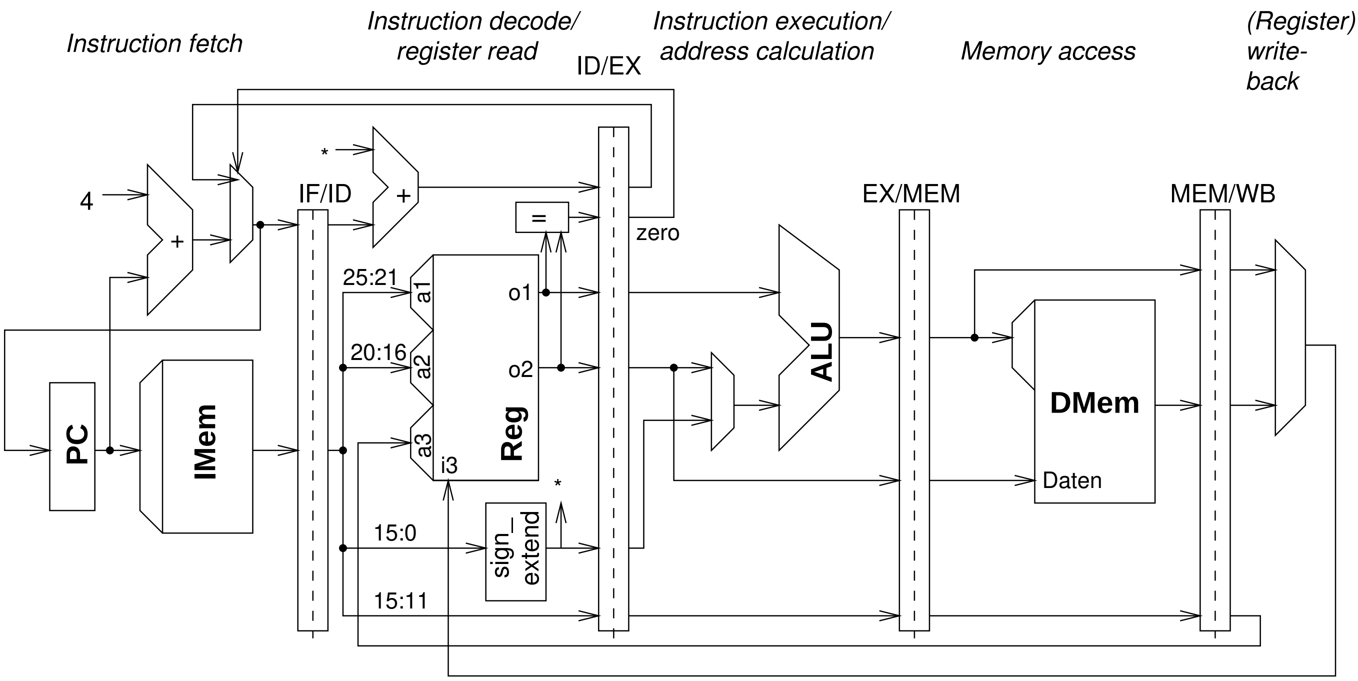


Abbildung 3.37: Fließband-Stufen

Zyklus	Befehl holen	Dekodieren	Ausführen	Speicherzugriff	Abspeichern
1	BEQ	...	...	...	...
2	SUB	BEQ	...	...	...
3	ADD	SUB	BEQ	...	...
4	...	ADD	SUB	BEQ	...
5	...	...	ADD	SUB	BEQ
6	...	...	...	ADD	SUB
7	...	...	...	...	ADD

Abbildung 3.38: Delayed Branch

Zyklus	Befehl holen	Dekodieren	Ausführen	Speicherzugriff	Abspeichern
1	BEQ	...	...	...	...
2	SUB	BEQ	...	...	...
3	SUB oder ADD	NOOP	BEQ	...	...
4	...	SUB oder ADD	NOOP	BEQ	...
5	...	...	SUB oder ADD	NOOP	BEQ
6	...	...	...	SUB oder ADD	NOOP
7	...	...	...	...	SUB oder ADD

Abbildung 3.39: Bedingter Sprung, ein Wartezyklus

Bei der ersten Möglichkeit fährt man zunächst mit der Bearbeitung fort, so als gäbe es keinen Sprung-Befehl. Man tut also so, als würde man vorhersagen, dass die Sprungbedingung nicht erfüllt ist (engl. *predicting the branch as not taken*). Stellt sich dann allerdings später heraus, dass tatsächlich gesprungen werden soll, so muss sichergestellt sein, dass durch die begonnene Bearbeitung der auf den Sprung folgenden Befehle keine dauerhafte Veränderung des Prozessorzustands eintritt.

Bei der zweiten Möglichkeit nimmt man an, dass der Sprung ausgeführt wird. Dies kann sinnvoll sein, wenn viele der Sprünge Rücksprünge in Schleifen sind. Ferner ist es von Vorteil, wenn das Sprungziel schnell, die Sprungbedingung aber nur langsam berechnet werden kann. Dies ist beim MIPS-Befehlssatz allerdings nicht der Fall.

3.4.5 Behandlung von Interrupts

Bei RISC-Maschinen, die mit starkem Pipelining arbeiten, kann aus dem Auftreten eines durch eine Befehlsausführung verursachten Interrupts nicht unmittelbar auf den verursachenden Befehl geschlossen werden, weil z.B. der Programmzähler schon mehrfach erhöht worden ist, bevor die Ausnahmebehandlung erforderlich wird. Dadurch wird die eigentliche Fehler- bzw. Ausnahmebehandlung schwierig.

Def.: Sind in einer Architektur Vorkehrungen getroffen, die im Falle eines Interrupts stets die verursachende Operation erkennen lassen, so spricht man von **präzisen Interrupts** (engl. *precise interrupts*).

Eine Möglichkeit besteht darin, auch den Inhalt des Befehlszählers von Fließbandstufe zu Fließbandstufe weiterzugeben. Dann ist zu jeder möglichen Ausnahme in einer Stufe auch der zugehörige Befehlszähler bekannt. Das Abbrechen der in Bearbeitung befindlichen Befehle und ggf. der Neustart von Berechnungen nach Bedienung des Interrupts bleiben aber in jedem Fall kompliziert. Man versucht daher, präzise Interrupts nur in den unbedingt notwendigen Fällen zu realisieren. Hierzu zählen auf jeden Fall Seitenfehler-Interrupts (siehe Kap. 4), aber nicht in jedem Fall arithmetische Überläufe.

3.4.6 Mehrzyklen-Operationen

Bislang haben wir angenommen, dass alle Fließbandstufen ihre Arbeit in einem Zyklus beenden können. Tatsächlich ist dies für viele Stufen aber nicht der Fall.

In diesem Zusammenhang müssen wir zwei Begriffe einführen

Def.: Unter der **Latenzzeit** (engl. *latency*) verstehen wir den Abstand (in Taktzyklen) zwischen dem Start einer Operation in einer Fließbandstufe und der Verfügbarkeit des Ergebnisses.

Def.: Unter dem *initiation interval* verstehen wir den Abstand (in Taktzyklen) zwischen zwei aufeinanderfolgenden Starts von Operationen einer Fließbandstufe.

Ein Problem von Mehrzyklen-Operationen ist die Behandlung von Interrupts. Als Beispiel betrachten wir die folgenden Sequenz von *single precision*-Gleitkomma-Befehlen:

```
div.s $F0,$F2,$F4   ; F0 := F2/F4
add.s $F10,$F10,$F8
sub.s $F12,$F12,$F14
```

Da es keine Datenabhängigkeiten gibt, können diese Befehle in verschiedenen Hardware-Einheiten gleichzeitig bearbeitet werden. Dabei wollen wir annehmen, dass der Divisionsbefehl langsamer ist als die übrigen. Dann werden die Befehle nicht mehr in der Reihenfolge beendet, in der sie oben angeführt sind. Man nennt dies *out-of-order completion*. Was passiert, wenn jetzt der `sub.s` einen Überlauf zur Folge hat? Wenn dieser Befehl dem Benutzer als Ursache eines Fehlers angezeigt wird, kann dieser sich durch die nicht abgeschlossene Ausführung der Division sehr verblüffen lassen.

Zur Lösung gibt es zwei Ansätze:

- Man sieht für das Debugging spezielle Maschinenbefehle vor, die die gleichzeitige Bearbeitung mehrerer arithmetischer Befehle blockieren. Diese Befehle reduzieren die Leistung des Prozessors erheblich. Sie können aber sehr hilfreich sein. Die Lösung wird in den Prozessoren DEC Alpha 21064 und 21164, IBM PowerPC-1, IBM PowerPC-2 und MIPS R8000 benutzt.
- Man sieht spezielle Puffer vor, in denen alle Informationen gehalten werden, bis die vorangehenden Befehle ihre Bearbeitung abgeschlossen haben und erzeugt erst danach den Interrupt. In diesem Fall können allerdings sehr viele Puffer erforderlich werden.

Manche Hardwareeinheiten benötigen zur Bearbeitung einer Operation mehrere Takte (besonders bei Gleitkommabefehlen). Ein CPI-Wert in der Nähe von 1 kann damit nur erreicht werden, wenn die Parallelarbeit von mehreren Hardwareeinheiten möglichst gut genutzt wird. Im Folgenden werden wir Techniken vorstellen, die es erlauben, die Parallelarbeit möglichst gut zu nutzen. Diese Techniken erlauben es auch, CPI-Werte kleiner als 1 zu erreichen (siehe Abschnitt 3.4.9).

3.4.7 Dynamisches Scheduling

3.4.7.1 Einführung

Eine der Techniken zur Realisierung von Parallelarbeit ist das dynamische Scheduling. Das dynamische Scheduling überwindet eine Einschränkung der bisherigen Techniken der Beschleunigung, nämlich die Tatsache, dass Befehle in der Reihenfolge gestartet werden, in der sie im Programmcode enthalten sind. Im folgenden werden wir Möglichkeiten betrachten, die Reihenfolge der Befehlsbearbeitung erst zur Laufzeit festzulegen. Techniken dazu bezeichnet man als dynamisches Scheduling (engl. *dynamic scheduling*). Vom dynamischen Scheduling zu unterscheiden ist die statische Festlegung der Befehlsreihenfolge zur Übersetzungszeit. Statisches Scheduling hatten wir als eine Möglichkeit kennengelernt, möglichst wenige Zyklen zu vergeuden, z.B. indem wir *branch delay slots* mit nützlichen Operationen füllen.

Der Aufwand für dynamisches Scheduling ist erheblich höher, bietet aber auch einige Vorteile.

Zunächst ist statisches Scheduling immer auf ein bestimmtes Prozessormodell beschränkt. Dies macht die Codegüte modellabhängig. Vom Compilerstandpunkt aus gesehen ist es besser, guten Code für eine ganze Prozessorfamilie generieren zu können, ohne sich um die modellabhängigen Eigenschaften von Fließbändern zu kümmern. Dynamisches Scheduling kann mit statischem Scheduling kombiniert werden. So kann der Compiler bereits versuchen, datenabhängige Befehle möglichst weit auseinander zu ziehen. Verbleibende modellabhängige Ressourceabhängigkeiten könnten dann durch dynamisches Scheduling aufgelöst werden.

Als erstes Beispiel betrachte man die folgende Sequenz doppelt genauer MIPS-Gleitkommabefehle, die sich auf die speziellen Gleitkomma-Register $Fx beziehen:

```
div.d $F0,$F2,$F4   ; F0  := F2/F4
add.d $F10,$F0,$F8  ; F10 := F0+F8
sub.d $F12,$F8,$F14
```

Der `sub.d`-Befehl kann bei T*in order execution* nicht sogleich ausgeführt werden. Für *out of order execution* ist zu beachten, dass der `add.d`-Befehl vom `div.d`-Befehl abhängt, der seinerseits das Fließband belegt. Andererseits ist der `sub.d`-Befehl aber von den übrigen der angegebenen Befehle nicht abhängig und er könnte vor dem `add.d`-Befehl gestartet werden. Um eine derartige *out of order execution* zu ermöglichen, muss in der Befehlsdekodierung zwischen der Erkennung von Datenabhängigkeiten und der Erkennung von Ressourcenabhängigkeiten getrennt werden.

In einem ersten Schritt wird der Befehl dekodiert und es wird geprüft, ob Ressourcenabhängigkeiten vorliegen. Falls solche nicht vorliegen, wird der betreffende Befehl in einer Befehlsqueue eingetragen. Für Befehle in der Befehlsqueue wird überprüft, ob ihre Datenabhängkeiten mit den vorhandenen Hardware-Maßnahmen berücksichtigt werden können. Falls ja, so wird ihre Bearbeitung gestartet.

Eine Möglichkeit der Berücksichtigung von Datenabhängigkeiten besteht im *score-boarding* (score board = Trefferbrett). Scoreboarding wurde zuerst im dem Hochleistungsrechner CDC 6600 benutzt. Score-boarding muss in der Lage sein, auch Antidatenabhängigkeiten und Ausgabeabhängigkeiten zu berücksichtigen, denn beide Formen der Abhängigkeit kommen bei *out of order execution* zum Tragen, auch wenn sie bei *in order execution* noch keine Probleme bereiteten. Man betrachte dazu die folgende modifizierte Befehlssequenz:

```
div.d $F0,$F2,$F4     ; F0  := F2/F4
add.d $F10,$F0,$F8    ; F10 := F0+F8
sub.d $F8,$F8,$F14    ; F8  := F8-F14
```

Falls jetzt der `sub.d`-Befehl vor dem `add.d`-Befehl ausgeführt wird, so liest der `add.d`-Befehl aufgrund der

Antidatenabhängigkeit zwischen beiden Befehlen den falschen Wert für `$F8`. Da in der bislang besprochenen Architektur stets in der letzten Fließbandstufe abgespeichert wurde und da sich Befehle nicht überholen konnten, traten derartige Probleme bislang nicht auf.

Auch die Ausgabeabhängigkeit zwischen Befehlen kann sich jetzt auswirken. Würde man im `sub.d`-Befehl das Zielregister `$F10` verwenden, könnte bei *out of order execution* in `$F10` der falsche Wert verbleiben.

3.4.7.2 *Scoreboarding*

Die Abhängigkeiten werden im Scoreboard wie folgt berücksichtigt:

Jeder Befehl, der von der *instruction fetch*-Einheit kommt, durchläuft das Scoreboard. Dieses führt über die Abhängigkeiten aller ihm bekannten Befehle Buch. Wenn es feststellt, dass für einen Befehl alle Operanden bekannt sind und wenn auch eine Ausführungseinheit frei ist, weist es den Befehl der Ausführungseinheit zu. Wenn es Datenabhängigkeiten zwischen noch nicht abgeschlossenen und von der fetch-Einheit empfangenen Befehlen gibt, werden die empfangenen Befehle zunächst einmal zwischengespeichert. Alle Ausführungseinheiten müssen abgeschlossene Berechnungen dem Scoreboard melden. Das Scoreboard erteilt der Ausführungseinheit die Berechtigung zum Abspeichern des Ergebnisses, sofern die Speichereinheit frei ist. Danach prüft es, ob für wartende Befehle nun alle Operanden bekannt sind und startet diese, sofern dies der Fall ist. *Scoreboarding* überwindet Grenzen der Reorganisation des Codes, die sich ansonsten durch Datenabhängigkeiten ergeben würden. Trotz der Datenabhängigkeit zwischen obigem div.d und add.d-Befehlen kann die Ausführung des sub.d-Befehls vorgezogen werden und schon starten, während sich der langsame div.d-Befehl noch in der Bearbeitung befindet. Nicht überwinden kann *scoreboarding* Grenzen der Reorganisation aufgrund der Ausgabe- und Antidatenabhängigkeiten.

3.4.7.3 Verfahren von Tomasulo

Diese Beschränkung wird mit dem Algorithmus von Tomasulo aufgehoben, der für die IBM 360/91 entwickelt wurde. Ziel war es, trotz der geringen Anzahl an Gleitkommaregistern und der langsamen Gleitkommaeinheiten möglichst viel Parallelarbeit zu ermöglichen.

Für den Algorithmus von Tomasulo ist der Begriff der *reservation stations* zentral. Jede funktionelle Einheit besitzt derartige *reservation stations* als eine Art Eingabewarteschlange.

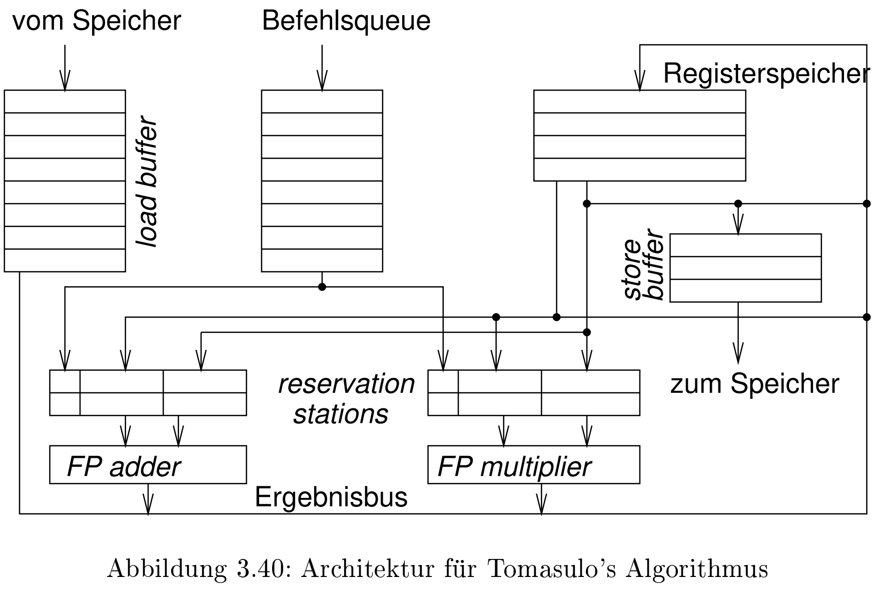


Abbildung 3.40: Architektur für Tomasulo's Algorithmus

Reservation stations enthalten auszuführende Operationen und -soweit bereits bekannt- die zugehörigen Operanden. Befehle werden gestartet, indem sie einem Befehlspuffer entnommen und einer *reservation station* zugewiesen werden. Sind alle Operanden bekannt und ist die zugeordnete funktionelle Einheit frei, so kann die Bearbeitung in der funktionellen Einheit beginnen. Am Ende der Bearbeitung wird das Ergebnis über einen gemeinsamen Bus von allen Einheiten übernommen, die das Ergebnis benötigen. Zu den Einheiten zählen insbesondere alle *reservation stations*, die auf das Ergebnis warten. Das Verteilen der Daten kann erfolgen, solbald die Daten bekannt sind. Es muss nicht erst auf die *write back*-Phase gewartet werden. Bei datenabhängigen Operationen ist es nicht notwendig, die Daten in einen Registerspeicher zu übernehmen und aus diesem wieder zu lesen. Auch das *forwarding* wird durch den Algorithmus von Tomasulo mit erledigt.

Jeder Operandenbereich einer *reservation station* enthält *tag bits*, welche den gewünschten Operanden bezeichnen. Aus den *tag bits* geht hervor, von welcher *reservation station* oder aus welchem *load buffer* der Operand kommen muss. Diese *tag bits* ersetzen die Registeradressen. Die Speicherbereiche für Operanden in den *reservation stations* realisieren damit de facto mehr Register als im Befehlssatz nach aussen sichtbar sind. Implizit wird damit ein *register renaming* durchgeführt. Das bedeutet, die Register des Befehlssatzes werden dynamisch auf eine größere Anzahl von Speicherplätzen in den *reservation stations* abgebildet. Durch diese Umbennenung können Ausgabe- und Antidatenabhängigkeiten beseitigt und so die Parallelität gegenüber dem *scoreboarding* erhöht werden.

Ein weiterer Unterschied zum *scoreboarding* ist die dezentrale Kontrolle des Algorithmus von Tomasulo.

3.4.8 Sprung-Vorhersage

Bei sehr komplexen Fließbändern können die Performance-Verluste durch bedingte Sprünge erheblich sein. Wenn vier oder fünf Befehle pro Zyklus gestartet werden können, ist bereits ein einzelner durch Sprünge verlorener Zyklus ein erheblicher Verlust. Man versucht daher, die Richtung von Sprüngen möglichst gut vorherzusehen und in dieser Richtung weiterzuarbeiten, bevor die Sprungbedingung tatsächlich bekannt ist.

Zu diesem Zweck realisiert man sogannte *branch prediction buffer*. Derartige Puffer werden mit einigen der unteren Adressbits (den sog. Index-Bits) des bedingten Sprungs indiziert. Ob die so angesprochene Zelle des Puffers tatsächlich zu dem gerade betrachteten Sprungbefehl gehört, ist nicht sicher, denn es könnte mehrere Sprünge geben, die sich nur in den oberen Adressbits (den Tag-Bits) unterscheiden. Um sicher zu sein, dass der Eintrag zum aktuellen Programmzähler-Stand gehört, muss man auch die Tag-Bits abspeichern und die aktuellen Tags mit den gespeicherten vergleichen (siehe Abb. 3.41). Solange man aber **im Mittel** fast immer den Eintrag zum aktuellen Sprungbefehl erhält, kann man auf diesen Tag-Vergleich evtl. auch verzichten.

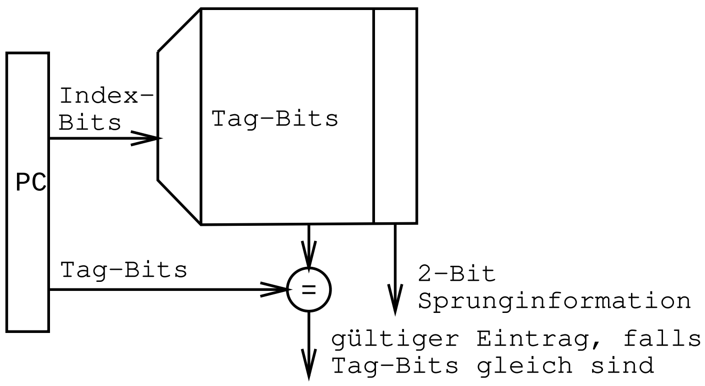


Abbildung 3.41: *Branch prediction buffer*

Der Puffer enthält in der so angesprochenen Zelle mindestens zwei Bits, welche Auskunft geben, in welche Richtung der Sprung bei seiner letzten Ausführung ging. Je eine Codierung ("0" und "11" in Abb. 3.42) besagt, dass der Sprung mindestens zweimal in dieselbe Richtung verzweigte. Die beiden übrigen Codierungen besagen, dass es eine Ausnahme von der bisherigen Sprungrichtung gab. Es wird angenommen, dass ein Sprung in eine Richtung verzweigt, solange nicht zweimal nacheinander in die andere Richtung verzweigt wurde.

Die 2-Bit Sprungvorhersage bewirkt im Falle der MIPS-Architektur keine Beschleunigung. Ein *branch delay slot* bleibt auch bei einer 2-Bit Sprungvorhersage erhalten.

Eine Beseitigung des *branch delay slots* gelingt mit der Einführung eines *branch target buffers*, der auch die Adresse des Sprungziels enthält. So braucht bei komplexen Adressberechnungen nicht erst das Ende der Adressberechnung abgewartet zu werden.

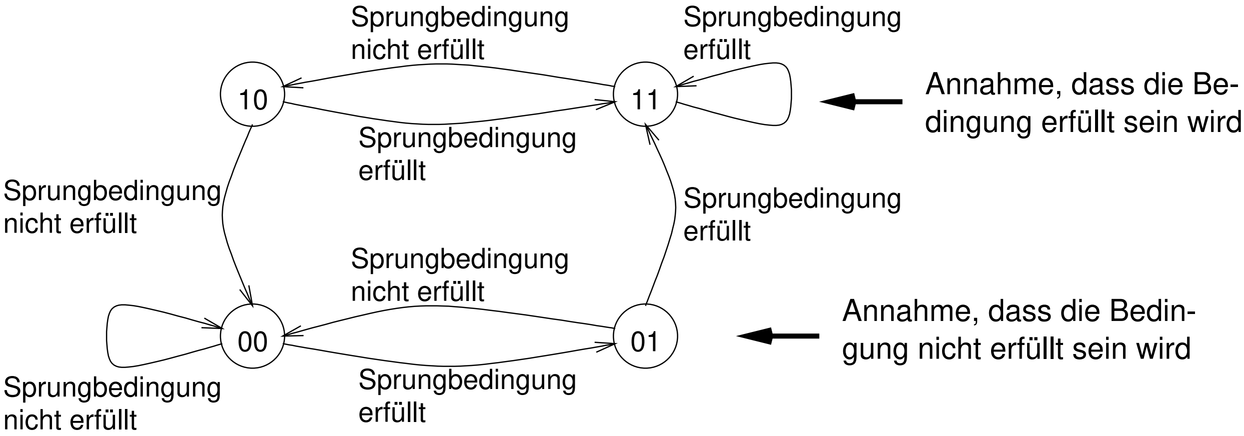


Abbildung 3.42: 2-Bit Sprungvorhersage

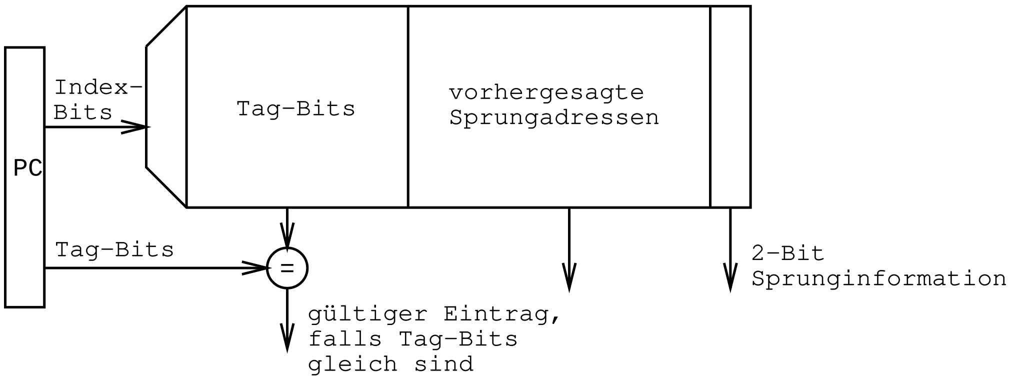


Abbildung 3.43: *Branch target buffer*

Sobald ein Eintrag im *branch target buffer* gefunden wird, werden die weiteren Befehle ab der dort angegebenen Adresse geholt. Im gleichen Puffer kann man auch die 2-Bit Sprungvorhersage-Information speichern.

Man kann *branch target buffer* und *branch prediction buffer* auch getrennt realisieren, denn beim *branch target buffer* muss zur Vermeidung eines Leistungsverlusts durch falsche Vorhersagen auf jeden Fall ein Tag-Vergleich durchgeführt werden. Die entsprechenden Einträge großer Wortbreite werden aber nur für erfüllte Bedingungen benötigt, während der *branch prediction buffer* für alle bedingten Sprünge benötigt wird. Aus diesem Grund werden *branch prediction buffer* und *branch target buffer* im PowerPC getrennt realisiert.

Durch die Sprung-Vorhersage ist es möglich, Befehle so schnell zu holen, dass das Fließband stets mit anderen Befehlen gefüllt bleibt, dass also die Sprünge selbst praktisch keine Zeit kosten.

Sofern man in einer der Verzweigungsrichtungen bereits Befehle holt und sie verarbeitet, bevor überhaupt der Sprungbefehl komplett abgearbeitet wurde, spricht man von *spekulativer Ausführung*. Bei spekulativer Ausführung muss man natürlich stets in der Lage sein, die Wirkung der spekulativ ausgeführten Befehle zu annulieren, falls man in der falschen Richtung vorgearbeitet hat.

3.4.9 *Multiple Instruction Issue*

Um die Rechenleistung weiter zu steigern, ist man vor einigen Jahren dazu übergegangen, auch Architekturen mit CPI-Werten <1 zu entwickeln. Zu diesem Zweck besitzen Maschinen mehrere funktionelle Einheiten wie z.B. Integer- und Gleitkomma-Rechenwerke. Da mehrere dieser Einheiten gleichzeitig arbeiten können, kann bei geigneter Befehlsdekodierung mehr als ein Maschinenbefehl pro Zyklus ausgeführt werden.

Zur Realisierung solcher Maschinen muss die *instruction fetch*-Stufe pro Takt mehrere Befehle zur Ausführung bereitstellen. Man spricht in diesem Zusammenhang auch von *multiple instruction issue*.

Es gibt im Wesentlichen zwei Methoden, mehrere Befehle pro Takt bereitzustellen:

- **VLIW-Architekturen**

  Bei VLIW-Prozessoren ist allein der Compiler dafür verantwortlich, mehrere Befehle zu finden, die während eines Taktes bereitgestellt werden können.

- **superskalare Architekturen**

  Superskalare Architekturen enthälten Hardwaremechanismen, welche zur Laufzeit pro Takt mehrere Befehle bereitstellen können. Bei ausreichend vielen funktionellen Einheiten benötigt ein Befehl im Mittel dann weniger als einen Takt zur Ausführung.

  Def.: Architekturen mit CPI-Werten < 1 heißen **superskalare Architekturen**.

  Beispielsweise stellen der Prozessor MIPS R8000 pro Zyklus 4, der Pentium Pro 5 Befehle pro Zyklus zur Verfügung. Es kann allerdings Einschränkungen hinsichtlich der gleichzeitigen Ausführung aller bereitgestellten Befehle geben.

  <u>Interne Struktur von Pentium-Prozessoren</u>

  Die vorgestellte Fließbandverarbeitung kann man mit überschaubarer Logik (wenn überhaupt) nur bei einfachen Befehlen und v.a. bei Befehlen fester Länge realisieren. Dies ist eine wesentliche Motivation für die Benutzung von RISC-Befehlssätzen. Aus diesem Grund konnte man Ende der 80er-Jahre eigentlich erwarten, dass CISC-Befehlssätze aussterben würden. Dennoch hat sich der x86er-Befehlssatz erstaunlich gut gehalten, besser als die meisten RISC-Befehlssätze. Der Grund liegt in dem Zwang zur Befehlssatz-Kompatibilität für PC-Applikationen und der Tatsache, dass man bei der Realisierung neuer Pentium-Architekturen die Grundideen der RISC-Architekturen aufgegriffen hat. Die Architekturen sind zwar extern weiter CISC-Architekturen. Intern werden jedoch x86-Befehle in RISC-Befehle zerlegt und diese werden in mehreren parallelen Fließbändern abgearbeitet (siehe Abb. 3.44). Dies ist zwar keine sehr effiziente Verwendung des Siliziums, aber darauf kommt es bei PC-Applikationen nicht an.

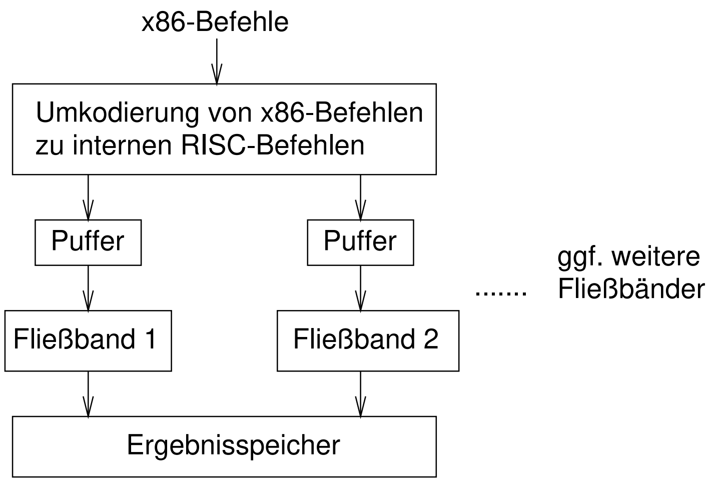


Abbildung 3.44: Interne Struktur von Pentium-Prozessoren

3.4.10 Mehrfädige Architekturen (*Multi-threading)*

Die Geschwindigkeit von Prozessoren steigt stärker als die Zugriffsgeschwindigkeit (genauer: Latenzzeit, engl. *memory latency*) auf den Speicher. Damit spielt die Latenzzeit eine immer größere Rolle. *Hiding memory latency* ist damit ein zentrales Problem moderner Rechnerarchitekturen. Die folgenden Zahlen [SRU99] zeigen die Verhältnisse für einen Alpha *Server* 4100 mit vier 300 MHz Alpha Prozessoren:

Eine Technik zum Verstecken von Latenzzeiten ist die gleichzeitige Bearbeitung mehrerer Prozesse oder Fäden (engl. *threads*). Man kann zwischen drei Techniken unterscheiden [SRU99]:

Miss	Treffer im	Latenzzeit [Prozessorzyklen]
L1-Cache	L2-Cache (Chip)	7
L2-Cache	L3-Cache (Platine)	21
L3-Cache	Lokaler Speicher	80
Lokaler Speicher	Speicher eines anderen Prozessors	125

Tabelle 3.2: Latenzzeiten bei einem Alpha-Server (1998)

- *Cycle-by-cycle interleaving*: Mit jedem Prozessorzyklus wird ein Befehl eines anderen *threads* gestartet.
- *Block interleaving*: Es wird dann auf einen anderen Faden umgeschaltet, wenn beim gegenwärtigen Faden eine Blockierung auftritt.
- *Simultaneous multithreading*: In einer superskalaren Architekturen werden Befehle von mehreren Fäden gleichzeitig gestartet.

Mehrfädige Architekturen zielen v.a. auf eine Verbesserung des Gesamtdurchsatzes, die Anwortzeit für einen einzelnen Prozess wird durch sie nicht verbessert. Sie eignen sich daher insbesondere für Server-Architekturen, weniger für Clients.

3.4.11 Compiler-Unterstützung für *instruction level parallelism*

3.4.11.1 Abrollen von Schleifen

Eine ganze Reihe von leistungsverbessernden Maßnahmen ergibt sich, wenn man den Compiler mit in die Überlegungen einbezieht. Das vielleicht bekannteste Beispiel ist das Abrollen von Schleifen (engl. *loop unrolling*). Zur Einführung betrachten wir das folgende kleine C-Programm.

```
for (i=1; i<=20; i++)
   x[i] = x[i] + s;
```

welches einen skalaren Wert zu Array-Elementen hinzu addiert. Eine geschickte Übersetzung in MIPS-Assemblercode könnte wie folgt aussehen:

```
      Li   $5,80         -- R5:=20*4 (4 Byte / Element; load lower immediate)
      Lw   $2,s          -- R2:=s;
Loop: Lw   $3,&x($5)     -- $3 = Array-Element; &x: Adr. erstes Elem.
      add  $4,$3,$2      -- addiere in R2 befindliches s
      sub  &x($5),$4     -- speichere Ergebnis
      subi $5,$5,4       -- dekrementiere pointer um 4 Bytes
      bne  $5,$0,Loop    -- springe, wenn $5<>$0 (=0)
```

Zusätzlich zu den 5 nützlichen Schleifen-Zyklen würde ein solches Programm möglicherweise noch einige Zyklen mit dem Warten verbringen. Eine drastische Beschleunigung dieser Schleife ist möglich, sofern der Programmcode nicht sehr kompakt sein muss und sofern die Schleifengrenzen fest sind. In diesem Fall kann man nämlich den Schleifenrumpf kopieren. Man spricht in diesem Zusammenhang vom **Abrollen der Schleife**. Bei einem zweimaligen Abrollen kommt man zu der folgenden Version:

```
for (i=1; i<=20; i++)
  { x[i] = x[i] + s; i++;
    x[i] = x[i] + s; }
```

Die Ausführung der beiden Kopien des Rumpfes kann sich zeitlich überlappen und so das Fließband besser füllen. Der Vorteil wird noch deutlicher, wenn -wie beim Pentium- zwei oder mehr Fließbänder vorhanden sind.

Im Extremfall, dem sog. **vollständigen Abrollen**, wird die Schleife komplett aufgelöst:

```
x[1] = x[1] + s;
x[2] = x[2] + s;
x[3] = x[3] + s;
x[4] = x[4] + s;
x[5] = x[5] + s;
...
```

In dieser Lösung ist kein Overhead für die Verwaltung der Schleife erforderlich.

3.4.12 Compiler-Unterstützung für die digitale Signalverarbeitung

DSP-Prozessoren beitzen die in Kapitel 2 erwähnten Eigenheiten, wie z.B. eine begrenzte Parallelität. Auch hierfür kann man eine spezielle Compilerunterstützung vorsehen. Dies ist Ziel der Arbeiten am Lehrstuhl 12 (siehe http://ls12-www.cs.uni-dortmund.de)

3.5 Erwartetete Entwicklung der Fertigungstechnologie

Für die Weiterentwicklung der Prozessoren werden seit einer Reihe von Jahren erstaunlich präzise Vorhersagen der technischen Entwicklung gemacht. Die wohl beste Vorhersage bilden die sog. *Roadmaps* der amerikanischen *semiconductor industries association* (SIA). Z.Z. (2000) ist als neueste Fassung die *Roadmap* von 1997 mit Ergänzungen von 1998 verfügbar [Ass98]. Die Tabelle 3.3 enthält Auszüge dieser *Roadmap* (©SIA).

Tabelle 3.3 zeigt, wie die Anzahl der Transistoren pro Chip, die Taktfrequenz, die Anzahl der Anschlüsse und die Leistung wachsen und die Betriebsspannung fallen sollen.

Jahr	1997	1999	2002	2005	2008	2011	2014
Halber Abstand Leitungsmitten beim DRAM [nm]	250	180	130	100	70	50	35
Transistoren pro Chip, μP	11M	21M	76M	200M	520M	1400M	3620M
Interner Takt, lokal [Mhz] *high-performane chip*	750	1250	2100	3500	6000	10000	16903
Interner Takt [Mhz], across chip *high-performance*	375	1200	1600	2000	2500	3000	3674
Takt Chip/Platine [Mhz] *high-performance chip*	375	1200	1600	2000	2500	3000	3674
Interner Takt [Mhz], across chip *cost/performance optim.*	400	600	800	1100	1400	1800	2303
Anzahl der Anschlüsse *μP, cost/performance optim.*	568	700	957	1309	1791	2449	3350
Minimale Spannung	1.8-2.5	1.5-1.8	1-2-1.5	0.9-1.2	0.6-0.9	0.5-0.6	0.37-0.42
Max. Leistung (Desktop) [W]	70	90	130	160	170	175	183
Max. Leistung (Batterie) [W]	1,2	1,4	2,0	2,4	2,8	3,2	3,7

Tabelle 3.3: Entwicklung einiger Kennwerte

Die Anzahl der Transistoren pro Mikroprozessor-Chip wird weiter exponentiell wachsen, und zwar auf etwa 3,620 Milliarden im Jahr 2014 (siehe auch Abb. 3.45). Dieses Wachstum entspricht *Moore's law* (Verdopplung alle 18 Monate). Dieses Wachstum setzt allerdings erhebliche Innovationen in der Fertigung voraus.

Es ist nicht ganz klar, wozu man diese Anzahl von Transistoren eigentlich verwenden will. Mehr als die heute vorkommenden 4 Fließbänder mit max. 13 Pipelinestufen kann man je Prozessor kaum sinnvoll einsetzen. Viele der Transistoren werden wohl Caches realisieren. Manche Chips werden mehrere Prozessoren enthalten.

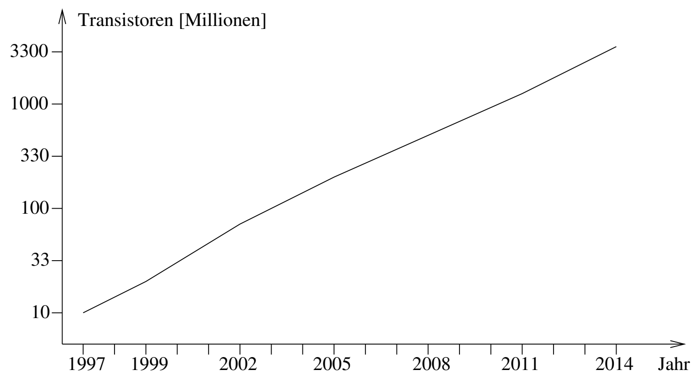


Abbildung 3.45: Wachstum der Anzahl der Transistoren

Innerhalb eines Chips wird man lokal schnelle Taktsignale verwenden, die aus einem langsameren Takt abgeleitet werden, der über den ganzen Chip (*across chip*) verteilt wird. Der Takt, der für die Übertragung zwischen Chip und Platine benutzt wird, kann wiederum ein anderer sein. Der Abbildung 3.46 ist zu entnehmen, dass die Frequenz lokaler Takte stark zunehmen wird. Der Verteilung schnellerer Takte über einen ganzen Chip hinweg sind durch die Laufzeiten auf dem Chip Grenzen gesetzt, bei auch auf Kosten optimierten Chips kann der Takt sogar langsamer sein als zwischen Platine und Chip.

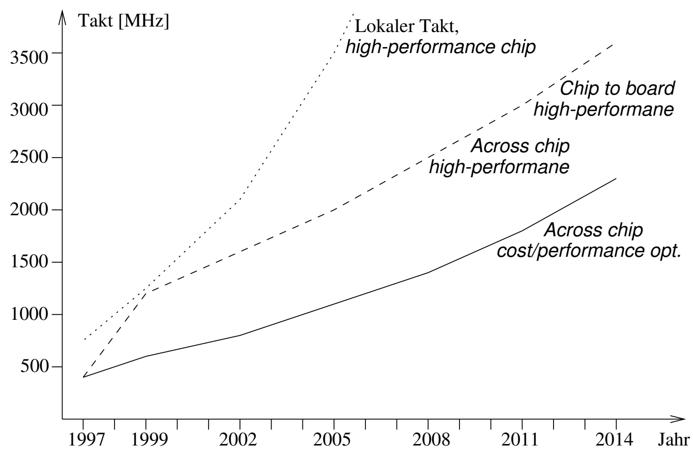


Abbildung 3.46: Taktfrequenzen

Die Anzahl die Anschlüsse soll mit 3350 Größen erreichen, die noch vor wenigen Jahren kaum vorstellbar waren.

Die Versorgungsspannung muss weiter absinken, um den sinkenden Abmessungen gerecht zu werden und um die Verlustleistung nicht zu stark steigen zu lassen (siehe Abb. 3.47).

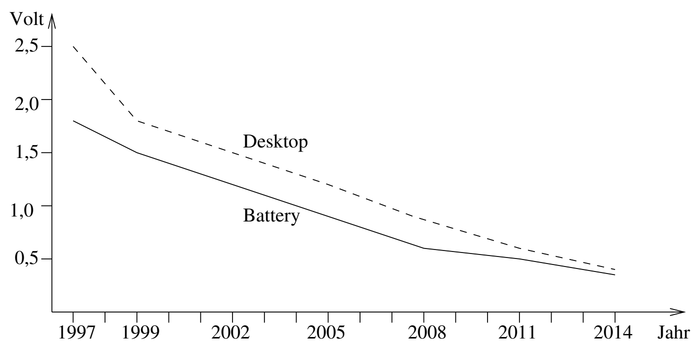


Abbildung 3.47: Reduktion der Versorgungsspannung

Die Verlustleistung selbst kann nur moderat wachsen (siehe Abb. 3.48). Dafür gibt es v.a. zwei Gründe:

erstens kann man per Luftkühlung nicht wesentlich mehr als 150 Watt abführen und Wasserkühlung ist für die meisten Anwender nicht akzeptabel (obwohl sie bei Rechenzentren effizienter wäre). Zweitens ist die Stromaufnahme ohnehin schon immens: bei 183 Watt und 0,42 Volt ergibt sich ein Strom von 435 Ampere (!). Bei nur 0,1 Milliohm Widerstand auf der Stromversorgungsleitung würden auf dieser bereits 10% der Betriebsspannung abfallen!

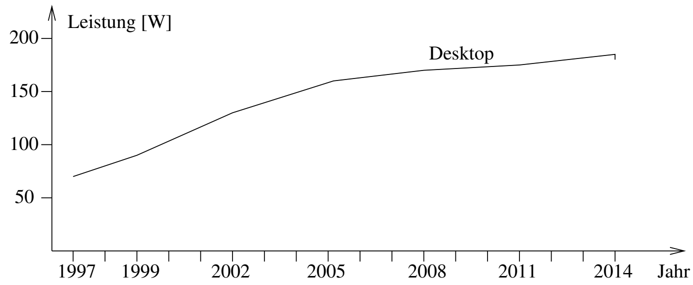


Abbildung 3.48: Vorhersage der Leistungsaufnahme von Prozessor-Chips

Die Leistungsaufnahme portabler Geräte ist weitgehend durch die verfügbare Akkutechnik begrenzt.

Kapitel 4

Speicherarchitektur

640k will be enough for everyone

[Bill Gates, 1981]

Im folgenden Kapitel werden wir die Speicher-Einheit behandeln. Dementsprechend ist diese in der Abbildung 4.1 hervorgehoben.

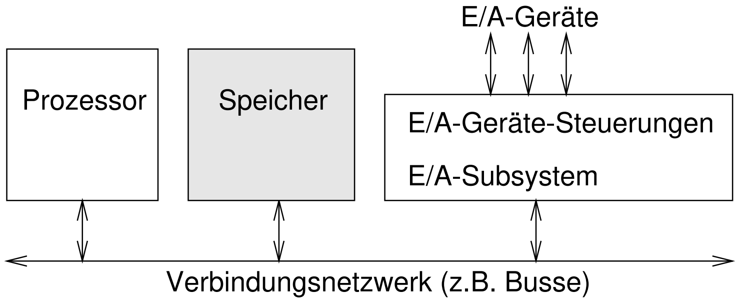


Abbildung 4.1: Speicher-Einheit

4.1 Speicherhardware

Im folgenden Abschnitt behandeln wir zunächst die verfügbaren Speicherkomponenten[1]. Wir tun dies in aller Kürze, da diese Komponenten nicht zu den zentralen Themen der Rechnerarchitektur gehören. Allerdings müssen sie hier erwähnt werden, da die logische Struktur heutiger Speicher zum guten Teil durch die verfügbare Technologie bestimmt ist und Kenntnisse darüber leider nicht vorausgesetzt werden können.

Die Abb. 4.2 zeigt eine Übersicht über gegenwärtig verfügbare Komponenten zur Realisierung von Speichern.

Um eine größerere Anzahl von Speicherzellen ansprechen zu können, werden die Speicherzellen in allen angegebenen Fällen in einer Matrix angeordnet (siehe Abb. 4.3).

Ein Teil der Adresse dient dann der Auswahl einer Zeile. Meist wird eine vollständige Zeile gelesen oder geschrieben. Die Auswahl eines Teiles einer Zeile erfolgt in der **Spaltenauswahl**. Diese besteht für das Lesen in der Regel aus einem Multiplexer. Für das Schreiben ist für die verschiedenen Speichertypen eine unterschiedliche Realisierung der Spaltenauswahl erforderlich.

[1] Siehe auch: Abschnitte 4.2 bis 4.4 und 4.4 des Buches von Bähring

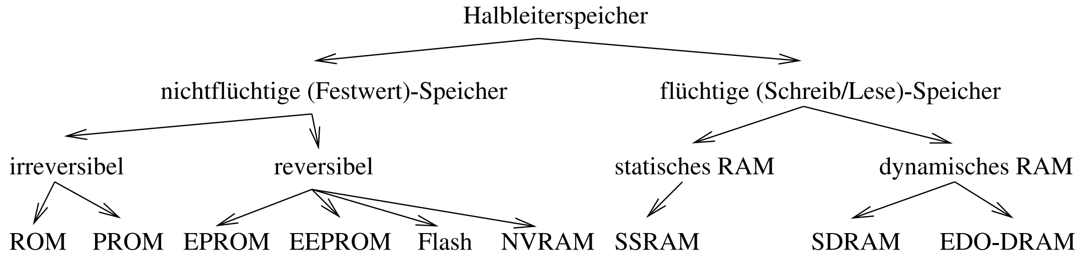


Abbildung 4.2: Klassifikation verfügbarer Speicherkomponenten

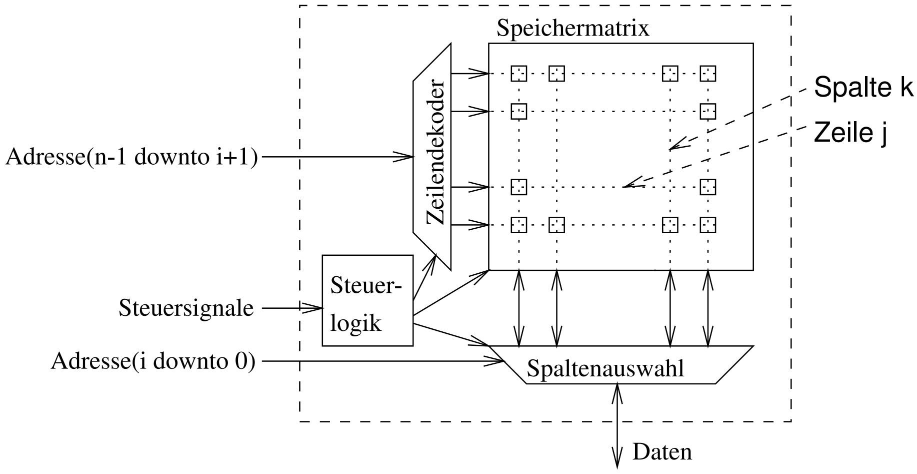


Abbildung 4.3: Matrix-Anordnung gängiger Speicher

4.1.1 Nichtflüchtige Speicher

Zunächst besprechen wir die nichtflüchtigen Speicher:

- ROM-Bausteine

  Abb. 4.4 zeigt Speicherelemente eines ROM-Speichers. Die Werte an der Datenleitung k sind durch die Verschaltung an der Kreuzung von Zeilen- und Spaltenleitung bestimmt. In der gezeigten Version liefert die Datenleitung eine '1', falls an der Kreuzung mit der ausgewählten Zeile eine Diode angeschlossen ist.

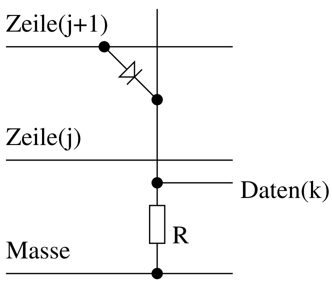


Abbildung 4.4: Elemente der Speichermatrix von ROMs

  Andere Versionen von ROMs benutzen als Schaltungselemente an der Kreuzung Transistoren. In jedem Fall wird bei der Fertigung entschieden, ob an einem Kreuzungspunkt ein solches Schaltungselement effektiv in Erscheinung treten wird oder nicht.

- PROM-Bausteine

  PROM-Bausteine (*programmable read only memories*) sind ähnlich aufgebaut wie ROM-Bausteine, jedoch wird die Information in diesen nicht bereits im Halbleiterwerk festgelegt. Die Programmierung erfolgt hier später beim Schaltungshersteller in speziellen PROM-Programmiergeräten. Diese sorgen beispielsweise durch Überspannung für ein Durchschmelzen der Schaltungen an den Kreuzungspunkten (vgl. Abb. 4.5). Dieser Vorgang ist irreversibel.

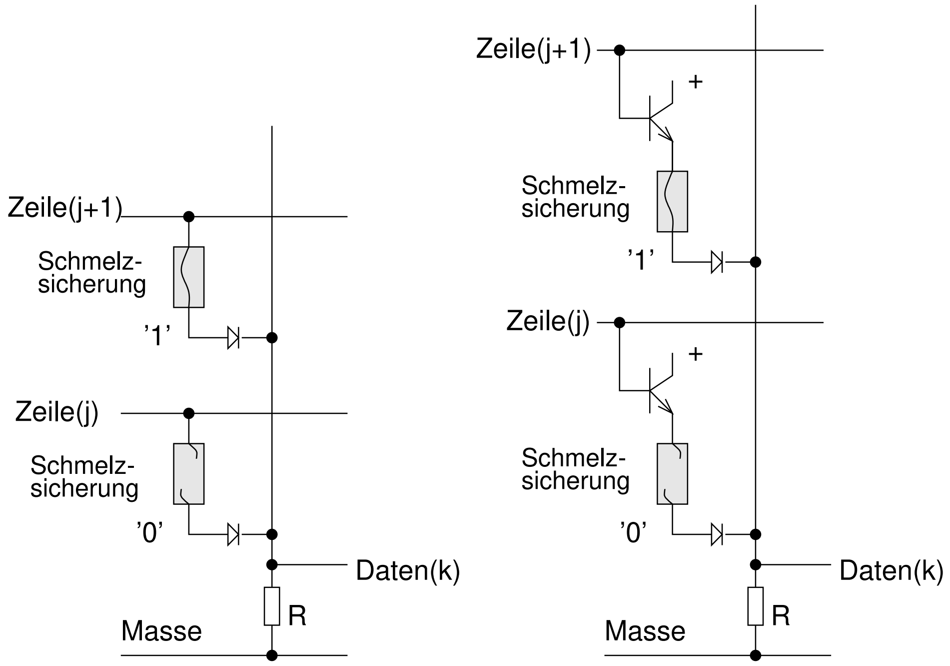
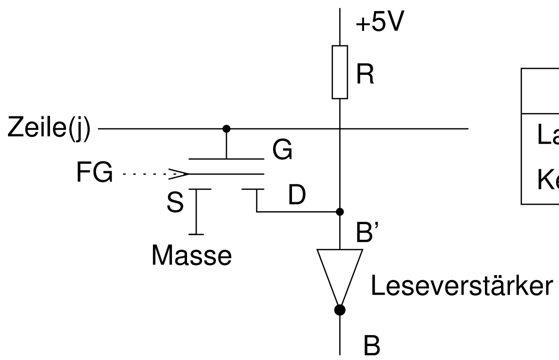


Abbildung 4.5: Speicherzelle von PROMs

- EPROM-Bausteine

  In Unterschied dazu handelt es sich bei EPROMs um *erasable programmable read only memories*, also um wieder löschbare Speicher. Bei diesen befinden sich an den Kreuzungspunkten der Speichermatrix spezielle MOS-Transistoren. Diese MOS-Transistoren besitzen zwischen Gate und Kanal eine isolierte leitende Zone, das sog. *floating gate*. Auf dieses kann mit hoher Spannung Ladung aufgebracht werden. Durch ein Fenster des Gehäuses kann die Gatezone mit ultraviolettem Licht bestrahlt werden. Dieses führt zu einer Entladung des Gates. Je nach Zustand des *floating gates* werden beim Lesen Nullen oder Einsen gelesen (vgl. Abb. 4.6).



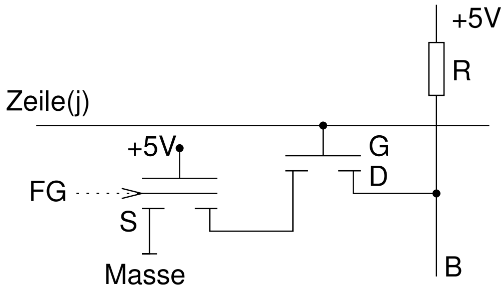
	B'	B
Ladung auf dem FG	'1'	'0'
Keine Ladung auf dem FG	'0'	'1'

Abbildung 4.6: Lesen von EPROM-Speicherzellen

- EEPROM-Bausteine

  Um das Löschen auch elektrisch auslösen zu können, sind die sog. *electrically erasable programmable read only memories* (EEPROMs) entwickelt worden. Diese benötigen pro gespeichertem Bit einen weiteren Transistor, sind also komplexer als EPROMs. Das Schreiben erfolgt auch hier wieder mit Überspannungen. Zum Lesen siehe Abb. 4.7.

- *Flash memories*



	B
Ladung auf dem FG	+5V '0'
Keine Ladung auf dem FG	0V '1'

Abbildung 4.7: Lesen von EEPROM-Speicherzellen

Flash memories sind nichtflüchtige, wiederbeschreibbare Speicher. Für das Beschreiben von Flash-memories werden weder eine besondere Überspannung noch UV-Licht benötigt. Sie werden über den normalen Daten- und Adressbus mittels speziellen Befehlen gelöscht und beschrieben. Sie sind preislich erheblich günstiger als EEPROMS. Ihr Einsatzgebiet liegt v.a. in der Speicherung von Software, die gelegtlich Revisionen unterworfen sein könnte, wie z.B. in Betriebssystem-Basissoftware, in BTX-Decodern oder in Mobiltelephonen. Als Arbeitsspeicher können sie nicht eingesetzt werden, da die Anzahl möglicher Schreibvorgänge begrenzt ist (ca. $10^5 - 10^6$ Schreibvorgänge) und da nicht wie beim RAM wahlfrei geschrieben werden kann.

- NVRAM-Bausteine

  NVRAMs (engl. *non-volatile RAMs*) sind Speicher, die im Betrieb mit der normalen Geschwindigkeit von flüchtigen Speichern arbeiten können. Sie bestehen intern aus einem relativ schnellen, flüchtigen Speicher und einem (langsameren) EEPROM. Beim Absinken der Betriebsspannung und vor dem drohenden Verlust der Information im flüchtigen Speicher kann der Inhalt des flüchtigen Speichers in das EEPROM kopiert werden. Bei der Rückkehr der Betriebsspannung kann der umgekehrte Vorgang gestartet werden.

  Derartige Speicher sind beispielsweise sehr wichtig zur Realisierung von Puffern, die häufig benötigte Plattendaten zwischenspeichern, den sog. **Plattencaches**. Viele Datenbanksysteme verlassen sich darauf, dass nach dem Abschluss des Schreibens einer Datei ein jederzeit wieder herstellbarer Zustand auf der Platte gesichert ist. Tatsächlich wird aber zur Verbesserung der Performance vielfach nur in den Plattencache geschrieben. Ist dieser in einem NVRAM-Bereich des Speichers realisiert, kann die Information nach Absturz und erfolgtem Neustart tatsächlich in das Dateisystem übernommen werden.

4.1.2 Flüchtige Speicher

Leider sind alle schnellen beschreibbaren Speicher heute flüchtige Speicher (engl. *volatile memories*), d.h. sie verlieren ihre Information kurz nach dem Ausschalten der Betriebsspannung. Da alle Zellen ohne Einschränkungen adressiert werden können, heißen sie *random access memories* (RAMs). Wir unterscheiden dabei zwischen den statischen Speichern und den dynamischen Speichern:

- Statische Speicher

  Statische Speicher enthalten pro gespeichertem Bit eine aus sechs Transistoren bestehende Registerzelle, auch Flip-Flop genannt (siehe Abb. 4.8).

  Die Transistoren T3 und T2 funktionieren dabei in erster Näherung wie Widerstände, lassen sich aber in integrierter Technik leichter herstellen als echte Widerstände.

  Sofern die Adressleitung eine '1' führt, verbinden die Transistoren T4 und T5 die eigentliche Speicherzelle (bestehend aus T0 bis T3) mit den beiden Datenleitungen. Beim Schreiben sind diese Datenleitungen mit einem **Ausgang** verbunden. Dieser Ausgang zwingt so der Zelle die Information auf. Sofern die beiden Datenleitungen tatsächlich komplementär sind (was dem gewünschten Betrieb entspricht), ist der resultierende Zustand der Speicherzelle stabil und bleibt auch erhalten, wenn die Adressleitung auf '0' gesetzt wird.

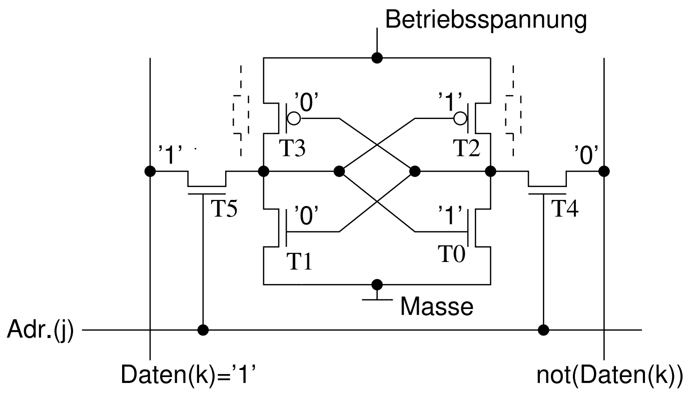


Abbildung 4.8: Speicherzelle statischer RAMS

Für das Lesen muss die Adressleitung wieder auf '1' sein. Die Datenleitungen werden in diesem Fall nicht aktiv getrieben, sondern mit einem **Eingang** der Spaltenauswahl-Schaltung verbunden. Die Transistoren T4 bzw. T5 (von denen nur einer zum Lesen benötigt wird) übertragen in diesem Fall die gespeicherte Information an diesen Eingang.

- Dynamische Speicher

  Die 6-Transistorzelle statischer Speicher benötigt pro Bit verhältnismäßig viel Platz. Man ist daher schon sehr früh dazu übergegangen, die Anzahl der Transistoren pro Bit zu reduzieren. Das Resultat ist die 1-Transistor-Speicherzelle, deren Prinzip in Abbildung 4.9 zu sehen ist.

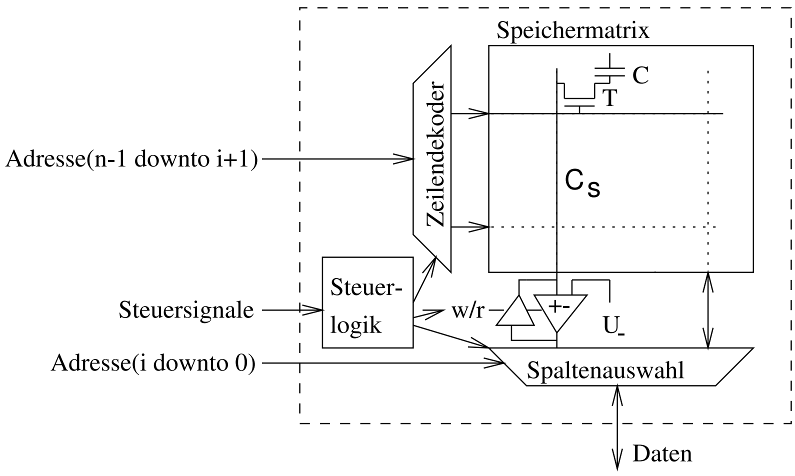


Abbildung 4.9: Speicherzelle dynamischer RAMS

Beim Schreiben ist die Zeilenleitung auf '1' zu setzen und die Datenleitung an einen Ausgang anzuschließen. Transistor T leitet und überträgt die Spannung auf der Datenleitung auf den Kondensator C. Dieser Kondensator ist sehr klein (< 0,1 pF) und wird tatsächlich lediglich durch die Kapazität des offenen Transistoranschlusses gebildet.

Das Lesen wird dadurch erschwert, dass die Kapazität der Datenleitung wesentlich größer ist als die des Kondensators C. Dies liegt v.a. daran, dass diese Leitung bei kleineren Speicherchips über die volle Länge des Chips, bei moderneren Chips immerhin noch über die Hälfte, ein Viertel oder ein Achtel der Länge zu führen ist. Daher muss der Lesevorgang wie folgt ablaufen: Zunächst wird die Datenleitung mit einem Differenzverstärker am Eingang der Spaltenauswahl verbunden. Der zweite Eingang des Differenzverstärkers wird auf eine Spannung U_- gelegt. U_- liegt möglichst genau zwischen der Spannung, die sich bei einer gespeicherten '0' und der Spannung, die sich bei einer gespeicherten '1' auf der Datenleitung einstellt. Der Differenzverstärker wird an seinem Ausgang eine '0' erzeugen, falls die Datenleitung eine etwas kleinere Spannung als U_- besitzt und eine '1', falls diese Spannung etwas größer ist als U_-. Nun wird eine '1' auf die ausgewählte Adressleitung geschaltet. Als Ergebnis öffnet der Transistor und es findet eine Ladungsverteilung auf Datenleitung und Kondensator C statt. Diese hat eine winzige Spannungsänderung zur Folge, die vom Differenzverstärker in ein logisches Signal umgesetzt wird.

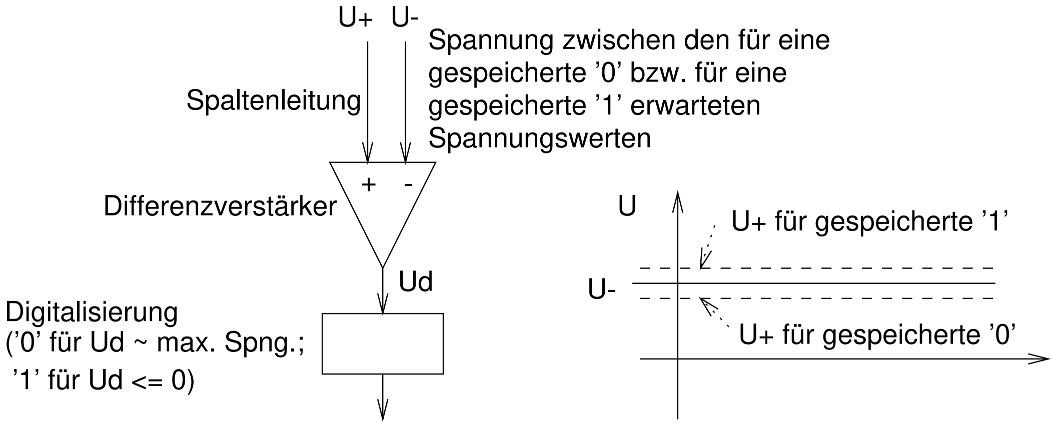


Abbildung 4.10: Lesevorgang mittels Differenzverstärker

Zu beachten ist, dass Kondensatoren nach dem Lesen nicht mehr die ursprüngliche Spannung enthalten, dass das Lesen also **zerstörend** ist. Also muss die gelesene Information anschließend wieder eingeschrieben werden. Dies gilt für die vollständige gelesene Zeile, auch für jene Spalten, die gar nicht benötigt wurden.

Da die Ladung auf den Kondensatoren nicht beliebig lange erhalten bleibt, müssen DRAMs periodisch alle Zeilen lesen und wieder zurückschreiben. Diese *refresh*-Zyklen werden entweder durch einen externen **DRAM-Controller** angestoßen oder -bei neueren Speichern- intern im Speicherchip erzeugt. Sie sind in Intervallen von einigen Millisekunden erforderlich. Man sollte diese Zyklen möglichst so organisieren, dass normale Speicherzugriffe durch sie nicht in ihrer Geschwindigkeit beeinflusst werden (sog. *hidden refresh*). Falls kein *hidden refresh* realisiert ist, ist das Realzeitverhalten von Programmen schlecht vorherzusehen.

Ein Beispiel soll die Zahlenverhältnisse vor Augen führen: der Abstand zwischen Refreshzyklen liegt gegenwärtig bei ca. 8 ms, die Zugriffszeit bei ca. 32 ns, das Verhältnis also bei 250.000.

In jüngster Zeit hat es einige Sonderentwicklungen flüchtiger Speicher gegeben, welche die effektive Speicherbandbreite erhöhen sollen:

- *EDO-RAMs*

  "EDO" steht dabei für *extended data out.* Bei diesen Speichern stehen die Daten am Ausgang auch noch eine Zeit nach dem Entfernen der Adresse zur Verfügung. Damit kann bereits mit dem Anlegen der nächsten Adresse begonnen werden, während noch die Daten zur vorherigen Adresse gelesen werden. Durch diese Art des Fließbandzugriffs wird die effektive Speicherzugriffs-Geschwindigkeit etwas erhöht.

- **synchrone statische** bzw. **synchrone dynamische Speicher** (SSRAMs bzw. SDRAMs)

  Synchrone statische bzw. synchrone dynamische Speicher enthalten zusätzlich zur Speichermatrix Adress- und Datenregister, die mit dem Prozessortakt verbunden werden können. Damit kann eine dreistufige Pipeline aufgebaut werden und der Durchsatz wird im Vergleich zum üblichen "asynchronen" RAM höher. Zum Teil werden auch noch intern verschiedene Speicherbänke aufgebaut. Damit ist dann auch beim Lesen der Speichermatrix selbst noch ein Pipelining möglich.

4.1.3 Multiportspeicher

Aus den bislang vorgestellten Bausteinen lassen sich sog. **Mehrportspeicher** (engl. *multiport memories*) aufbauen. Da Speicher in mehreren Blöcken realisiert sind, geht eine mögliche Nebenläufigkeit verloren, wenn zu jedem Zeitpunkt höchstens eine Speicheroperation aktiv sein kann. Speziell für Prozessoren mit mehreren Pipelines, für Mehrprozessoranlagen, aber auch zur Beschleunigung von E/A-Operationen, realisiert man Mehrportspeicher.

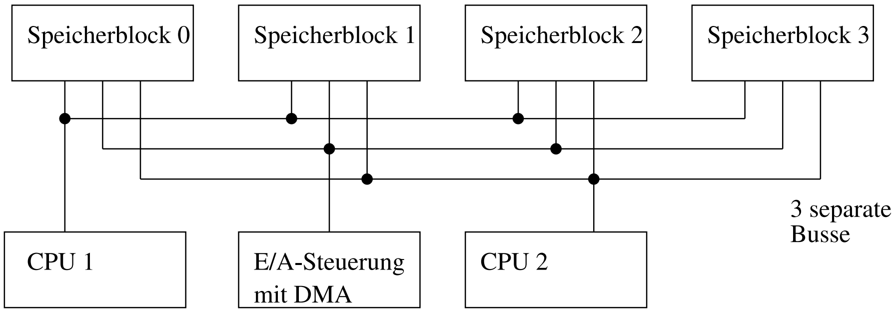


Abbildung 4.11: 3-Port Speicher

Anhand der von einem Prozessor abgelieferten Adresse wird von einer Steuerlogik entschieden, welcher Speicherblock für das Datum zuständig ist. In der Regel ist die Zahl der Speicherblöcke eine Zweierpotenz und die Entscheidung erfolgt anhand der am wenigsten signifikanten Bits der Speicherwortadressen. Sofern alle Prozessoren auf verschiedene Blöcke zugreifen wollen, behindern sie sich gegenseitig nicht. Ansonsten entscheidet eine Vorranglogik über die Reihenfolge der Abarbeitung. Ein grobes Schema zeigt die Abbildung 4.11.

Um Leitungen einzusparen, sind auch Varianten im Gebrauch, bei denen Speicher und Prozessoren an einem einzigen im Zeitmultiplex betriebenen Bus angeschlossen sind.

Als integrierte Schaltungen sind Multiportspeicher mit einer kleinen Zahl von Ports (2-3) erhältlich. Im Fließband des Kapitels 3.1 wurde ebenfalls ein 3-Port Speicher benutzt. In modernen superskalaren Rechnern werden zur Realisierung des Registersatzes sehr viel mehr Ports benötigt. 1996/97 waren offenbar Werte von 11-13 Ports üblich.

4.1.4 Entwicklung der Speichertechnologie

Die *SIA-Roadmap* enthält auch Aussagen zur Entwicklung der Speichertechnologie (siehe Tabelle 4.1).

Jahr	1997	1999	2002	2005	2008	2011	2014
Halber Abstand Leitungsmitten beim DRAM [nm]	250	180	130	100	70	50	35
Bits proChip (DRAM)	267 M	1,07 G	4,29 G	17,2 G	68,7 G	275 G	1,1 T

Tabelle 4.1: Entwicklung der Speicherkapazität

Die Steigerung liegt bei einem Faktor von 4 in 3 Jahren.

4.2 Speicherverwaltung

Neben der physikalischen Realisierung von Speichern ist v.a. die Organisation der Zuordnung von Speicher zu Benutzerprozessen wichtig. In diesem Zusammenhang müssen wir zunächst verschiedenen Adressen unterscheiden:

Def.: **Prozessadressen** bzw. **virtuelle Adressen** sind die effektiven Adressen nach Ausführung aller dem Assemblerprogrammierer sichtbaren Adressmodifikationen, d.h. die von einem Prozess dem Rechner angebotenen Adressen[2].

Def.: **Maschinenadressen** bzw. **reale Adressen** sind die zur Adressierung der realen Speicher verwendeten Adressen.

[2] Quelle: Buch von Jessen [Jes75].

Im folgenden betrachten wir verschiedene Methoden, Prozessen Speicher zuzuordnen. Als erstes betrachten wir Methoden, die angewandt werden können, wenn die Prozessadressen aufgrund der Hardware-Gegebenheiten gleich den Maschinenadressen sein müssen. Wir nennen diesen Fall den Fall der Identität (von Prozessadressen und Maschinenadressen).

4.2.1 Identität

Bei einfachen Rechensystemen werden die von den Maschinenbefehlen erzeugten Adressen auch direkt zur Adressierung des Speichers benutzt, virtuelle Adressen sind also gleich den realen Adressen. Die Größe des virtuellen Adressraums ist dann natürlich durch Ausbau des realen Speichers beschränkt. Dieser Fall wird bei Tanenbaum als *memory management without swapping or paging* bezeichnet [Tan92]. Dieser Fall kommt bei neueren PCs oder Workstations kaum noch vor, wohl aber bei **eingebetteten Prozessoren**, z.B. in der Fahrzeugelektronik.

Sehr einfache Rechnersysteme erlauben lediglich einen einzigen Prozess. In diesem Fall beschränkt sich die Aufteilung des Speichers auf die Aufteilung zwischen Systemroutinen und dem Benutzerprozess.

Bei Systemen, die Mehrprozessbetrieb erlauben, findet man häufig Einteilung des Speichers in feste **Laufbereiche** oder *core partitions*. Prozesse werden den Partitionen aufgrund ihres angemeldeten Speicherbedarfes fest zugeordnet (vgl. Abb. 4.12).

	Betriebssystem	Prozesse
→	- - - - - - - - - - -	
Grenzregister	Partition 1	1,2
→	- - - - - - - - - - -	
(Umladen bei	Partition 2	4,5
Prozesswechsel)	- - - - - - - - - - -	
	Partition 3	3
	- - - - - - - - - - -	
	Partition 4	6,7,8

Abbildung 4.12: Aufteilung des Speichers in Core-Partitionen

Da zusammenhängende Adressbereiche im virtuellen Adressraum benutzt werden, sind auch die Bereiche im realen Adressraum zusammenhängend.

Eine Ausdehnung über Partitionsgrenzen hinaus (siehe Abb. 4.13) ist meist nicht realisiert.

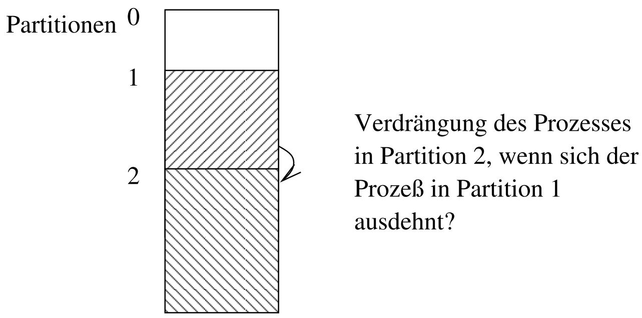


Abbildung 4.13: Zur Ausdehnung über Partitionsgrenzen

Eigenschaften dieser Organisationsform sind die folgenden:

- Es ist kein **Verschieben** (siehe Abb. 4.14) möglich, nachdem Programme einmal geladen und gestartet sind, da in der Regel Daten und Adressen im Speicher für das Betriebssystem ununterscheidbar gemischt werden und nur letztere zu modifizieren wären.

  Ausnahme: Ausschließlich PC-relative Adressierung (würde bei rekursiven Prozeduren ein Kopieren des Codes erfordern).

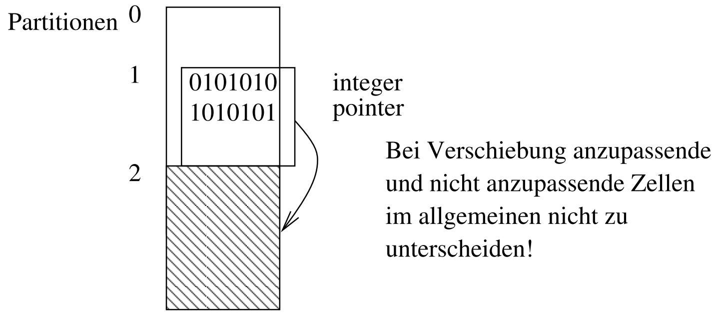


Abbildung 4.14: Zum Verschieben von Partitionen

- Es kann nur der gesamte, einem Prozess zugeordnete Speicher auf die Platte gerettet werden (sog. *swapping*). Er muss an die gleiche Stelle zurückgeschrieben werden.
- Schutz der Prozesse gegeneinander ist durch Grenzregister möglich; in diesem Fall ist der (schreibende) Zugriff nur zwischen Grenzregistern erlaubt.
  Verbleibende Probleme: Zugriff auf gemeinsame Daten; Schreibschutz für Code; Verhindern des lesenden Zugriffs auf private Daten (*privacy*).

  Hardware-mäßig gibt es genau **zwei** Grenzregister. Diese enthalten die Grenzen des **laufenden** Prozesses (Annahme: Monoprozessorsystem). Diese Grenzregister werden beim Prozesswechsel aus dem Prozesskontrollblock geladen (siehe Kap. 2).
- Compiler erzeugen in der Regel Code, der ab der Adresse 0 beginnend aufsteigende Adressen benutzt. Da den Prozessen in der Regel nicht bei 0 beginnende Speicherbereiche zugeordnet werden können, müssen die vom Compiler erzeugten Adressen meist um die Anfangsadresse des Speicherbereichs erhöht werden.

  Ebenso können getrennt übersetzte Prozeduren, für die der Compiler ebenfalls bei 0 beginnende Adressen erzeugt, nicht ab dieser Adresse in den Speicher geladen werden und bedürfen daher einer Adressanpassung entsprechend ihrer Lage im Speicherbereich des aufrufenden Prozesses.

  Dazu ist *relocating information* im Binärfile erforderlich.

Exkurs: Darstellung von Binder- und Laderinformation

Wir betrachten hier exemplarisch das Unix-a.out-Format, welches in vielen Unix-Systemen in einer erweiterten Form als *common object file format* (COFF) benutzt wird[3].

Die folgende vereinfachte Darstellung zeigt die wesentlichen Informationsblöcke dieser Formate:

n (Zahl der Codeworte)
$\updownarrow$ n Codeworte
$\updownarrow$ n Worte Binder/Lader-Information
$\updownarrow$ n Worte Zeilennummern (nur COFF)
Symboltabelle (z.B. zur Abbild. ASCII-Strings $\rightarrow$ int. Zahl)

Das Wort i des 2. Blocks enthält kodierte Binder/Lader-Information für das Wort i des Codeblocks.

Der Block **Binder/Lader-Information** enthält dabei die folgenden Informationen:

- Relocating? (ja/nein). Für welches Segment? (Code, Stack, Daten, ..)
- Internes Symbol: Adresse eines nach außen sichtbaren internen Symbols (z.B. Adresse der Einsprungstelle in ein Unterprogramm).

[3] Im Windows-Bereich ist für denselben Zweck das DLL-Format definiert worden [Lev00].

- Externes Symbol: Adresse, an der Wert eines externen Symbols eingesetzt werden muss (z.B. Aufrufstelle eines externen Unterprogramms).
- Reservierung von Speicherplatz
- Initialisierung von Speicherplatz
- Startadresse

Das Wort i des 3. Blocks enthält die Zeilennummer des entsprechenden Wortes des Codeblocks.

Ein Beispiel zeigt die Abb. 4.15 (Annahme: 16 Bit für Befehle & Adressen).

		Assembler-Listing-	Binärfile
		PROGRAM q	5 (Länge des Codes)
		EXTERNAL p	
		INTERNAL a	
0	b:	MOVE a,R1	(MOVE) ..,1
1			4
2		JMP p	(JMP)
3			0 (Offset relativ zu p)
4	a:	DC 3	3 (abs. Initialwert)
		END	(ABS)
			(REL)
			(ABS)
			(EXT. REF. SYMBOL 1)
			(ABS)
			<Symbol-Typ,Name,Nr,Wert>
			ext, “p”, 1, undef
			int, “a”, 2, 4
		SUBROUTINE p	5 (Länge des Codes)
		EXTERNAL a	
		INTERNAL p	
0	p:	MOVE a,R1	(MOVE) ..,1
1			0
2		MOVE R1,b	(MOVE) 1,..
3			4 (relative Adr. von b)
4	b:	DC 0	0 (abs. Initialwert)
		END	(ABS)
			(EXT. REF. SYMBOL 1)
			(ABS)
			(REL)
			(ABS)
			<Symbol-Typ,Name,Nr,Wert>
			ext, “a”, 1, undef
			int, “p”, 2, 0

Abbildung 4.15: Darstellung von Binder- und Laderinformation

Übungsaufgabe: Man überlege sich, wie die echte Speicherbelegung aussieht, wenn das Programm und das Unterprogramm ab einer gewissen Anfangsadresse in den Speicher geladen worden sind.

4.2.2 Seitenadressierung (Paging)

Die nächste Form der Speicherzuordnung ist die der Seitenadressierung (engl. *paging*). Hierbei erfolgt eine Einteilung des virtuellen Adressraumes in Bereiche konstanter Länge. Diese Bereiche heißen **Seiten** (Pages). Weiter erfolgt eine Einteilung des realen Adressraumes in gleich große Bereiche. Diese Bereiche heißen **Seitenrahmen** oder **Kacheln**. Die Größe der Bereiche ist praktisch immer eine 2er-Potenz und liegt zwischen 512 Bytes und 8 KBytes (mit steigender Tendenz). Den Seiten werden nun Kacheln zugeordnet und zur

Laufzeit werden die Seitennummern durch die Kachelnummern ersetzt. Eine mögliche Zuordnung von Seiten zu Kacheln zeigt die Abb. 4.16.

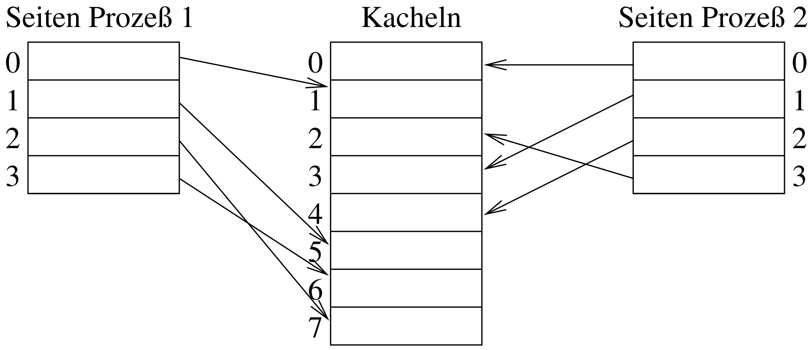


Abbildung 4.16: Zuordnung von Seiten zu Kacheln bei der Seitenadressierung

Der große Vorteil der Seitenadressierung ist, dass jede Kachel jede Seite aufnehmen kann. Dadurch ist ein Verschieben von Speicherinhalten nie erforderlich und jeder Bereich des Speichers kann genutzt werden (keine **externe Fragmentierung**). Weitere Eigenschaften sind die folgenden:

Relocating: Jeder Prozess kann an der virtuellen Adresse 0 beginnen. Nur für das Binden ist eine Adressenkorrektur erforderlich.

Auslagern: Auslagern einzelner Seiten und Rückschreiben in beliebige Kachel ist möglich

Man kann zwischen folgenden Varianten unterscheiden:

Demand Paging: (mit automatischem Einlagern von der Platte)

Paging (ohne *demand*): Ohne autom. Einlagern; nur als Hilfe für die Speicherverwaltung; Anwendung z.B. bei Realzeitsystemen, bei denen unbegrenzte Zeiten für Plattenzugriffe nicht zulässig sind

Pre-Paging: Versuch des vorsorglichen Einlagerns, ohne Seitenfehler abzuwarten

Zu jedem Prozess muss zu jedem Ausführungszeitpunkt eine Abbildung **Seitennummern → Kachelnummern** definiert sein, mit Hilfe derer die reale Adresse gefunden werden kann. Bei der Neuzuordnung von Seiten kann sich diese Abbildung ändern.

Einer besonderen Betrachtung bedarf die Benutzung von Daten oder Code (in Form sog. *shared libraries*) durch mehrere Prozesse. Eine einfache Idee besteht darin, die Seiten-Abbildungen mehrerer Prozesse dieselbe Kachel zuordnen zu lassen, wie in Abb. 4.17 gezeigt. Darin steht LDT für eine Tabelle, die den Seiten Kacheln zuordnet.

Eine genauere Überlegung zeigt, dass diese Methode nur für solche Speicherbereiche funktioniert, die keine Adressen beinhalten. Das folgende Beispiel (siehe Abb. 4.18[4]) zeigt, dass ein Problem existiert, falls die gemeinsamen Seiten Adressen enthalten (siehe auch: Bic [BS88], Abb. 5-26).

Ein gemeinsam benutztes Bibliotheksegment Xlib sei im Prozess A der virtuellen Seitennummer 3 zugeordnet. Diese sei per Seitentabelle A auf die Kachel 3 abgebildet. Wir wollen zunächst annehmen, dass dieselbe Bibliothek im Prozess B der virtuellen Seitennummer 6 zugeordnet sei und dass diese Seitennummer über die Seitentabelle B ebenfalls der Kachel 3 zugeordnet wird. Zunächst sieht es so aus, als würde dies funktionieren. Wenn jetzt aber Xlib eine (absolute) Adresse enthält, die sich z.B. auf Xlib selbst bezieht (also beipielsweise einen Sprung an den Anfang von Xlib), so muss für diese Adresse eine passende Seitennummer im realen Speicher eingetragen werden. Die Seitennummer muss so gewählt werden, dass sie sowohl bei Abbildung über die Seitentabelle A wie auch bei Abbildung über die Seitentabelle B auf die Kachel 3 abgebildet wird. Wegen des Prozesses A müsste die Speicherzelle in Xlib also eine Seitennummer 3, wegen Prozesses B eine

[4]Der Übersichtlichkeit halber geht diese Abbildung von kleinen Seitenzahlen und einer regelmäßigen Adressabbildung aus.

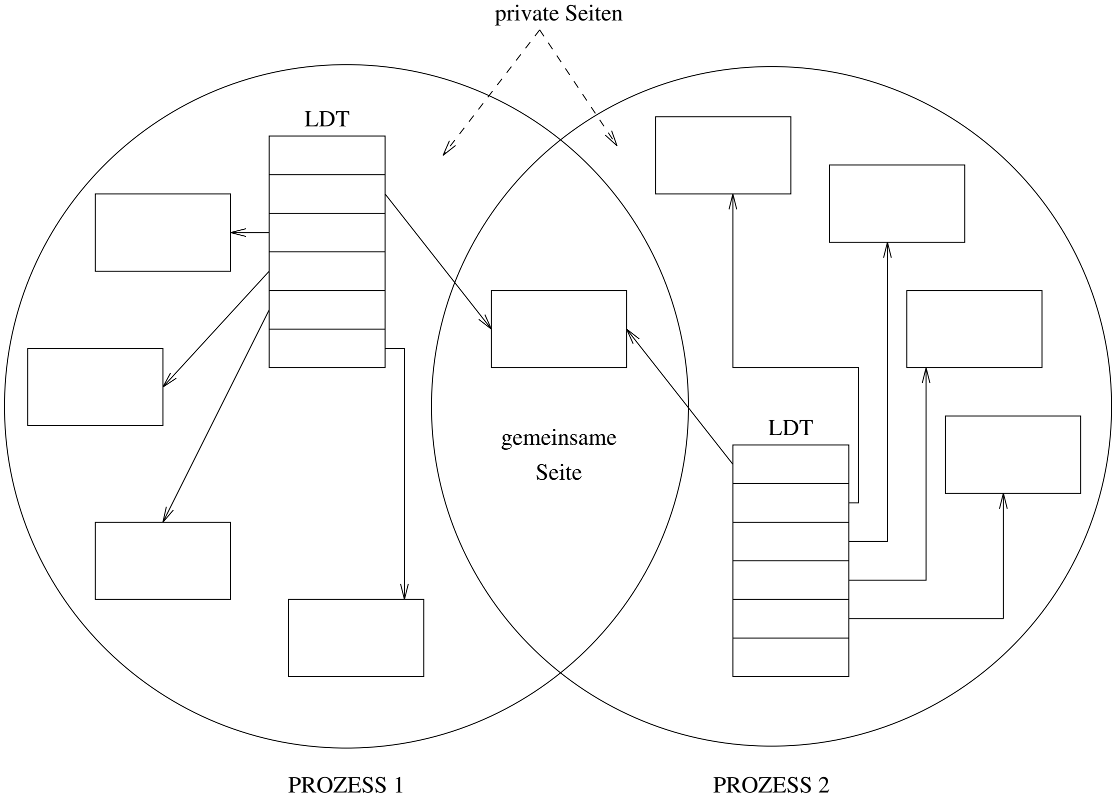


Abbildung 4.17: Sharing für adressenfreie Speicherbereiche

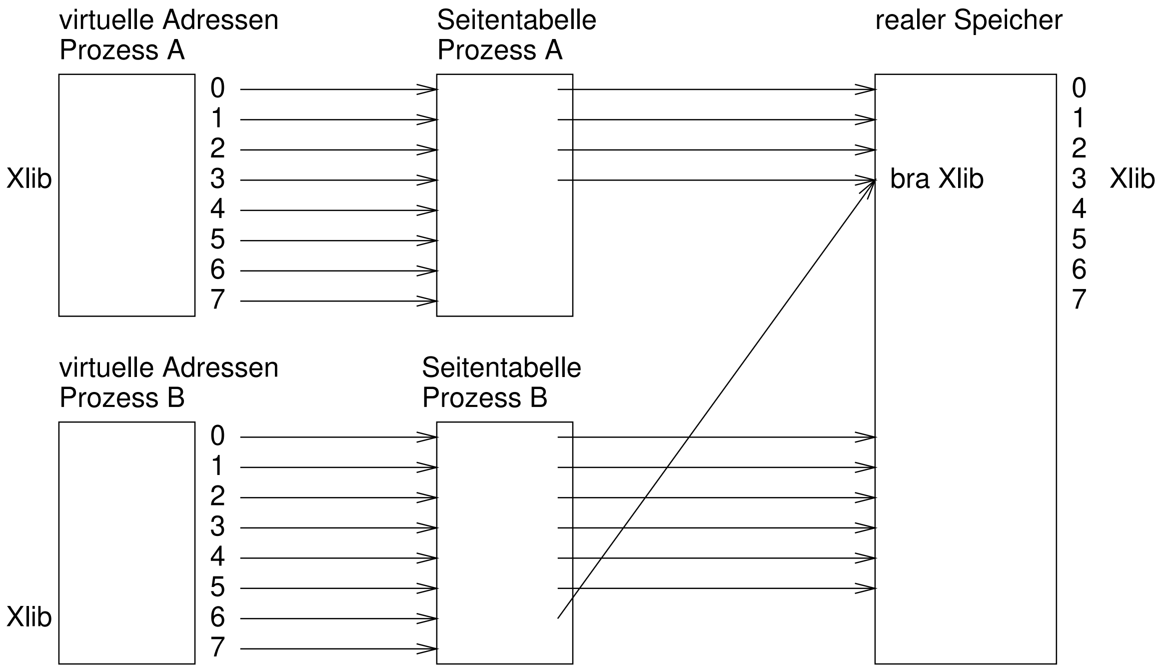


Abbildung 4.18: Problem des Sharings bei der Seitenadressierung

Seitennummer 6 enthalten. Dieselbe Speicherzelle kann bei Ausführung von A und B nicht unterschiedliche Inhalte haben. A und B müssen daher für Xlib dieselbe Seitennummer verwenden. Es folgt: **alle Prozesse, die sich eine Seite teilen, müssen dieser die gleiche Seitennummer geben!**

Die wesentliche Ursache für die Notwendigkeit einer Absprache über die Zuordnung von gemeinsam benutzten Speicher-Bereichen zu Teilen des virtuellen Adressraums liegt darin, dass die Seitenadressierung nicht erlaubt, die vom gemeinsam benutzten Code verwendeten Seitennummern von dem aufrufenden Prozess abhängig zu machen.

Eine Folge davon ist, dass Windows 9x die obere Hälfte des virtuellen Adressraums für *shared libraries* reserviert und dass für diese Hälfte globale Absprachen über die Zuordnung von Code zu virtuellen Adressen notwendig ist.

Im Buch von Bic und Shaw [BS88] wird beschrieben, wie das Sharing unter Multics im Zusammenhang mit dem dynamischen Laden realisiert ist. Unter dem dynamischen Laden versteht man dabei ein Laden angesprochener Segmente, das erst dann erfolgt, wenn dieses Segment auch zur Laufzeit angesprochen wird. Ein besonderes Problem besteht darin, dass der Code beim ersten Ansprechen eines Segments nicht modifiziert werden soll (der Code soll rein bleiben (engl. *pure code*)).

4.2.3 Segmentadressierung

Die nächste Form der Zuordnung von virtuellen Adressen zu realen Adressen ist die Segmentierung. Segmente sind dabei **zusammenhängende Bereiche im Prozessadressenraum, die bezüglich der Zugriffsrechte homogen sind und die sich höchstens an ihrem Ende ausdehnen können**. Segmente entsprechen Speicherbereichen, deren Größe durch das Problem definiert wird. Die Struktur des Programms und der Daten bestimmt die Einteilung des virtuellen Adressraums in Segmente (z.B. Laufzeitstack, Heap, Konstantenbereiche, Code, evtl. auch einzelne Prozeduren). Virtuelle Adressen bestehen aus einer Segmentbezeichnung und einer Adresse innerhalb des Segments, d.h. aus einem Paar: (Segmentbezeichnung, Offset).

In der einfachsten Form werden den Segmenten auch zusammenhängende Bereiche im realen Adressraum zugeordnet. Hierzu reicht es, jedem Segment eine Segmentbasis zuzuordnen, welche zur Laufzeit zur Adresse innerhalb des Segments addiert wird (vgl. Abb. 4.19). Da diese Basis von normalen Benutzern nicht geändert werden kann, heißt diese Organisation auch **Segmentadressierung mit verdeckter Basis**.

- - - - - - - - - - -	← verdeckte Basis
Segment 1	Code für Prozess 1
(Code Prozess 1;	
schreibgeschützt)	
- - - - - - - - - - -	
Segment 2	
(Stack Prozess 2)	
- - - - - - - - - - -	
- f r e i -	
- - - - - - - - - - -	← verdeckte Basis
Segment 5	Stack für Prozess 1
(Stack Prozess 1)	
- - - - - - - - - - -	
- f r e i -	
- - - - - - - - - - -	
Segment 4	
(Code Prozess 2)	
- - - - - - - - - - -	
- f r e i -	
- - - - - - - - - - -	← verdeckte Basis
Segment 3	Heap für Prozess 1
(Heap Prozess 1)	

Abbildung 4.19: Speicherbelegung bei der Segmentierung mit verdeckter Basis

Segmente können sich zur Laufzeit an ihrem Ende ausdehnen und verkürzen. Falls eine Ausdehnung aufgrund belegten Speichers nicht möglich ist, muss das Segment umkopiert werden.

Allgemeine Eigenschaften der Segmentierung:

Relocation: Beim Laden nicht erforderlich. Beim Binden je nach Organisation (s.u.)

Auslagern: Auslagern und Einlagern ganzer Segmente mit Rückschreiben an beliebige, ausreichend große Lücke im realen Speicher möglich.

Verschieben: Möglich, da alle Adressen durch die verdeckte Basis automatisch angepasst werden.

Auswahl einer ausreichend großen Lücke mit dem First-Fit oder Best-Fit-Algorithmus (Suche der ersten bzw. der kleinsten passenden Lücke; siehe Abb. 4.20 a))

Speicherschutz: Durch Abfrage der Segmentlänge.

Untere Grenze = 0;

Read/Write/Execute-Modi

Nachteile:

- **Externe** Fragmentierung (große, ungenutzte Lücken)
- Ggf. Kompaktierung (Beseitigung ungenutzter Lücken) im realen Adressraum notwendig (siehe Abb. 4.20 b); zeitintensiv!).
- Bei Ausdehnung ggf. Umspeichern erforderlich

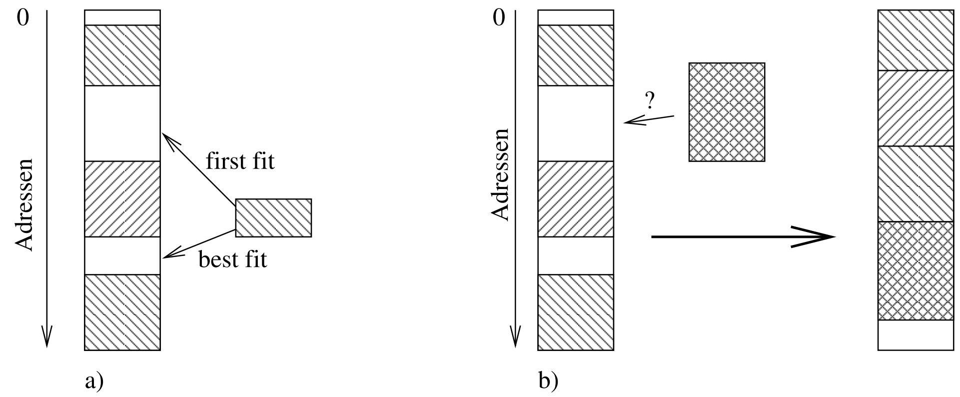


Abbildung 4.20: a) Wahl einer Lücke; b) Kompaktierung

Es gibt zwei Kriterien, nach denen Ansätze zur Segmentierung klassifiziert werden können:

1. Speicherung der Segmentnummern

   (a) Die Segmentbezeichner sind in separaten Segmentregistern enthalten. Die aktuellen Segmentnummern werden für Code, Stack und Heap über Extra-Befehle gesetzt. Es liegt eine echte 2-dimensionale Adressierung vor. Offset und Segmentnamen haben separate Wertebereiche.

   Das o.a. Problem des *sharings* kann bei Verfügbarkeit von Segmentregistern vermieden werden. In dem o.a. Beispiel könnte man im Prozess A Xlib das Segment 3 und im Prozess B das Segment 6 zuordnen. Vor einer Verzweigung an Xlib würde man das Codesegment-Register mit 3 bzw. 6 laden. Während der Ausführung von Xlib würden diese Segmentnummern über die Adressabbildungstafeln korrekt auf den Speicherbereich von Xlib abgebildet werden[5].

   (b) Die Segmentbezeichner sind Teil der in Adressteilen und in einzelnen Speicherzellen gespeicherten Adressen.

   Eine unmittelbare Konsequenz ist, dass Sharing dieselben Probleme bereitet wie bei der Seitenadressierung.

2. Gültigkeitsbereich der Segmentbezeichner

   (a) Die Segmentbezeichner sind lokal innerhalb eines Prozesses eindeutig.

[5]Interessanterweise besitzt die Intel x86er-Architektur Segmentregister. Das Problem der Absprachen über die Benutzung von Adressbereichen könnte also vermieden werden. Tatsächlich nutzt Windows 9x aber nur die Seitenadressierung und benötigt damit trotz vorhandener Segmentregister Absprachen über den Adressraum.

(b) Die Segmentbezeichner sind für ein ganzes Rechensystem oder sogar weltweit eindeutig (sog. globale Segmenttabelle).

Die Segmentbezeichner werden vom System verwaltet.

Relocating ist weder beim Binden noch beim Laden erforderlich, wenn getrennt übersetzte Programmteile zu getrennten Segmenten werden.

Einfaches Sharing von mehrfach benutzten Programmen (*shared library*) bzw. gemeinsam benutzte Daten: Bei Zugriff auf den gleichen (globalen) Namen erfolgt über die globale Segmenttafel automatisch ein Zugriff auf den gleichen physikalischen Speicher (realen Speicher).

Segmentnummern sollten auch beim Löschen von Segmenten nicht freigegeben werden (es gibt in der Regel noch Backups auf Bändern).

Segmente können mit Files identifiziert werden. Zugriffe auf Files können damit als Zugriffe auf z.Z. nicht geladene Segmente (Segmentation Fault) behandelt werden. Kopien der im Speicher befindlichen Segmente (ggf. nach Rückschreiben auf die Platte) als Files vorhanden. Abgebrochene Programme können daher relativ leicht wieder an der unterbrochenen Stelle fortgeführt werden.

Hierfür wird natürlich eine größere Anzahl von Bits zur Segmentbezeichnung benötigt. Daher wird diese Form der Segmentierung meist nur dann benutzt, wenn separate Segment-Register vorhanden sind.

Globale Segmentbezeichner lassen sich mit separaten Segmentregistern gut realisieren. Fehlen diese in einer Architektur, so ist dies kaum möglich und man realisiert meist nur lokale Segmentbezeichner. Dies belegt auch die folgende Tabelle 4.2.

	Segmentregister	Globale/lokale Segmentnamen
Lineare Segmentierung (IBM 360/370/390-Serie)	nein	lokal
PowerPC	nein	vermutl. lokal
segmented name space-Konzept bei Unix-Vorläufern Multics und Aegis	ja	global
Intel 80x86-Hardware (nicht in Windows 95 genutzt)	ja	lokal und global

Tabelle 4.2: Eigenschaften verschiedener Segmentierungs-Implementierungen

Die folgende Liste geht auf einige Details dieser Beispiele ein:

1. **Lineare Segmentierung**

   Die **Lineare Segmentierung** wurde und wird in den IBM-Rechnern der 360/370-Linie und dazu kompatiblen Systeme benutzt. Ein Grund dafür war sicherlich die beschränkte Anzahl von Adressbits. Jahrelang besaßen die erwähnten Rechner nur 24-Bit Adressen. Damit standen für Segmentnummern und Adressen innerhalb der Segmente nur wenige Bits zur Verfügung. Die folgende Tabelle zeigt diese Aufteilung:

Segmentnr.	Adresse innerhalb des Segments	
	Seitennr.	Offset innerhalb der Seite
6 Bit 14 Bit bei XA	6 Bit	12 Bit

   Zur Adressierung innerhalb eines Segments stand nur ein Adressraum von 256 kB zur Verfügung. Daher mussten logische Segmente (Heap, Stack, Code) ggf. in mehrere Segmente im Sinne des Betriebssystems aufgeteilt werden, d.h. es gab sog. **Überläufe** vom Offset in die Segmentnummern und somit keine echte 2-dimensionale Adressierung.

   Das Relocating beim Binden ist weiter erforderlich, da der Vorrat an Segmentbezeichnern zu klein ist, um jedem Unterprogramm einen eigenen Bezeichner zuzuordnen.

2. **Kombination von lokalen und globalen Segmenttabellen (Intel 80x86)**

   Zunächst zur Segmentierung bei dieser Prozessorlinie überhaupt. Einen Einstieg bieten die Verhältnisse beim 80286. Bei diesem Prozessor stehen Segmentregister für Code, Stack, Daten und **Extra-Daten** zur Verfügung. Über diese können Bereiche im Speicher erreicht werden (siehe Abb. 4.21).

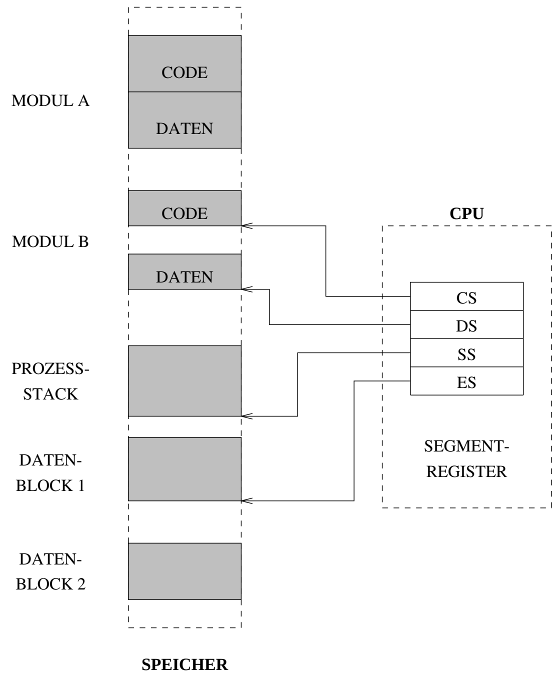


Abbildung 4.21: Segmentadressierung bei Intel-Prozessoren

Im *protected mode* zeigen die LDT- und GDT-Segmentregister auf Einträge in den lokalen und globalen Segmenttabellen, über die gemäß Abb. 4.22 die eigentlichen Daten- und Codebereiche erreicht werden können. Ein Bit im aktuellen Segmentregister entscheidet, ob eine Adressabbildung über die lokalen oder die globalen Tabellen erfolgt.

Da Windows 95 die Möglichkeiten der Segmentregister nicht nutzt und eine reine Seitenadressierung realisiert, sind für dieses Betriebssystem Konventionen über virtuelle Adressen gemeinsamen Codes (sog. DLL-Dateien) notwendig.

4.2.4 Segmentadressierung mit Seitenadressierung

Die beschriebene Segmentadressierung besitzt Vorteile gegenüber der Seitenadressierung, z.B. bei Realisierung von *shared libraries*. Andererseits kommt es aufgrund der externen Fragmentierung zu ineffizient genutztem Speicher und evtl. zu Speicher-Kopieroperationen. Die Nutzung der Vorteile beider Verfahren ergibt sich durch deren Kombination: man teilt Segmente in eine Anzahl von Seiten und sieht für jedes Segment eine eigene Abbildung von Seitennummern auf Kachelnummern vor (siehe Abb. 4.23).

Es ergeben sich die folgenden Vorteile:

- Die Fragmentierung reduziert sich auf die interne Fragmentierung.
- Kopieroperationen entfallen völlig.
- Shared libraries sind bei Vorhandensein von Segmentregistern ohne Absprachen über Adressräume zu realisieren.

4.3 Organisation der Adressabbildung

Zur Realisierung der Abbildung von virtuellen Adressen auf reale Adressen werden geeignete Datenstrukturen benötigt. Im Prinzip handelt es sich bei Seitennummern und Segmentnummern bzw. -namen um **Schlüssel**, für die Datenstrukturen aus der Vorlesung "Datenstrukturen" verwendbar sind. Man könnte etwa binäre Suchbäume verwenden, die als Schlüssel die Seitennummern und als zugeordnete Information

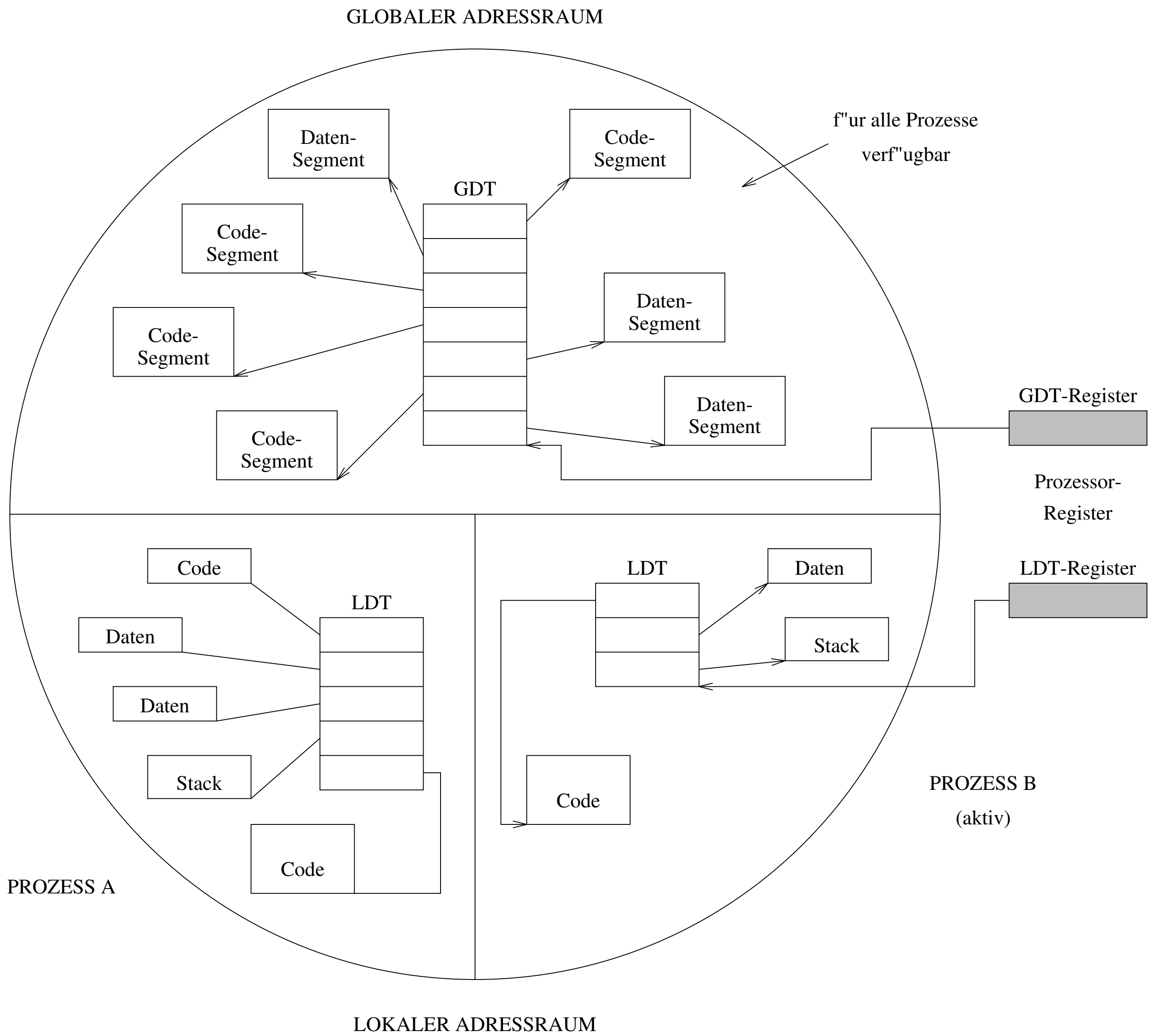


Abbildung 4.22: Globale und lokale Segmenttabellen (ab Intel 80286)

Segmentnr.	Seitennr.	Offset

Abbildung 4.23: Aufteilung von Segmenten in Seiten

die Kachelnummer enthalten. Allerdings muss die Adressabbildung schnell berechnet werden können und sie sollte meist so einfach sein, dass eine Realisierung im Mikroprogramm oder in Hardware möglich ist. Sie sollte daher die speziellen Gegebenheiten ausnutzen.

4.3.1 Zusammenhängende virtuelle Adressbereiche

Am einfachsten gestaltet sich die Abbildung, wenn die virtuellen Adressbereiche zusammenhängend sind oder nur kleine Lücken aufweisen. Dann kann man mit einer Tabelle arbeiten, auf die per Seiten- bzw. Segmentnummer zugegriffen wird. Die Abbildung 4.24 zeigt dies für die Seitenadressierung.

Ein Hardware-Register enthält die sog. Seitentafel-Basis. Dieses zeigt auf den Anfang der Tabelle des jeweils laufenden Prozesses. Das Register wird bei Prozesswechsel umgeladen.

Der indizierte Zugriff auf die Seitentabelle ist effizient, solange im virtuellen (!) Adressraum ein zusam-

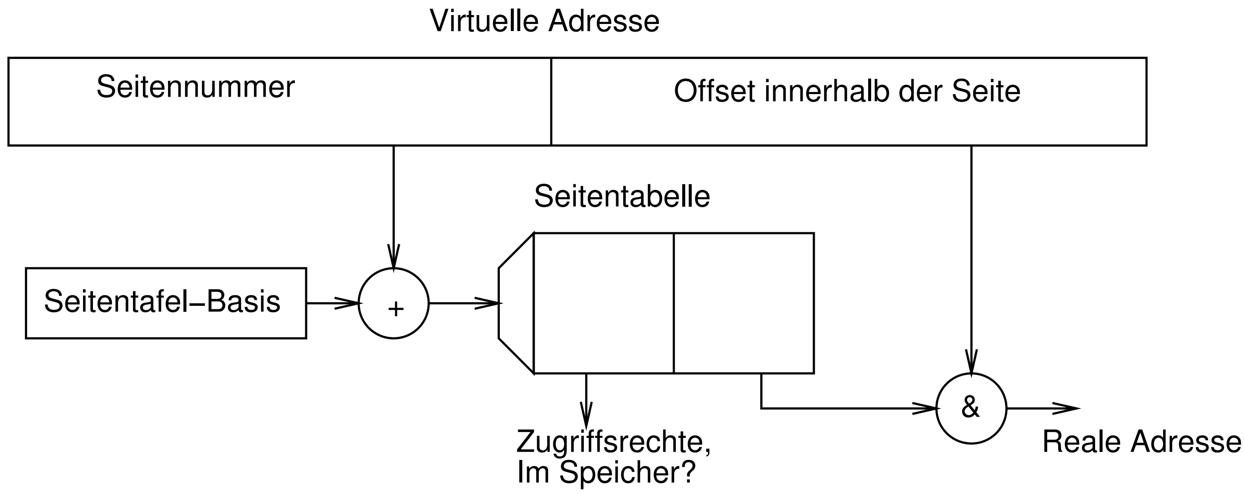


Abbildung 4.24: Prinzip der Adressabbildung bei der Seitenadressierung

menhängender Block belegt wird. Werden im virtuellen Adressraum größere Lücken nicht genutzt, so bleibt auch eine entsprechende Anzahl von Einträgen in der Seitentabelle ungenutzt. Man beachte die mögliche Größe einer Seitentabelle! Mögliche Auswege werden später beschrieben.

Im Falle der Segmentadressierung ist diese Form der Abbildung leicht zu modifizeren. Da die Größe von Segmenten nicht mehr unbedingt eine Zweierpotenz ist, muss die reale Adresse jetzt über eine Addition berechnet werden (siehe Abb. 4.25).

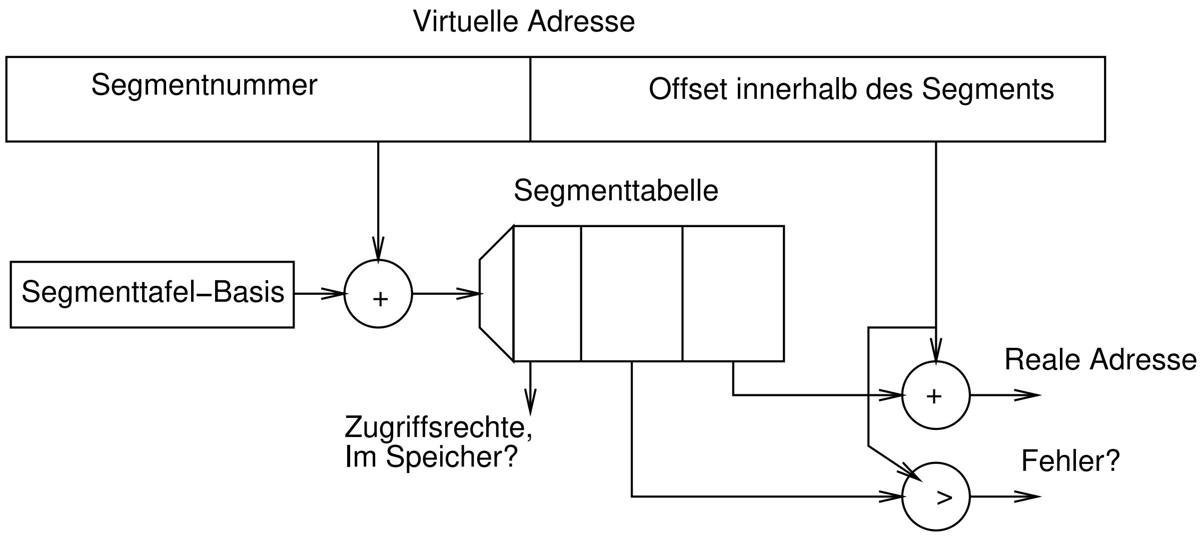


Abbildung 4.25: Prinzip der Adressabbildung bei der Segmentadressierung

Im Falle der Kombination von Segment- und Seitenadressierung ist die Abbildung zweistufig (siehe Abb. 4.26).

Nur die für die Abbildung **Segmentnummer → Kachelnummer** benötigte **Segmenttabelle** muss sich im Speicher befinden. Tabellen für die Abbildung der Seitennummern können ggf. nachgeladen werden.

Ebenso werden für nicht benötigte Segmente auch keine Seitentabellen mehr benötigt. Damit wird bei gewissen Lücken im verwendeten Adressraum bereits kein Speicher für die Seitentabelle mehr benötigt.

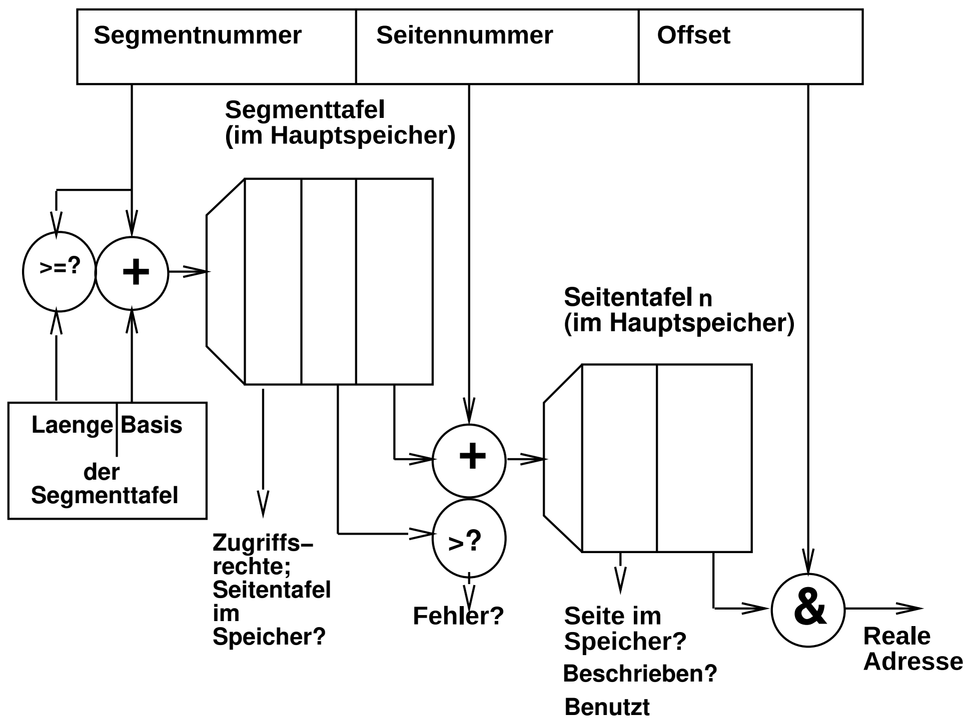


Abbildung 4.26: Segmentadressierung mit Seitenadressierung

4.3.2 Lücken im virtuellen Adressraum

Im allgemeinen wird man im virtuellen Adressraum Lücken haben, insbesondere bei der Seitenadressierung. Gründe für die Existenz solcher Lücken liegen z.B. in der Aufteilung zwischen Code-, Daten- und Heapbereichen. Zum Teil wird auch der Adressraum des Betriebsystems "eingeblendet´´, um Betriebssystemsaufrufe ohne Umschaltung zwischen virtuellen Adressräumen abwickeln zu können. So kennzeichnet häufig das oberste Adressbit, ob Adressen des Prozesses oder des Betriebssystems gemeint sind. Weiterhin sind manchmal Ein/Ausgabe-Geräte oder Grafikspeicher unter bestimmten Adressen direkt ansprechbar.

Man muss daher nach Verfahren suchen, die den für Seitentabellen benötigten Speicher möglichst klein halten.

Eine Möglichkeit dazu ist die dreistufige Umsetzung bei Intel-Prozessoren. Abb. 4.27 zeigt die Verhältnisse beim Intel 80386. Im Falle größerer ungenutzter Speicherbereiche kann im Seitentabellenverzeichnis kenntlich gemacht werden, dass zu einem Bereich von Seitennummern keine Seitentabellen existieren. Bei dieser Umsetzung wird auch dann Speicher für Seitentabellen eingespart, wenn die Segmentierung (wie bei Windows 95) nicht genutzt wird.

Zusätzlich gab es bei der Intel-Familie (wie auch vielfach sonst) das Bestreben, dass Tabellen möglichst selbst wieder auf eine Seite passen.

Der Bezug zu Datenstrukturen für die Suche von Schlüsseln (vgl. FIND-Operation der Datenstruktur-Vorlesung) wird noch deutlicher, wenn man die Datenstrukturen von Speicherverwaltungeinheiten wie der *memory management unit (MMU)* Motorola MC 68851 betrachtet. Diese ca. 1987 vorgestellte Coprozessor-Einheit zum Prozessor MC 68020 und nachfolgende Einheiten verwenden baumartig-organisierte Seitentabellen, die praktisch Vielweg-Suchbäume darstellen. Auf jeder Ebene des Baumes kann eine Verzweigung über einen wählbaren Teil der Seitennummer erfolgen (siehe Abb. 4.28). Es sind allerdings maximal vier Verzweigungen erlaubt (Baumtiefe 4 bzw. 5, je nach Definition). Dieses Verfahren vermeidet das Problem großer Seitentafeln bei Nutzung weit verstreuter Adressbereiche, da bei ungenutzten Adressbereichen keine Verweise auf Seitentabellen erfolgen müssen.

Im konkreten Fall gibt es drei verschiedene Wurzeln (Betriebsystem, User, DMA).

Da die Speicherverwaltungseinheit für diese Datenstruktur fest programmiert ist, ist auch diese Datenstruk-

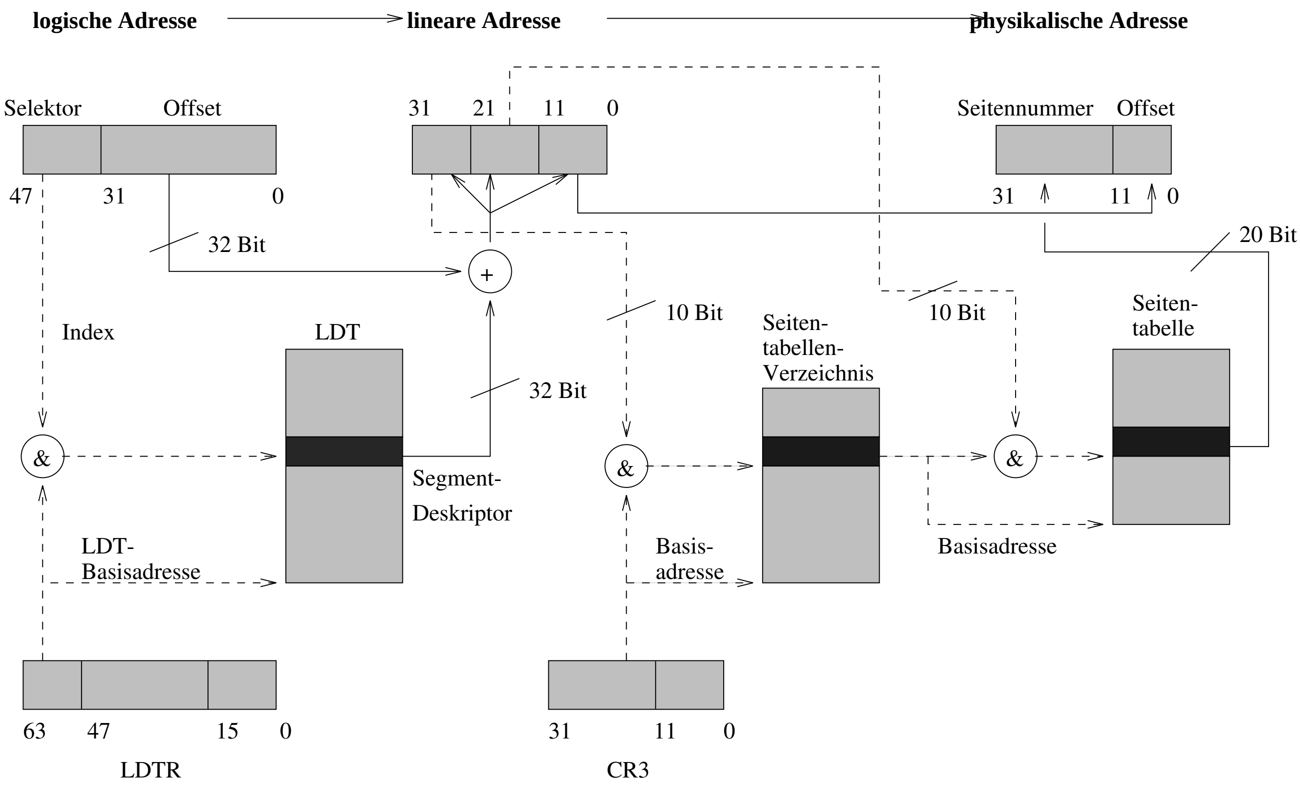


Abbildung 4.27: 3-stufige Adressumsetzung beim Intel 80386

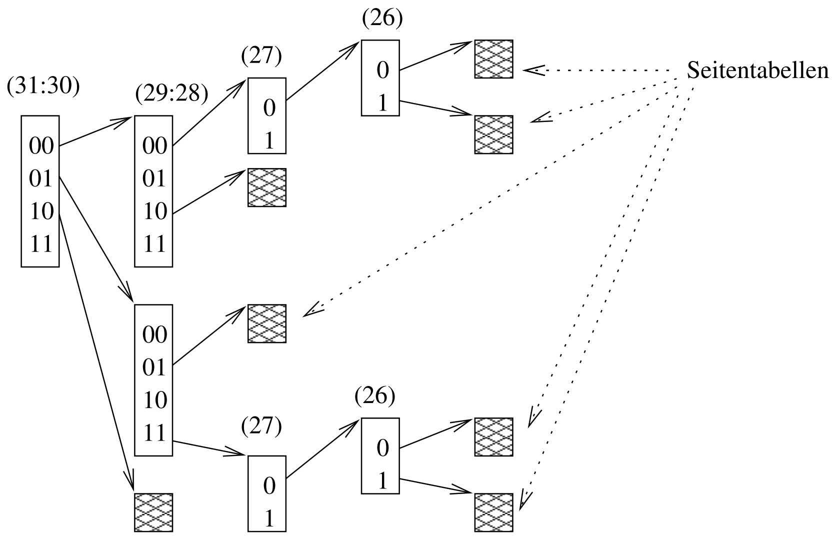


Abbildung 4.28: Vielweg-Bäume zur Abbildung auf reale Adressen

tur für diese Hardware verbindlich. Neuere Architekturen versuchen zum Teil, die häufigen Adressumsetzungen durch spezielle Puffer schnell durchzuführen und bei fehlendem Eintrag im Puffer einfach per Interrupt eine Routine des Betriebsystems zu rufen. Dieser Übergang von einer Behandlung von Seitenfehlern in Software statt in Hardware oder im Mikroprgamm bringt Flexibilität bei der Wahl der Datenstruktur und reduziert den Aufwand für Speicherverwaltung im Prozessor. Allerdings muss der Puffer dann eine große Anzahl von Einträgen enthalten, damit die Leistung nicht leidet.

4.4 Translation Look-Aside Buffer

Die bisherigen Überlegungen basieren auf der Realisierung der Abbildung mittels Tabellen im Hauptspeicher. Wegen der möglichen Größe der Tabellen müssen diese auch im Hauptspeicher untergebracht sein. Auf diese Weise können sie auch leicht auf Platten ausgelagert werden.

Allerdings ist diese Umsetzung über Tabellen im Hauptspeicher zu langsam (30 ns .. 60 ns /Zugriff) und zumindest für die häufigen Fälle ist eine Beschleunigung erforderlich. In Sonderfällen benutzt man hierzu die normalen Pufferspeicher (*caches*, siehe nächsten Abschnitt), die generell Speicherzugriffe beschleunigen sollen und daher auch Adressumsetzungen beschleunigen. Allerdings würden Adressumrechnungen diese Speicher bereits zum gutem Teil auslasten und man zieht daher meist spezielle Puffer zur Adressumrechnung vor. Man nennt diese Puffer *translation look-aside buffer* (TLB) oder auch *address translation memory*. Sie haben eine Verzögerung von 5ns bis 30ns pro Zugriff und können leichter als allgemeine Caches oder Speicher in die Fließbandverarbeitung einbezogen werden.

Im folgenden orientieren wir uns an der Seitenadressierung. Es gibt für TLBs drei Organisationsformen: *direct mapping, set associative mapping* und *associative mapping*.

4.4.1 Direct Mapping

In diesem Fall adressiert die Seitennummer oder ein Teil davon einen kleinen, schnellen Speicher.

Überwiegend liegt dabei eine Situation vor, in der der Bereich möglicher Seitennummern so groß ist, dass nur ein Teil der Seitennummer zur Adressierung verwendet werden kann. Diese Situation zeigt die Abbildung 4.29.

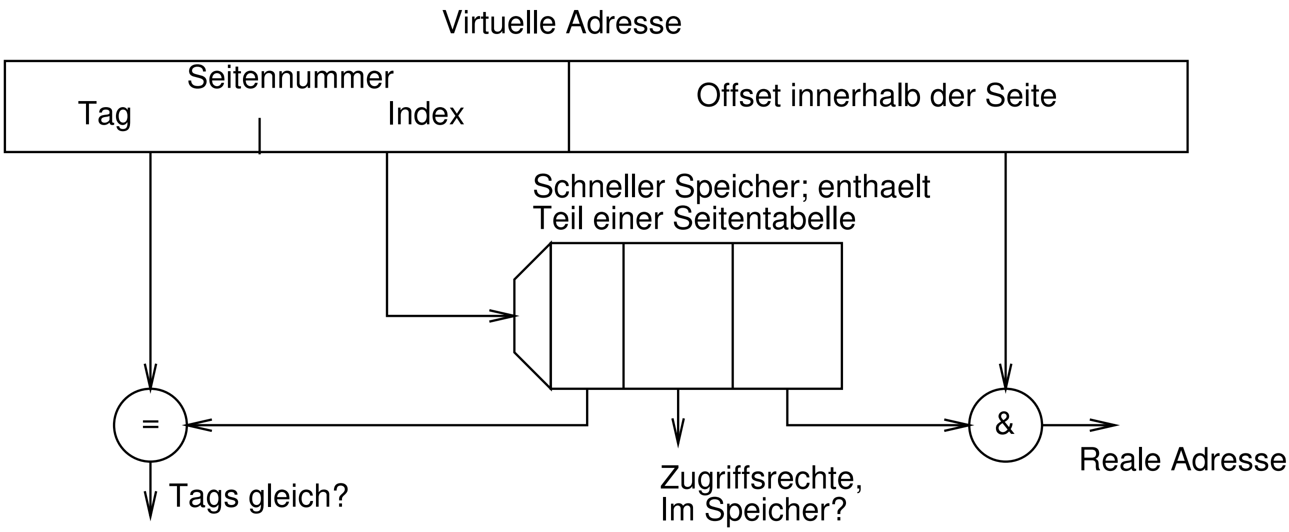


Abbildung 4.29: Direct Mapping für übliche Seitentafel-Größen

Zunächst erfolgt eine Adressierung des kleinen Speichers mit dem **Index**-Teil der Seitennummer, einem Teil weniger signifikanter Adressbits. Dann erfolgt ein Vergleich der **Tag**-Bits aus dem signifikanteren Teil der Seitennummer. Bei Gleichheit enthält die Zelle die richtige Basis. Beu Ungleichheit enthält der Speicher einen Eintrag für eine virtuelle Seitennummer, welche in den weniger signifikanten Bits mit der gesuchten übereinstimmt, in den signifikanteren aber nicht. Es gibt also jeweils eine gewisse Menge von Seiten, die demselbem TLB-Eintrag zugeordnet werden (siehe Abb. 4.30).

In diesem Fall erfolgt die Umsetzung über den Hauptspeicher. Nachteilig ist, dass bei abwechselndem Zugriff auf deratige kongruente Seiten jede Umsetzung über den Hauptspeicher erfolgen muss. Dies kann für gewisse Programme zu drastischen Performance-Einbußen führen.

Andererseits liegt ein Vorteil dieser Organisationsform in der Verwendbarkeit von Standard-RAM-Speichern. Durch die Aufteilung der Seitennummer sind große Adressräume bei kleinem Speicher (z.B. 64 Zellen) möglich

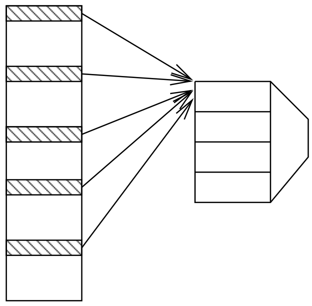


Abbildung 4.30: Seiten, die demselben TLB-Eintrag zugeordnet werden

und die Zeiten für das Umladen bei Prozesswechseln bleiben moderat.

Bei kleinen Adressräumen sind die beiden folgenden Sonderfälle des *direct mapping* möglich:

1. Der kleine, schnelle Speicher enthält **eine vollständige** Seitentafel.

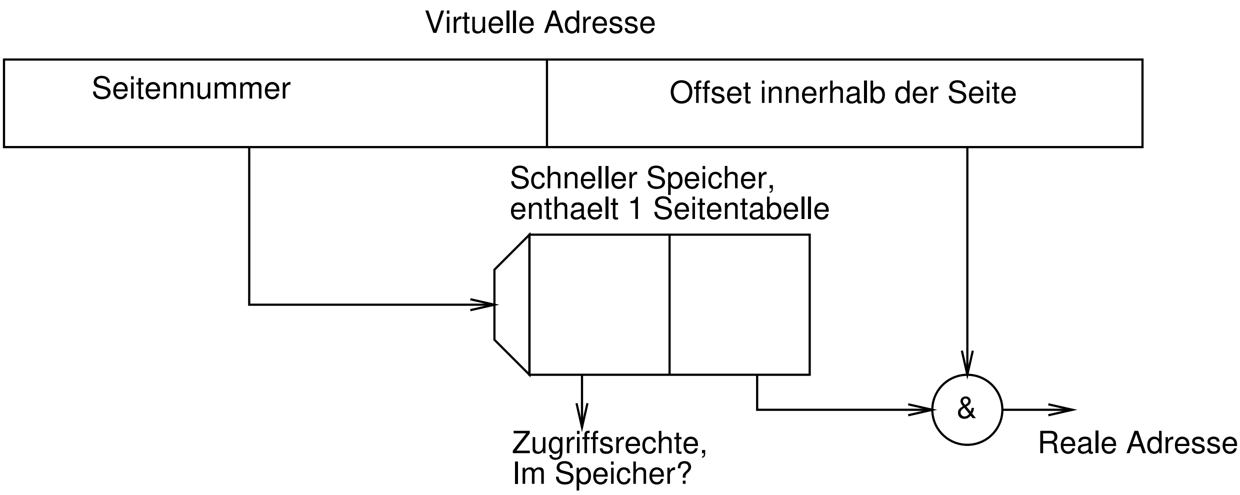


Abbildung 4.31: Direct Mapping für kleine Seitentafeln

Nachteile:

Größe des Speichers = Zahl möglicher Seiten.

Bei großem Adressraum werden große Speicher benötigt, z.B.:
(32-Bit-Adressen, 1 KB/Seite
→ 4 Mill. Seiten
→ bei 4 Byte/Eintrag: 16 MB Speicher)

Große Umladezeiten bei Prozesswechsel.

→ Umladen bei BS-Aufrufen nicht möglich;

Alternativen:

(a) Das BS adressiert real, ohne Speicherschutz, ohne Demand Paging;
Beispiel: Krupp Prozessrechner EPR 1300 (1980)

(b) Adressraum des BS und des Benutzerprozesses sind Teil eines virtuellen Adressraums
Beispiel: VAX: Bit 31 und 30 der virt. Adresse dienen der Unterscheidung zw. Benutzer- und BS-Teil des Adressraums.

2. Der kleine Speicher enthält Seitentafeln für **mehrere** Prozesse.

Vorteile:

Einfach, schnelle Umsetzung, Standard-RAMs .

BS kann virtuelle Adressierung nutzen, indem (zumindest Teile des BS) als Prozess ausgeführt werden.

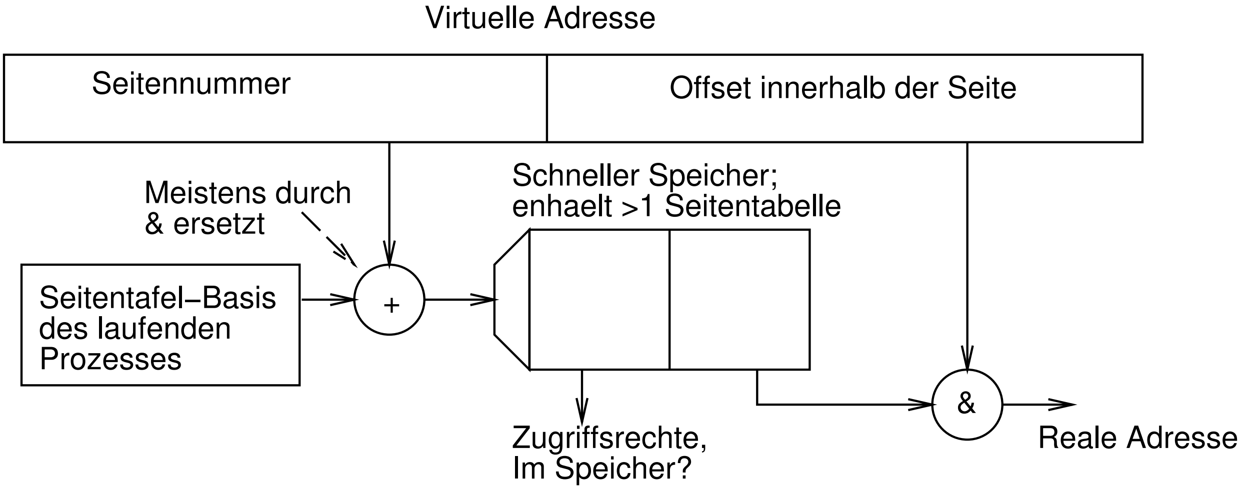


Abbildung 4.32: Direct Mapping mit beschleunigter Prozessumschaltung

	Zuordnung von Tabellennr. zu Prozessen höherer Priorität fest.
	Schneller Prozesswechsel bei Prozessen hoher Priorität.
Nachteile:	Noch größerer Speicher;
	langsamer Prozesswechsel für Prozesse ohne eigene Tabellennr.

Aufgrund der Konflikte in Abb. 4.30 hätte man gern mehrere **Tag**-Einträge und würde gern parallel suchen, ob der passende Eintrag dabei ist.

4.4.2 Mengen-assoziative Abbildung

Dies wird bei der **Mengen-assoziative Abbildung** (engl. *set associative mapping*) realisiert (siehe Abb. 4.33).

Mit einem Teil der Seitennummer wird adressiert. Unter den adressierten Einträge wird parallel nach einem Eintrag gesucht, der im verbleibenden Teil der Seitennummer übereinstimmt.

Bei Prozesswechsel werden die Einträge als ungültig erklärt.

Falls kein Treffer, Umsetzung über Hauptspeicher

Vorteile:	Noch relativ einfach zu realisieren. Weitgehende Freiheit von Anomalien des *direct mapping*.

Im Fall $||Index|| = 0$ entsteht der (voll) assoziative Speicher.

4.4.3 Assoziativspeicher, *associative mapping*

Benutzung eines **inhaltsadressierbaren Speichers** (engl. *content addressable memory* (CAM)).

Gleichzeitige Suche nach der Seitennummer für alle Einträge des TLB:

Falls kein Treffer: Umsetzung über den Hauptspeicher; Eintrag der Seite in eine der Zellen des CAM. Auswahl der Zelle gemäß Abschnitt 4.7. Löschen bei Prozesswechsel.

Vorteile:	Keine Abhängigkeit von bestimmten Folgen von Seitenreferenzen (i. Ggs. zum *direct mapping*)
	Gute Trefferrate selbst bei kleinem Speicher und großem Adressraum, wenn sich der Prozess lokal verhält (Angaben von 98 % Treffern)

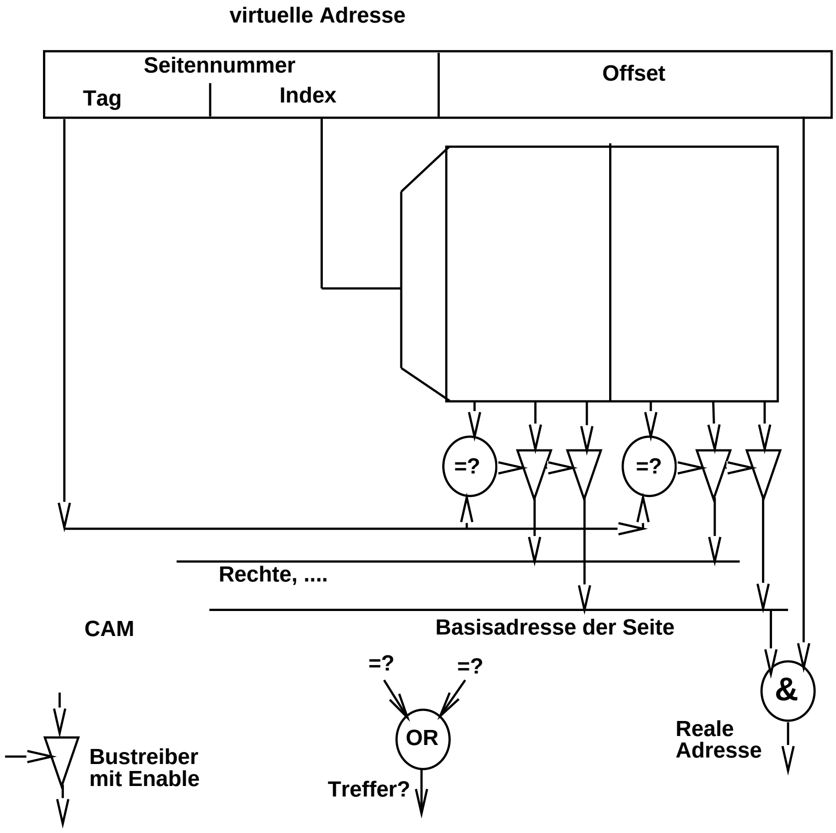


Abbildung 4.33: Mengen-assoziative Adressabbildung (Setgröße=2)

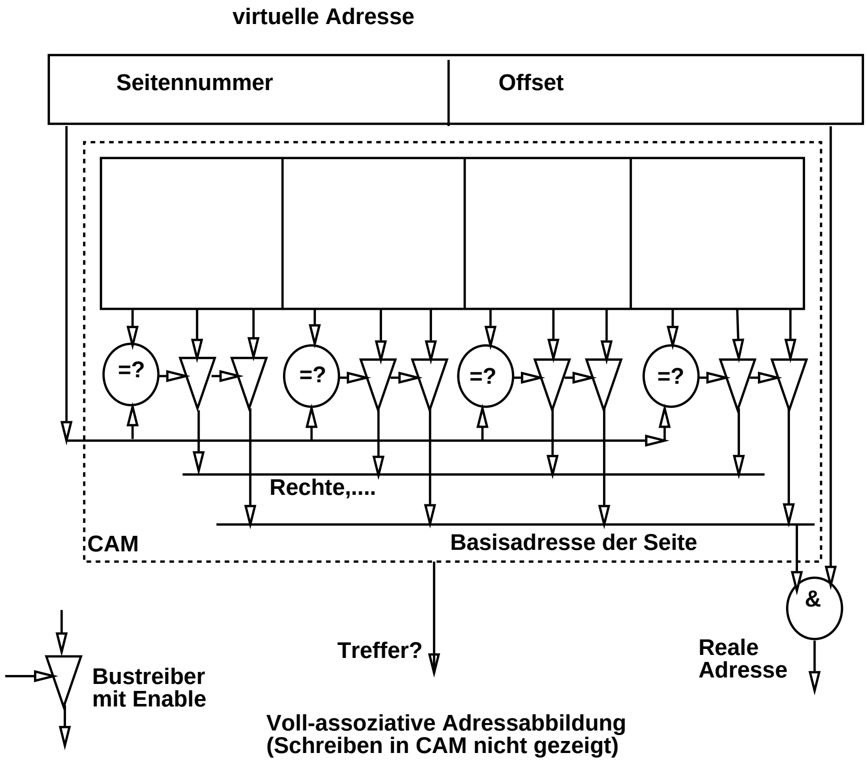


Abbildung 4.34: (Voll)-assoziative Adressabbildung

Nachteil: Wegen des großen Aufwands nur geringe Größe des CAM. Durch VLSI-Technik leichter realisierbar.

<u>Realisierung: Memory Management Unit (MMU) Motorola MC 68851</u>

Eigenschaften der MMU MC 68851:

- Voll-Assoziativer TLB mit 64 Einträgen
  Umsetzung über den Speicher, falls kein Treffer im TLB
- Baumartig-organisierte Seitentabelle (s.o.).
- 3 Bit Prozess- bzw. Adressraum-Identifikation (PID).

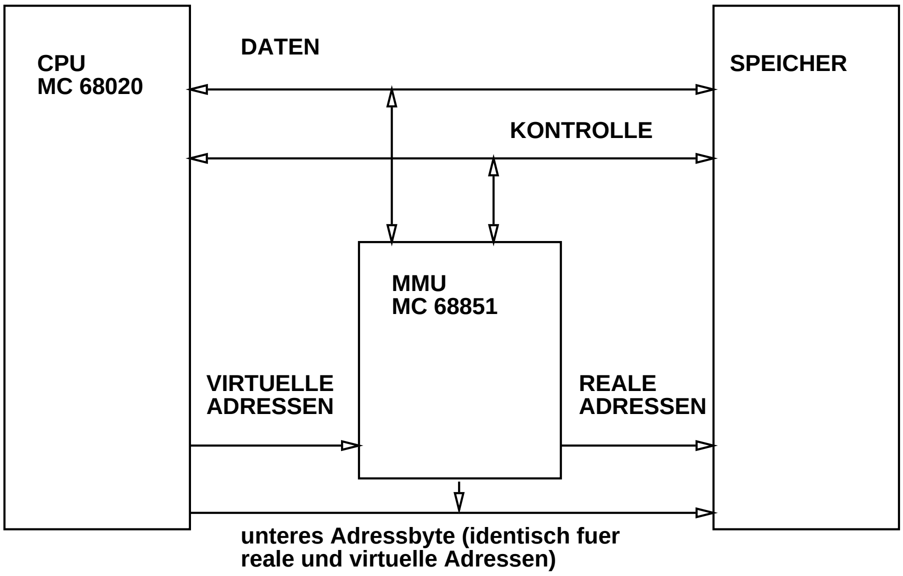


Abbildung 4.35: Adressabbildung mittels MMU

Least Significant Byte (Offset) gemeinsam.

4.5 Caches

4.5.1 Begriffe

Ein **Cache** ist ein Speicher, der vor einen größeren, langsameren Speicher M geschaltet wird, um die Mehrzahl der Zugriffe auf M zu beschleunigen. Zu diesem Zweck enthält der Cache einen Ausschnitt häufig benötigter Daten aus M. Im weiteren Sinn handelt es sich beim Cache also um einen Puffer zur Aufnahme häufig benötigter Daten. Im engeren Sinn handelt es sich um einen Puffer zwischen Hauptspeicher und Prozessor:

$\boxed{\text{CPU}}$ ———— $\boxed{\text{Cache}}$ ———— $\boxed{\text{Haupt-Speicher}}$

Das Wort **Cache** stammt vom Französischen *cacher* (verstecken), da Caches für Programmierer normalerweise nicht explizit sichtbar sind.

Organisation von Caches (im engeren Sinn):

Es wird stets zunächst geprüft, ob die benötigten Daten im Cache vorhanden sind. Dies geschieht anhand der (virtuellen oder realen) Adressen. Falls die Daten im Cache nicht enthalten sind, so werden sie aus dem Hauptspeicher (bzw. der nächsten Stufe einer Cache-Hierarchie) geladen.

Im Detail ist der Ablauf im Allgemeinen wie folgt:

- Zur Prüfung der Cache-Einträge wird zunächst mit einem Teil der Adresse, dem sog. Index-Teil, eine Zeile (engl. *line*) eines Caches adressiert. Anschließend muss der Tag-Teil der Adresse überprüft werden, um festzustellen, ob die Cachezeile auch tatsächlich zu der Adresse gehört. Dies geschieht anhand der **Tag-Bits** im Cache (siehe auch Abb. 4.36) . Eine Cachezeile kann wiederum noch aus Blöcken bestehen, die nicht alle gültige Inhalte enthalten müssen. Anhand von Gültigkeitsbits muss daher noch überprüft werden, ob die benötigten Daten auch zu einem gültigen Block gehören.

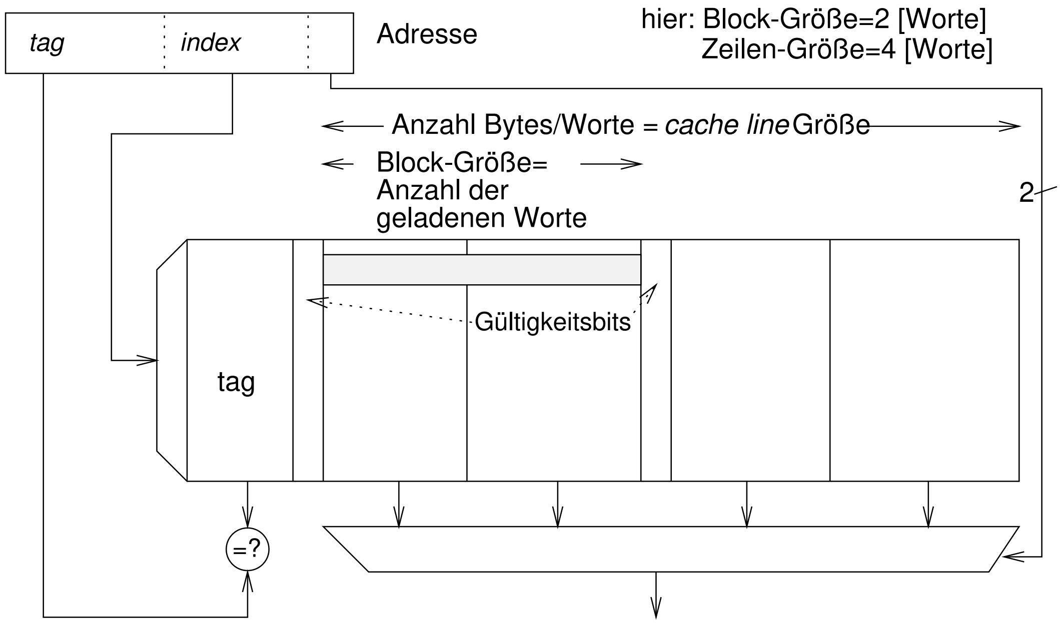


Abbildung 4.36: Allgemeine Organisation eines Caches

- Sind die Daten im Cache nicht vorhanden (sog. *cache miss*), so erfolgt ein Zugriff auf den Hauptspeicher. Die gelesenen Daten werden in den Cache eingetragen.

- Hierzu wird jeweils ein Block eingelesen (2..8 Worte). Unter der **Block-Größe** verstehen wir die Größe des nachgeladenen Bereichs (z.B. 4 Byte, auch wenn nur ein Byte angefordert wurde).

  Die Verbindung Speicher/Cache ist so entworfen, dass der Speicher durch das zusätzliche Lesen nicht langsamer wird. Methoden dazu:

  1. schnelles Lesen aufeinanderfolgender Adressen (nibble mode, block mode der Speicher)
  2. Interleaving, d.h. aufeinanderfolgende Adressen sind in verschiedenen Speicherbänken, Speicher-ICs etc. untergebracht.
  3. Block-Mode beim Bus-Zugriff, **Page-Mode** des Speichers
  4. Pipelining, z.B. bei Verwendung von EDO-RAMs oder SDRAMs (synchroner dynamischer RAMs)
  5. breite Speicherbusse, die mehrere Worte parallel übertragen können (64 Bit bei der SPARC 10)

  Stets wird zunächst das benötigte Datum, danach die Umgebung gelesen.

Würde man als Größe einer Zeile immer ein Wort nehmen, dann würde man bei fester Kapazität des Caches viel Platz zur Speicherung der Tag-Bits verwenden; größere Zeilengrößen reduzieren diesen Overhead.

Im Prinzip sind beliebige Relationen zwischen der Block- und der Zeilengröße möglich: die Abb. 4.36 zeigt den Fall Blockgröße < Zeilengröße. Vielfach ist die Blockgröße gleich der Zeilengröße; dann können die Gültigkeitsbits evtl. entfallen. Auch der Fall Blockgröße > Zeilengröße ist möglich: dann werden im Falle eines *cache-misses* mehrere Zeilen nachgeladen.

In der Regel muss es möglich sein, gewisse Adressen vom Caching auszunehmen (z.B. Adressen für I/O-Geräte). Hierzu kann man in der Seitentabelle ein *non-cacheable*-Bit vorsehen.

Übungsaufgabe:

Wie groß ist die mittlere Zugriffszeit als Funktion der Trefferrate (%) und der beiden Speicherzugriffszeiten?

Implizit haben wir bislang *direct mapping* als Cache-Organisation benutzt. Die Besonderheiten der Übertragung der Organisationsformen von TLBs auf Caches zeigt die folgende Liste:

- *direct mapping*

  Der Vergleich mit `Tag` erfolgt stets, da der Cache kleiner als der Hauptspeicher ist.

  Für Befehlscaches ist *direct mapping* besonders sinnvoll, weil aufeinanderfolgende Zugriffe mit unterschiedlichem Tag unwahrscheinlich sind.

- *set associative mapping*

  Wegen der Probleme des *direct mapping* mit Zugriffen mit gleichem Index der Adresse ist *set associative mapping* die häufigste Cache-Organisation. Set-Größe häufig 2 oder 4. Auswahl des zu überschreibenden Eintrags nach der LRU-Regel (siehe 4.7).

- Associative Mapping

  Wegen der Größe des Caches (z.B. 64 KB) kommt diese Organisation für Caches kaum in Frage. Ausnahme: Sektororganisation (ältere IBM-Rechner): wenige parallele Vergleicher für große Datenblöcke, die nicht vollständig nachgeladen werden.

4.5.2 Virtuelle und reale Caches

Caches können mit virtuellen und mit realen Adressen (sog. **virtuelle und reale Caches**) arbeiten. Die unterschiedliche Anordnung im System zeigt die Abb. 4.37.

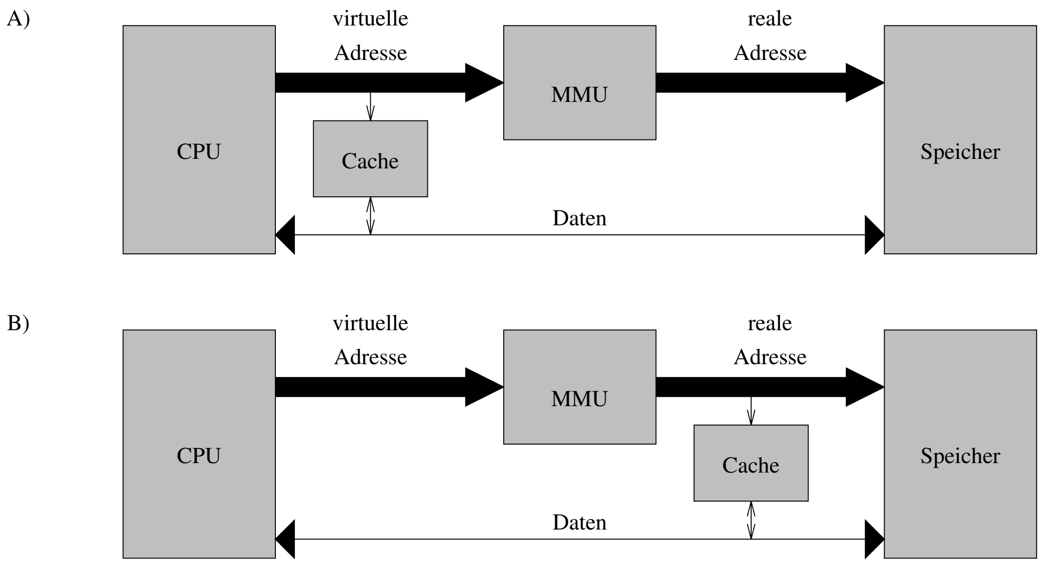


Abbildung 4.37: Anordnungsmöglichkeiten von Caches

Abhängig von der Anordnung ergeben sich unterschiedliche Geschwindigkeiten: virtuelle Caches sind meist schneller als reale. Die unterschiedliche Geschwindigkeit kann gut über ein **Petri-Netz-Modell** erklärt werden. Bei virtuellen Caches sind die Zugriffe auf den Cache und auf den TLB voneinander nicht kausal abhängig, können also nebenläufig ausgeführt werden (siehe Abb. 4.38).

Bei realen Caches muss zunächst der Zugriff auf den TLB ausgeführt werden, wie dies in Abb. 4.39 zu sehen ist.

Im Gegensatz zu älteren SPARC-Systemen benutzt die Fa. Sun bei der neuen SPARC 10-Workstation (1992) reale Caches. Um dennoch eine große Geschwindigkeit zu erreichen, hat die Fa. Sun einen Trick realisiert: beim Zugriff auf den realen Cache wird zwischen der Adressierung mittels Index und dem Tag-Vergleich unterschieden. Sofern der Index-Teil bei realer und virtueller Adresse übereinstimmt, kann nebenläufig auf

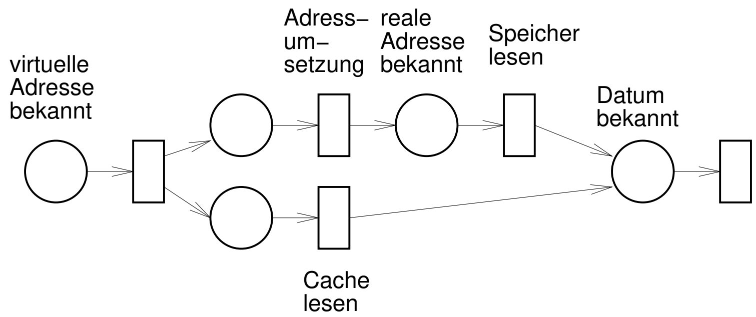


Abbildung 4.38: Kausale Abhängigkeiten bei virtuellen Caches

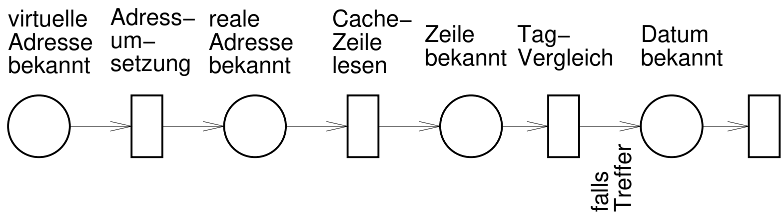


Abbildung 4.39: Kausale Abhängigkeiten bei realen Caches

den Cache über den Index sowie auf den TLB zugegriffen werden. Erst zum Tag-Vergleich muss die Kachelnummer bekannt sein (siehe Abb. 4.40).

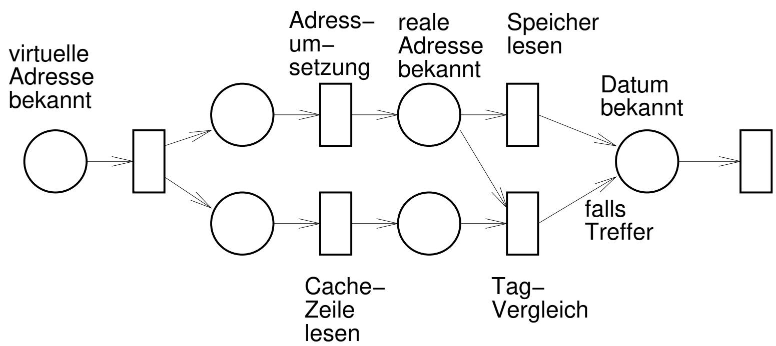


Abbildung 4.40: Kausale Abhängigkeiten bei der SPARC 10

Die Übereinstimmung von virtueller und realer Adresse im Index-Teil bedeutet allerdings, dass de facto so adressiert wird, als würde der Index noch zum Offset gehören, d.h. als wären die Seiten viel größer. Entsprechend nimmt die interne Fragmentierung zu.

4.5.3 Schreibverfahren

Bezüglich des Rückschreibens in des Hauptspeicher gibt es verschiedene Strategien:

1. Write-Through

   Alle Schreiboperationen beschreiben auch den Haupt-Speicher. Rückschreiben entfällt. Anwendbar nur, wenn gilt: Häufigkeit des Schreibens $<<$ Häufigkeit des Lesens und bei geringen Unterschieden in den Zugriffszeiten.

2. Copy-Back, Write-Back, Conflicting Use Write-Back

   Im laufenden Betrieb wird nur in den Cache geschrieben. Erst vor dem Überschreiben im Cache erfolgt ein Rückschreiben in den Haupt-Speicher.

Wird in der Regel mit *dirty bit* kombiniert, welches angibt, ob überhaupt geschrieben wurde. Inhalt wird nur zurückgeschrieben, falls Bit gesetzt.

4.5.4 Cache-Kohärenz

4.5.4.1 Kohärenz zwischen Hauptspeicher und einem Cache-Speicher

Bei der Verwendung von Caches entsteht aufgrund der Duplizierung von Informationen (Daten oder Code) das Problem der Konsistenz bzw. Kohärenz.

Schreibvorgänge in den Hauptspeicher lassen sich bei realen Caches behandeln, indem man die Caches alle Datentransfers auf dem Bus beobachten lässt (sog. *bus snooping*, **Bus-Lauschen**). Beobachtet der Cache einen Schreibvorgang in eine Speicherzelle von deren Inhalt er eine Kopie enthält, so erklärt er seine entsprechenden Cache-*line* für ungültig. Beim Schreibvorgang kann es sich dabei um das Nachladen einer Seite von der Platte oder um das Schreiben durch einen anderen Prozessor in einem Mehrprozessor-System handeln.

Für Schreibvorgänge in den Cache gilt das folgende: Im Falle eines *copy-back*-Caches genügt es, vor dem Überschreiben von Hauptspeicher-Kacheln den Cache in den Hauptspeicher zu kopieren. Soll also eine Seite aus dem Speicher verdrängt werden, so muss sie zunächst einmal mit dem Cache-Inhalt aktualisiert werden. Man nennt dies **den Cache ausspülen** (engl. *cache flushing*). Dies kann z.B. beim Seitenfehler vom Betriebssystem aus angefordert werden. Im Befehlssatz muss dazu ein *cache-flush*-Befehl realisiert werden.

Der Inhalt virtueller Caches wird nach Prosswechseln ungültig, da der neue Prozess mit denselben Adressen ja in der Regel völlig andere Informationen meint. Man muss solche Caches also beim Prozesswechsel für ungültig erklären. Da Caches heute zum Teil recht groß sind und da Betriebssysteme wie UNIX viele Prozesswechsel haben können, geht dadurch evtl. viel nützliche Information verloren. Daher werden bei neueren virtuellen Caches die virtuellen Adressen um **Prozessidentifikatoren** ergänzt, die die virtuellen Adressen wieder eindeutig machen. Da evtl. mehrere Prozesse im gleichen Adressraum ausgeführt werden, kann man die Identifikatoren präziser auch **Adressraumidentifikatoren** nennen. Auch mit Adressraumidentifikatoren bleiben aber noch Probleme: im Falle des Sharings von Daten zwischen Prozessen mit unterschiedlichen Adressräumen würden die Prozesse eigene Kopien im Cache erhalten. Eine Notlösung besteht darin, gemeinsam benutzte Seiten über ein *non-cacheable*-Bit in der Seitentabelle vom Caching auszunehmen.

Eine andere, aufwendige Technik besteht darin, sowohl virtuelle als auch reale Tags zu benutzen. Der Cache besteht dann zwei entkoppelten Teilen: vom Prozessor her wird der Zugriff über virtuelle Tags realisiert, während gleichzeitig mit den realen Tags ein Bus-Lauschen durchgeführt wird.

Abb. 4.41 enthält einen Vergleich der Eigenschaften virtueller und realer Caches.

	Realer Cache	Virtueller Cache mit PIDs	Virtueller Cache
Inhalt nach Prozesswechsel	gültig	gültig, falls ausreichend PIDs	ungültig
Inhalt nach Seitenfehler	ungültig (teilweise)	gültig	gültig
Geschwindigkeit	langsam	schnell	schnell
automatische Konsistenz bei Sharing	ja	nein (!!)	ja, aber Cache muss neu geladen werden
primäre Anwend.	große Caches	kleine Caches; Befehlscaches, falls dynamisches Überschreiben von Befehlen verboten ist	

Abbildung 4.41: Vergleich von Caches

4.5.4.2 Systeme mit mehreren Caches

Viele moderne Rechnerarchitekturen bieten 2 Cache-Ebenen: einen interen Cache auf dem Prozessor-Chip (L1-Cache) sowie einen größeren, evtl. externen Cache (L2-Cache). Außerdem ist der interne Cache vielfach in separate Daten- und Befehlscaches aufgeteilt. wird man dise beiden Caches organisieren? Möglich ist z.B. folgende Wahl: der interne Befehls-Cache wird virtuell organisiert, sofern Befehle nicht dynamisch überschrieben werden dürfen. Eine Rückschreib-Technik ist damit überflüssig. Es ist möglich, dass der interne Befehlscache Befehle enthält, die im externen Cache inzwischen wieder verdrängt wurden. Ein Konflikt zwischen der Belegung von Zeilen des externen Caches mit Daten und Befehlen entfällt damit. Der interne Datencache ist real adressiert und verwendet eine *write-through*-Strategie. Dies ist möglich, weil die Geschwindigkeitsunterschiede zwischen beiden Caches kleiner sind als die zwischen externem Cache und Hauptspeicher. Der externe Cache (**L2-Cache**) ist real addressiert und verwendet *copy-back* und *bus snooping*. .

Wir werden auf das Thema Cache-Kohärenz im Kapitel 6 noch einmal zurückkommen.

4.6 Allgemeine Speicherhierarchie

Cache und Hauptspeicher sind Teil einer allgemeinen Speicherhierarchie mit sehr unterschiedlichen Zugriffszeiten. Während bei den schnelleren Stufen der Hierarchie die Hardware für das Prüfen auf Vorhandensein in einem lokalen Ausschnitt und den ggf. erforderlichen Austausch zuständig ist, erfolgt dies bei den langsameren Stufen per Software oder gar von Hand:

Name	Realisierung	Zugriffszeit (ca.)	Gültig-Prüfung	Austausch durch
Tertiärspeicher	Bänder	$3 * 10^1$ s	BS	Operateur
Sekundärspeicher	Platten	$1 * 10^{-2}$ s	BS	BS
(Platten-Cache)	DRAM	$1 * 10^{-?}$ s	BS	BS
Primärspeicher	DRAM	$5 * 10^{-8}$ s	HW	BS
Cache	SRAM	$1 * 10^{-8}$ s	HW	HW
Register	SRAM	$1 * 10^{-9}$ s	Compiler	Compiler

4.7 Austauschverfahren

Innerhalb der Speicherhierarchie ist es auf jeder Stufe erforderlich, Informationen in der schnelleren Stufe der Hierarchie auszutauschen, um häufig benötigter Information Platz zu machen. Es gibt sehr viele Verfahren zur Wahl der Information im schnellen Speicher, die verdrängt wird. Im folgenden werden drei anhand der Hierarchiestufe Hauptspeicher/Magnetplatte besprochen[6]:

1. **Random-, bzw. Zufallsverfahren**
   - Zufällige Auswahl der auszulagernden Kachel
   - lokal oder global, fest oder variabel
   - keine zusätzliche Hardware erforderlich
   - sinnvoll nur dann, wenn ohnehin keine Regularität im Zugriffsverhalten erkennbar ist; sonst ungeeignet
2. ***NRU, Not Recently Used***

   Dieses Verfahren basiert auf der Analyse von zwei Einträgen, die in den Seitentabellen üblicherweise existieren: dem *used bit* und dem *modified bit*. Aufgrund der Werte dieser beiden Bits kann man zwischen vier Klassen von Seiten unterscheiden:

   (a) Nicht benutzte, nicht modifizierte Seiten

   (b) Nicht benutzte, modifizierte Seiten

[6] Beschreibungen weiterer Verfahren findet man bei Baer [Bae80] sowie in vielen Büchern über Betriebssysteme (siehe z.B. Tanenbaum [Tan76]).

(c) benutzte, nicht modifizierte Seiten

(d) benutzte, modifizierte Seiten

Die Klasse 2 ist deshalb möglich, weil die Used-Bits periodisch mit Hilfe eines Timers zurückgesetzt werden und daher Seiten der Klasse 4 zu Seiten der Klasse 2 werden können. Das Modified-Bit kann nicht periodisch zurückgesetzt werden, weil modifizierte Seiten sonst nicht auf die Platte zurückgeschrieben werden würden.

Der NRU-Algorithmus überschreibt jetzt eine zufällig ausgewählte Seite der niedrigsten, nicht-leeren Klasse.

3. ***LRU = least recently used***

Es wird stets die Kachel überschrieben, die am längsten nicht benutzt wurde.

Lösungen zur Realisierung:

(a) verkettete Liste von Kachelnummern (Bei Baer als *stack* bezeichnet)

Es wird ständig eine verkettete Liste von Kachelnummern gehalten, die nach dem Alter des letzten Zugriffs sortiert ist.

Operationen auf dieser Datenstruktur:

- Zugriff auf eine Kachel i:
  i wird als Element am Anfang der Liste eingefügt und, falls in der Liste sonst noch vorhanden, dort entfernt.
- Seitenfehler:
  Die Kachel, die zu dem letzten Element der Liste gehört, wird dort entfernt

Beispiel:

Zugriff auf Seite	1	2	3	4	5	3	5	4	6
und damit auf Kachel	1	2	3	4	1	3	1	4	2
verkettete Liste von Kacheln (Anzahl der Kacheln = 4)	1	2 1	3 2 1	4 3 2 1	1 4 3 2	3 1 4 2	1 3 4 2	4 1 3 2	2 4 1 3

Einfügen am Anfang und Entfernen am Ende einer Liste lässt sich relativ einfach in $O(1)$ Zeiteinheiten realisieren, kann also durch die Hardware relativ leicht während der Zugriffe auf Seiten durchgeführt werden. Das Entfernen eines Elements ist nur dann hinreichend schnell zu realisieren, wenn es durch eine geeignete Datenstruktur unterstützt wird. Möglich ist z.B. das Führen eines *bucket arrays.* Zu jeder Kachel i könnte dieses Array einen Verweis auf den Platz der Kachel i innerhalb der Liste enthalten. Bei doppelt-verketteter Liste kann dann auch das Entfernen des Listenelements in $O(1)$ erfolgen.

(b) Dreiecksmatrix[7]

Für alle Paare (i, j) von Kacheln wird in einer Matrix die Relation "Der letzte Zugriff auf i ist älter als der letzte Zugriff auf j" gespeichert, also z.B.:

$$f[i,j] = \begin{cases} 1, & \text{falls der letzte Zugriff auf } i \text{ älter ist als der auf } j \\ 0, & \text{sonst} \end{cases}$$

Wegen der Antisymmetrie braucht lediglich eine Δ-Matrix realisiert zu werden.

Operationen auf dieser Datenstruktur:

- Zugriff auf eine Kachel i: i wird jüngste Kachel, d.h. es ist zu setzen:

$$\forall j \neq i : f[i,j] := 0$$

$$\forall j \neq i : f[j,i] := 1$$

- Seitenfehler:
  gesucht ist die älteste Kachel, d.h. die Kachel i, für die gilt:

$$\forall j \neq i : f[i,j] = 1 \wedge f[j,i] = 0$$

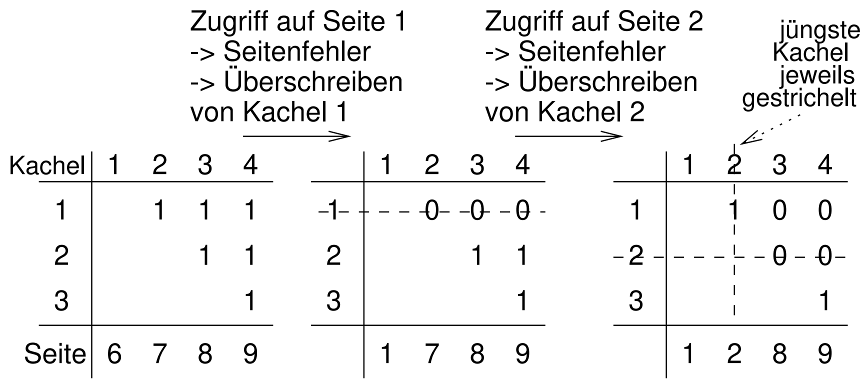


Abbildung 4.42: Benutzung einer Dreicksmatrix

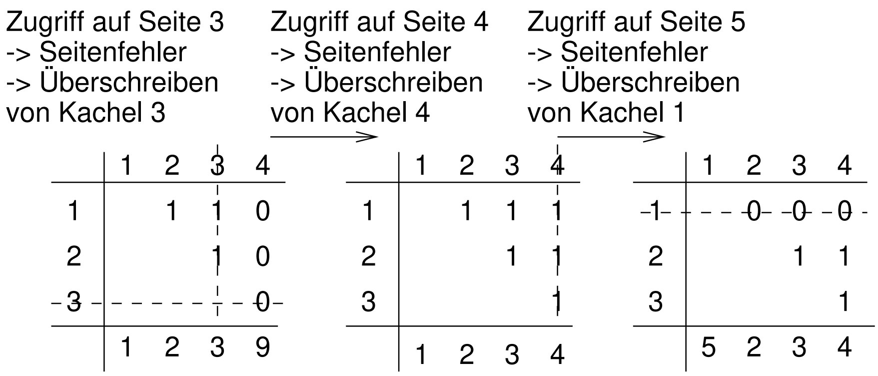


Abbildung 4.43: Benutzung einer Dreicksmatrix

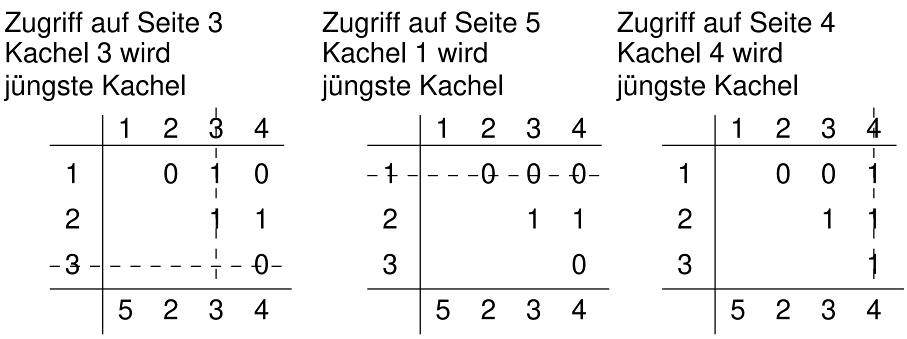


Abbildung 4.44: Benutzung einer Dreicksmatrix

Abbildungen 4.42 bis 4.44 zeigen ein Beispiel (1. Index=Spalte, 2. Index=Zeile).

Das zeilen- bzw. spaltenweise Setzen und Suchen nach 0 bzw. 1 kann durch eine geeignet Hardware schnell ausgeführt werden. Für eine größere Zahl von Kacheln ergibt sich aber ein erheblicher Aufwand.

Benutzt wird LRU in der reinen Form nur für wenige Einträge im lokalen Speicher, z.B. bei Caches oder TLBs nach dem *set associative*-Prinzip, dort vor allem für den Fall einer zweielementigen Menge (Δ-Matrix entartet zu einem Bit).

[7]Quelle: Liebig [Lie80]

Kapitel 5

Ein-/Ausgabe

Rechensysteme ohne Ein- und Ausgabemöglichkeiten[1] (E/A) wären sinnlos. Alle Rechensysteme enthalten daher Ein- und Ausgabesysteme. Ein- und Ausgabesysteme können z.B. über einen Bus (deutsch: "Datensammelschiene") von der CPU aus angesprochen werden. Zwischen dem Bus und den eigentlichen E/A-Geräten werden in der Regel spezielle Geräte-Steuerungen (engl. *controller*) benötigt. Diese Controller übernehmen eine Reihe von Geräte-spezifischen Aufgaben, deren Bearbeitung in der CPU diese zu stark belasten würden und evtl. gar nicht möglich wären.

In der Abb. 5.1 sind der Bus, die Steuerungen und die E/A-Geräte eines Rechensystems hervorgehoben.

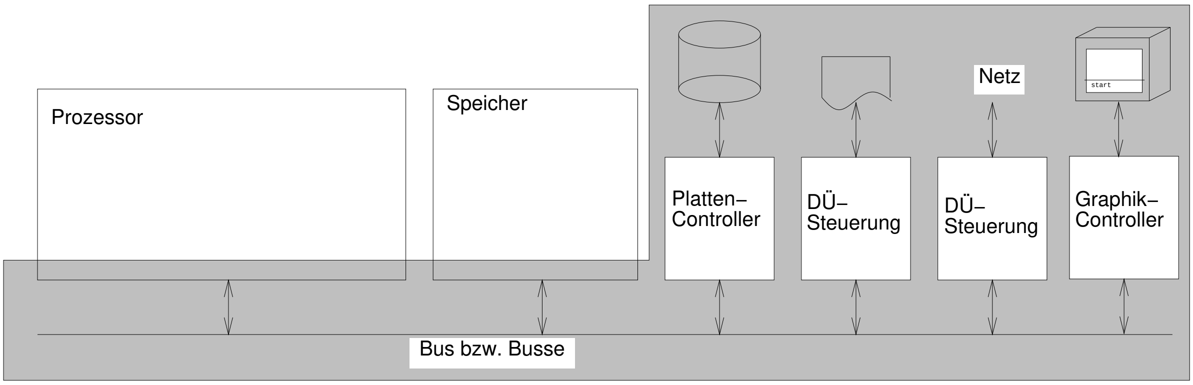


Abbildung 5.1: Rechner mit Ein-/Ausgabesystem

5.1 Bussysteme

Zunächst beschäftigen wir uns mit Bussen und deren Anschluß an die Controller. Die Bus-Organisation müssen wir uns im folgenden genauer ansehen.

5.1.1 Topologien

Aufgrund der jeweils vorhandenen Entwurfsbeschränkungen und Zielvorstellungen ist eine Vielzahl von Bus-Topologien im Gebrauch.

- In einer Reihe von Fällen findet man in der Praxis Bussysteme, die auch bei detaillierter Betrachtung weitgehend das Schema der Abb. 5.1 widerspiegeln. Ein Beispiel dafür ist der Bus der 68000er-Prozessorfamilie. Eine Ursache hierfür ist die begrenzte Anzahl von Pins, die an derartigen Prozessoren

[1] Die relativ dürftige Behandlung der Ein-/Ausgabe in den meisten Rechnerarchitektur-Büchern ist ein Grund dafür, dass dieser Vorlesung nicht einfach ein vorhandenes Buch zugrunde gelegt wurde.

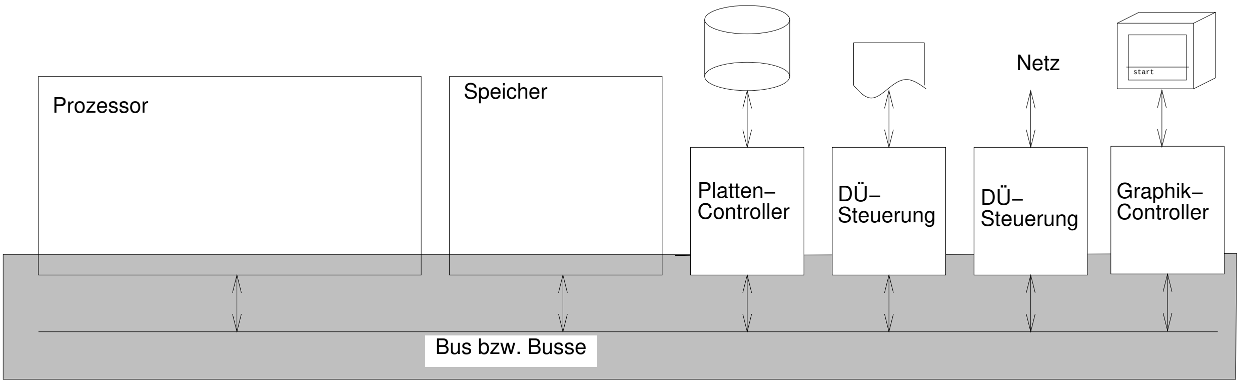


Abbildung 5.2: Zum Thema des Abschnitts 4.1

vorhanden ist. An diesem Bus werden bei einfachen Systemen sowohl der Speicher als auch Gerätesteuerungen direkt angeschlossen (siehe Abb. 5.3).

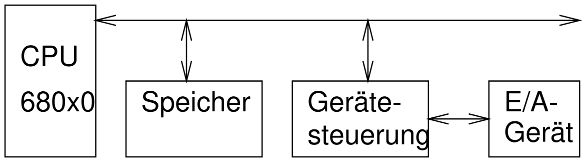


Abbildung 5.3: Bus der 68000er-Prozessorfamilie (vereinfacht)

Die in Abb. 5.3 gezeigte Busstruktur ist sehr einfach, schränkt aber die Möglichkeiten der gleichzeitigen Übertragung zwischen verschiedenen Geräten stark ein und bewirkt eine enge Kopplung der Spezifikationen für die Speicher- und die E/A-Schnittstelle.

- Einen extremen Gegensatz zu der Struktur in Abb. 5.3 finden wir bei Großrechnern (wie z.B. der IBM S/370-Serie). Bei derartigen Rechnern sind die Pfade stärker entkoppelt. Der Speicher ist als Multiport-Speicher ausgelegt, und es existieren spezielle E/A-Einheiten (die sog. **Kanäle**), welche ohne CPU-Belastung Daten zwischen Speicher und E/A-Steuerungen transportieren können und sogar einfache Programme abarbeiten können.

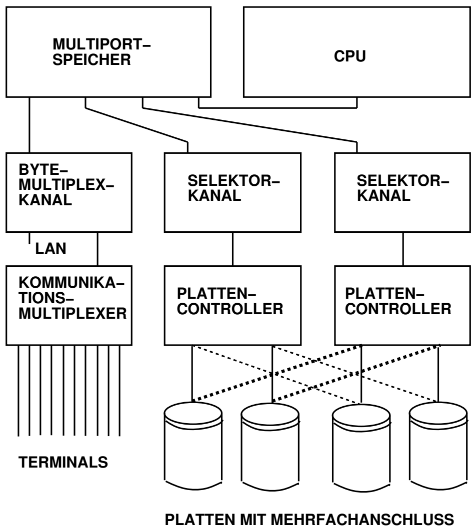


Abbildung 5.4: Beispiel der Anschlußorganisation eines kommerziellen Großrechners

- Systeme, die auf Mikroprozessoren basieren, gehen häufig einen anderen Weg zur Entkopplung von Speicher- und E/A-Bus. In der Regel wird hierzu ein sog. **Busadapater** bzw. eine **Bus-Brücke** realisiert (siehe Abb. 5.5).

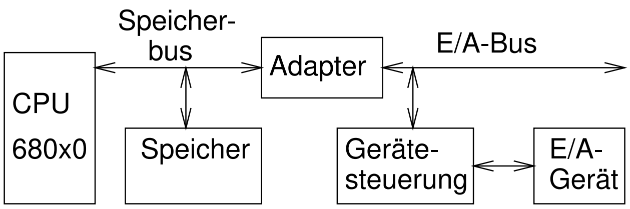


Abbildung 5.5: Separate Speicher- und E/A-Busse

In dieser Organisationsform werden Geräte entweder durch spezielle Speicheradressen oder anhand der Werte auf einer Kontroll-Leitung $IO/\overline{MEMORY}$ von Speicherzellen unterschieden.

Durch die Trennung von Speicher- und E/A-Bus kann für den Speicher eine höhere Geschwindigkeit erreicht werden. Beispielsweise, indem für den Speicherbus eine größere Wortbreite (z.B. 128 Bit) verwendet wird.

Die hier angegebene Trennung findet man in vielen Rechensystemen, so z.B. bei 80486-Systemen die Trennung in VESA local (VL-) Bus und ISA-Bus.

- Abb. 5.6 zeigt die Grundstruktur älterer PCI-basierter Systeme.

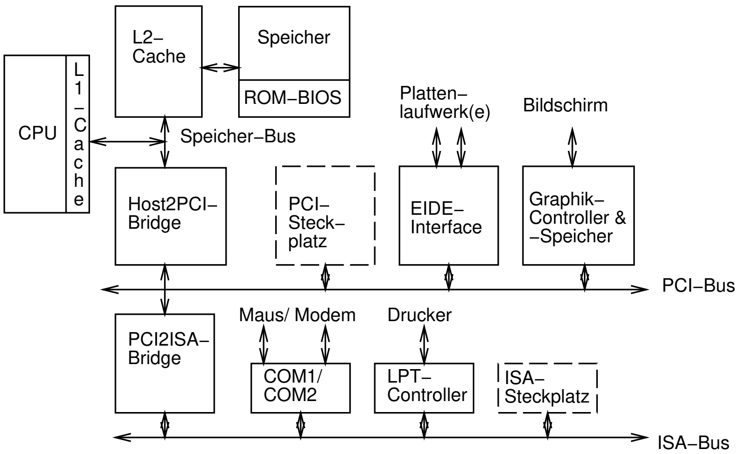


Abbildung 5.6: Blockschaltbild eines PCI-Systems

Der schnellste Bus innerhalb eines solchen Systems ist der Speicher-Bus, der häufig eine große Wortbreite besitzt. Danach kommt der PCI-Bus (*Peripheral Component Interconnect* -Bus). Der PCI-Bus ist für eine relativ kleine Anzahl von Steuerungen ausgelegt. Wollte man mehr Steuerungen an diesen Bus anschließen, so würde die Geschwindigkeit unter der kapazitiven Belastung stark leiden. Weitere Steuerungen müssen daher über weitere Brücken angeschlossen werden, wie z.B. über eine PCI-to-ISA- oder über eine PCI-to-PCI-Brücke. Die PCI-to-ISA-Brücke erlaubt dabei den Anschluß von Standard ISA-Karten, die in PCs schon lange verwendet werden. Mittels PCI-to-PCI-Brücken kann man die Anzahl anschließbarer Einheiten erhöhen und ein **hierarchisches Bussystem** aufbauen.

- Ein Trend bei PC-ähnlichen Systemen geht zu immer aufwendigeren Speicherzugängen. Damit wird der Tatsache Rechnung getragen, dass der Umfang der Parallelarbeit außerhalb der CPU wächst und beispielsweise Graphikkarten auch einen schnellen Zugriff auf den Speicher benötigen (der PCI-Bus ist hier bereits wieder ein Engpass).

5.1.2 Elektrotechnische Aspekte

Alle im Rahmen der Vorlesung Rechnerarchitektur benötigten Kenntnisse der elektrotechnischen Aspekte sollten eigentlich im Grundstudium vermittelt werden. Die Erfahrung zeigt, dass dies meist nicht geschieht (oder wieder vergessen worden ist). Wir behandeln hier daher sicherheitshalber die für Busse wichtigen Aspekte.

5.1.2.1 Bus-Treiber

Die meisten der Leitungen eines Busses können potentiell von verschiedenen Hardware-Einheiten erzeugt werden. Damit müssen mehrere Einheiten in der Lage sein, die entsprechenden Leitungen zu treiben. Um Konflikte durch die Beschaltung des Busses mit mehreren Ausgängen zu verhindern, müssen die Ausgänge von Einheiten fast alle abgeschaltet werden können. Abb. 5.7 zeigt die Standard-Ausgangsschaltung (sog. *totem-pole*-Schaltung, totem pole = Marterpfahl) üblicher Logikfamilien wie TTL oder CMOS. Die beiden Transistoren schalten komplementär. Damit liegt der Ausgang entweder auf '0' oder auf '1'. Ein Abschalten des Ausgangs ist nicht möglich.

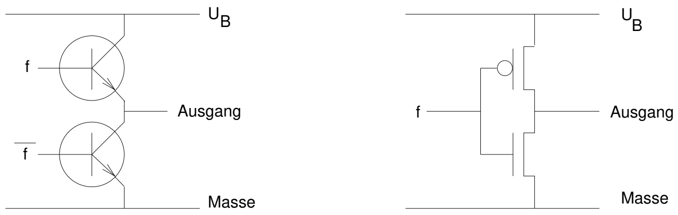


Abbildung 5.7: Standard-Ausgangsschaltungen

Es gibt zwei Schaltungstechniken, mit denen die Ausgänge abgeschaltet werden können.

1. **Open-Collector-Ausgänge**

   Bei dieser Schaltungstechnik wird der obere Transistor nicht innerhalb des Schaltungsausgangs realisiert. Damit liefert diese Technik am Ausgang zwei mögliche Werte: '0' und "hochohmig" ('Z')[2]. Allen Ausgängen gemeinsam wird ein einziger Widerstand zugeordnet, welcher auf der Busleitung eine '1' erzeugt, falls alle angeschlossenen Einheiten ein 'Z' liefern (vgl. Abb. 5.8). Man spricht davon, dass die Busleitung mit einem Widerstand (passiv) **terminiert** ist. Da das Aufladen der Buskapazität auf '1' über diesen Widerstand relativ lange Zeiten benötigt, versucht man, durch spezielle Schaltungen (**aktive Terminatoren**) das Aufladen auf '1' zu beschleunigen.

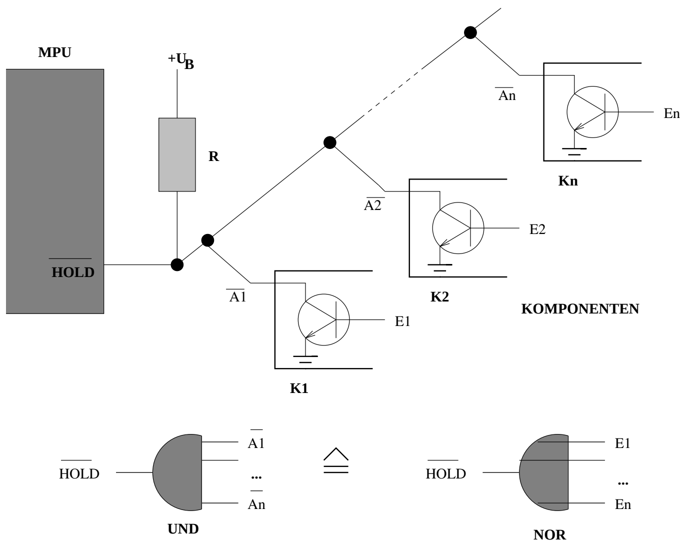


Abbildung 5.8: Open-Collector Anwendung

Damit gilt für das Signal $\overline{HOLD}$ der Abb. 5.8:

$$\begin{aligned} \overline{HOLD} &= (\overline{A_1} \wedge \overline{A_2} ... \wedge \overline{A_n}) \\ HOLD &= E_1 \vee E_2 ... \vee E_n \end{aligned}$$

[2] Der Wert 'Z' ist in den üblichen VHDL-Paketen ebenfalls vordefiniert.

Man kann eine Open-Collector-Schaltung also gut nutzen, wenn eine Oder-Verknüpfung der Ausgänge aller Einheiten (*wired-OR*) benötigt wird. Beispielsweise, können so Unterbrechungs-Wünsche oder Fehlermeldungen angezeigt werden. Sollen Daten übertragen werden, so müssen zu übertragende Einsen durch 'Z' am jeweiligen Ausgang dargestellt werden und alle nicht benötigten Ausgänge müssen ebenfalls 'Z' liefern.

2. **Tristate-Schaltungen**

   Bei Tristate-Schaltungen sorgt man durch einen separaten Steuereingang dafür, dass beide Transistoren der Ausgangsstufe nichtleitend werden können (siehe Abb. 5.9).

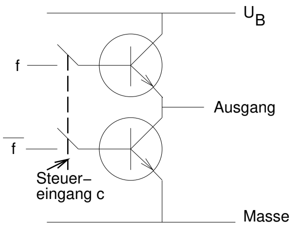


Abbildung 5.9: TriState-Ausgangsstufe (Prinzip)

Abb. 5.10 zeigt die Funktion und eine Anwendung derartiger Tristate-Schaltungen.

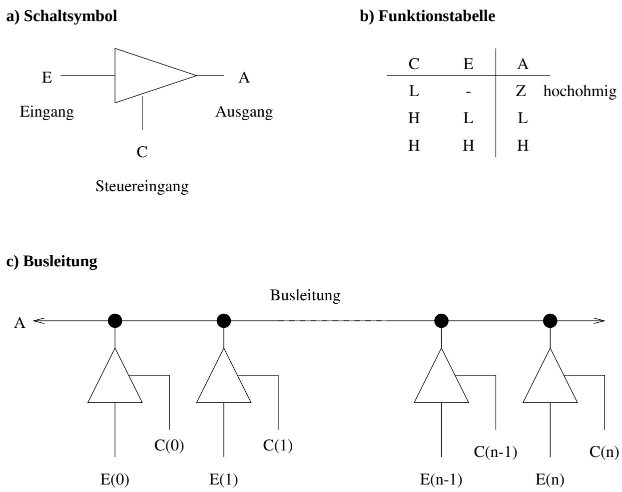


Abbildung 5.10: Tristate-Schaltungen

Ein Vorteil der Tristate-Schaltungen ist, dass sowohl '0'- als auch '1'-Pegel durch geschaltete Transistoren hergestellt, und damit Leitungskapazitäten rasch umgeladen werden können. Der Wechsel auf '1' ist damit schneller als bei Open-Collector-Schaltungen. Andererseits ist die Realisierung einer Oder-Verknüpfung schwieriger.

5.1.2.2 Leitungsreflektionen

Elektrische Signale breiten sich als elektromagnetische Wellen auf Leitungen mit einer bestimmten Geschwindigkeit aus. Deshalb muß man bei Leitungen auch die Laufzeiteffekte berücksichtigen.

Als erstes betrachten wir dazu ein Leitungspaar, welches am Ende kurzgeschlossen ist (siehe Abb. 5.11 [ZH74]). Am Leitungende muß wegen des Kurzschlusses die Spannung 0 sein. Eine genauere Betrachtung ergibt diesen Wert als Überlagerung des eingehenden Signals und eines invertierten, reflektierten Signals. Diese Reflektion ergibt sich bei elektrischen Wellen analog zu der Reflektion von Wasserwellen oder Reflektionen eines an einem Ende fest eingespannten Seils.

An einem offenen Leitungsende findet ebenso wie bei einem geschwungenen Seil, welches am Ende lose ist, eine Reflektion statt. Die reflektierte Welle hat allerdings in diesem Fall dieselbe Polarität wie die eingehende

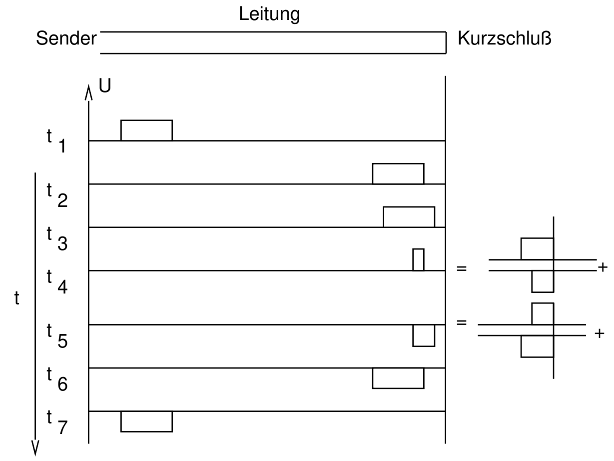


Abbildung 5.11: Reflektionen an einem kurzgeschlossenen Leitungsende

Welle. Als Folge davon wird lokal durch die Überlagerung eine Verdopplung des Signals beobachtet (siehe Abb. 5.12).

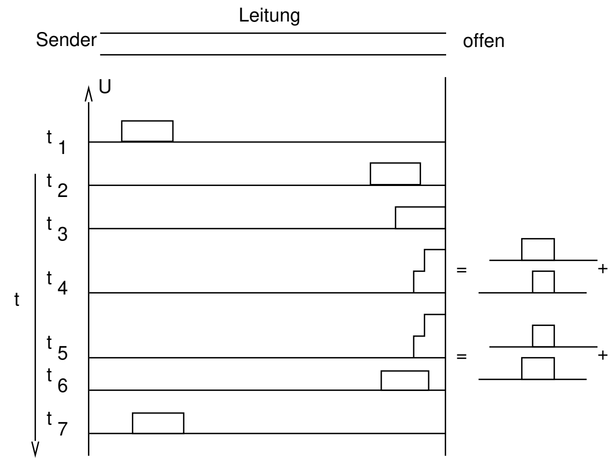


Abbildung 5.12: Reflektionen an einem offenen Leitungsende

Bei Widerstandswerten zwischen 0 und unendlich besitzen die reflektierten Amplituden kleinere Werte. Man kann zeigen [], dass für die Amplitude der reflektierten Welle gilt:

$$r = \frac{R - Z_0}{R + Z_0}$$

Darin ist

$$\begin{array}{rcl} r & = & \dfrac{\text{Amplitude der reflektierten Welle}}{\text{Amplitude der einlaufenden Welle}} \\ R & : & \text{der Widerstand am Ende der Leitung} \\ Z_0 & : & \text{der sog. } \textbf{Wellenwiderstand} \end{array}$$

Z_0 ist ein durch Leitungsform und Material bestimmter Widerstand. Für Koaxialkabel gilt beispielsweise

$$Z_0 \quad = \quad \sqrt{\frac{\ell}{c}}$$

Darin ist ℓ die Induktivität und c die Kapazität des Kabels pro Längeneinheit [ZH74].

Nach einem Sprichwort kann man Koxialkabel mit jedem gewünschten Wellenwiderstand fertigen, vorausgesetzt, der Wert liegt zwischen 50 und 100 Ohm. Für andere Kabel liegen die Werte etwas höher.

Der entscheidende Punkt an dieser Stelle ist: schließt man Leitungen mit einem Widerstand $R = Z_0$ ab, so erfolgt keine Reflektion des Signals. Derartige Abschluß-Widerstände werden auch **Terminatoren** genannt. Der korrekte Leitungsabschluß mit Abschluß-Widerständen ist überall dort erforderlich, wo sich reflektierte Signalen so mit den Nutzsignalen überlagern könnten, dass Störungen (wie z.B. duplizierte Taktimpulse) entstehen könnten. Für digitale Signale gilt die folgende Daumenregel:

> Immer dann, wenn die Laufzeit des Signals auf der Leitung größer wird als die Anstiegszeit des Signals, ist am Leitungsende ein Abschluß-Widerstand erforderlich.

Die Motivation für diese Daumenregel ist folgende: wenn die Laufzeit des Signals auf der Leitung kleiner ist als die Anstiegszeit des Signals, dann äußern sich eventuelle Reflektionen lediglich in kleinen Verformungen der Signalflanken.

Benötigt werden Abschluß-Widerstände beispielsweise bei

1. Bus-Kabeln,
2. Fernseh-Kabelanschlüssen und
3. ISDN S_0-Bussen

5.1.2.3 Differenzielle Signale

Bei der Übertragung von Informationen über Kabel ist es vielfach üblich, Informationen durch Spannungen darzustellen, die auf eine Masseleitung oder auf "Null" bezogen gemessen werden. So kann eine '1' durch einen Spannungsbereich (z.B. von 3,5 bis 5 Volt) zwischen einem Signalkabel **und Masse** kodiert werden. Man bezeichnet diese Form der Signalübertragung als *single ended* oder **asymmetrisch**.

Wenn man an einer hohen Übertragungssicherheit interessiert ist, verwendet man statt *single ended*-Signalen sog. **differenzielle** oder **symmetrische Signale**. Für jedes differenzielle Signal werden zwei Leitungen benötigt, über die jeweils gegenpolige Spannungen geführt werden, bei einer Spannung von U_+ auf der einer Leitung also -U_+ also auf der anderen (siehe Abb. 5.13). An der Empfangsseite befindet sich ein Differenzverstärker. Der Differenzverstärker verstärkt die Spannungsdifferenz zwischen seinen beiden Eingängen um einige Zehnerpotenzen, wobei der resultierende Wert durch die Betriebsspannung oder einen anderen festen Wert (z.B. 0 Volt) begrenzt wird. Ist die Spannung zwischen beiden Eingängen dieses Verstärkers positiv, so generiert er so eine '1' und sonst eine '0'.

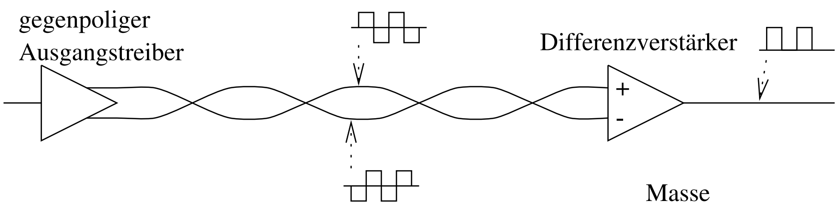


Abbildung 5.13: Differenzielle Signale

Meist kombiniert man die Verwendung differenzieller Signale mit der Verwendung von verdrillten Kabeln (sog. *twisted pairs*), da auf diese Störungen relativ gleichmäßig übertragen werden.

Die Vorteile differenzieller Signale sind die folgenden:

- Störsignale addieren sich in der Regel zu den Signalen auf beiden Leitungen, bleiben also durch die Differenzbildung wirkungslos.
- Der resultierende Logikwert hängt nur von der gegenseitigen Polarität der Signale am Differenzverstärker ab. Änderungen der absoluten Größe der Spannungen durch Dämpfung des Signals entlang der Leitung oder Reflexion spielen keine Rolle.
- Die Anforderungen an die Qualität der Masseverbindungen sind geringer, da über diese keine Signalströme fließen.
- Mit differenziellen Signalen sind wegen der bereits genannten Eigenschaftten meist höhere Übertragungsraten als mit *single ended* Signalen möglich.

Die Nachteile differenzieller Signale sind:

- Es werden negative Spannungen benötigt (es sei denn, man arbeitet mit komplementären statt mit gegenpoligen Signalen).
- Es werden doppelt so viele Signalkabel und Steckeranschlüsse benötigt.

Differenzielle Signale finden u.a. in den folgenden Bereichen Anwendung: *differential SCSI* (s.u.), *token ring*-Netze (s.u.), ISDN, hochwertige Audioübertragung, u.a.m.

5.1.2.4 Optische Nachrichtenübertragung

Aufgrund der zunehmenenden Geschwindigkeitsanforderungen werden vermehrt die Geschwindigkeitsvorteile der optischen Nachrichtenübertragung genutzt. Mittels optischer Nachrichtenübertragung kann eine wesentliche höhere Datentransferrate erzielt werden als bei elektrisch leitenden Kabeln. Der Grund dafür ist, dass es sich bei Licht um eine elektromagnetische Wellen handelt, deren Frequenz deutlich größer ist als die der üblichen Frequenzen in elektrisch leitenden Kabeln und Schaltungen.

Die Übertragung von Licht basiert auf der Möglichkeit, Glasfasern mit sehr geringer Dämpfung herzustellen, wodurch der Abstand zwischen Lichtverstärkern groß sein kann.

Man unterscheidet zwischen Monomode- und Multimode-Glasfasern. Monomode-Fasern besitzen einen niedrigen Querschnitt, sind preislich günstig, erlauben aber nur die Übertragung von Licht einer Wellenlänge. Multimode-Fasern besitzen einen größeren Querschnitt (siehe Abb. 5.14).

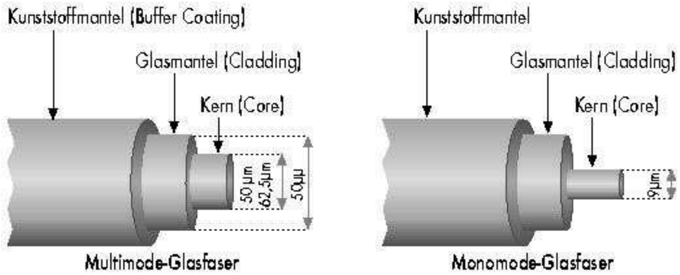


Abbildung 5.14: Vergleich von Multi- und Monomode-Faser (©c't, 1999)

Bei Multimode-Fasern werden Lichtimpulse aufgrund der unterschiedlichen Laufzeiten verbreitert. Diesen Effekt kann man reduzieren, indem man statt eines Stufenprofils des Brechungsindex ein kontinuierliches Profil benutzt (siehe Abb. 5.15).

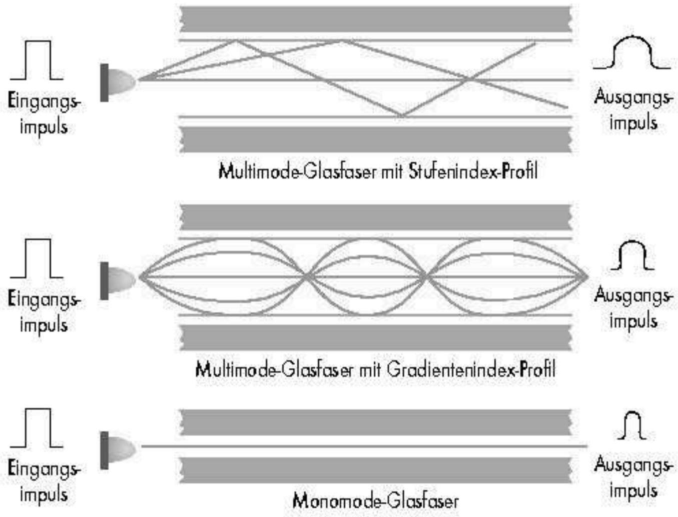


Abbildung 5.15: Breite von Ausgangsimpulsen bei verschiedenen Faserprofilen (©c't, 1999)

5.1.3 Adressierung

Die Geräte-Steuerungen sowie die Geräte müssen von der CPU aus unter bestimmten Adressen angesprochen werden können. Hierfür befinden sich zwei verschiedene Konzepte im Einsatz.

1. Speicherbezogene Adressierung

   Bei der speicherbezogenen Adressierung der E/A-Geräte (engl. *memory mapped I/O*) erfolgt die Kommunikation zwischen Geräten und CPU mittels normaler LOAD- und STORE-Befehle (zumindest soweit die CPU die Kommunikation initiiert). Die Geräte erhalten bei diesem Konzept spezielle Speicheradressen, z.B. Adressen innerhalb einer Seite (siehe Abb. 5.16 a)).

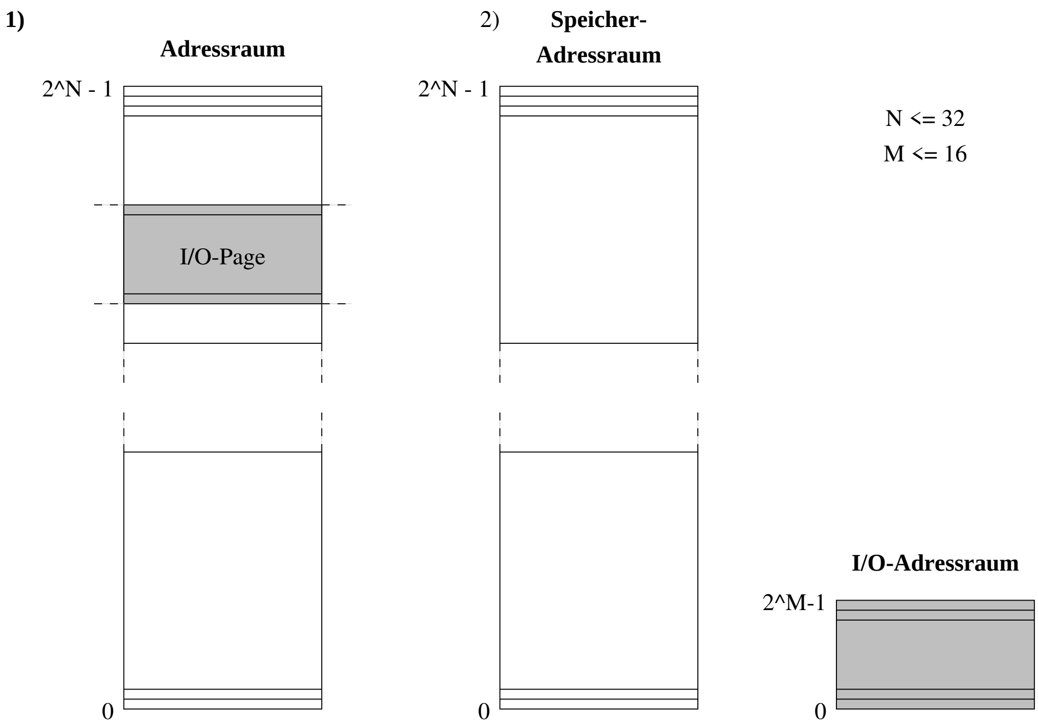


Abbildung 5.16: Adressierung von E/A-Geräten und Gerätesteuerungen

Die CPUs selbst können bei diesem Konzept keine separaten Speicher- und E/A-Schnittstellen haben. Wenn überhaupt eine Trennung der beiden Schnittstellen erfolgt, so müssen die Adressen extern dekodiert und evtl. über Bus-Brücken zur Trennung in zwei Busse genutzt werden. Sonst werden Gerätesteuerungen wie Speicher an einem Bus angeschlossen.

Vorteile:

- Ein großer E/A-Adreßraum kann aufwandsarm realisiert werden. Auf die Größe dieses Raumes muß man sich beim Entwurf des Prozessors noch nicht festlegen, da in der Regel genügend Speicheradressen "abgezweigt" werden können.
- Es werden keine separaten E/A-Maschinenbefehle benötigt. Damit wird der Befehlssatz kleiner.

Nachteile:

- Es ist kein besonderes E/A-Timing möglich.
- Es ist nur über die Speicherverwaltungs-Hardware möglich, normalen Benutzern die direkte Adressierung der E/A-Geräte zu verbieten. Dies wird gewünscht, damit sich Benutzer nicht gegenseitig stören können.

Beispiele: MIPS, MC 680x0, ...

2. Adressierung der E/A-Geräte über eigene E/A-Adressen

   Bei diesem Konzept erfolgt die Kommunikation zwischen Geräten und CPU mittels spezieller E/A-Befehle und einem von den Speicheradressen separaten I/O-Adreßraum (siehe Abb. 5.16 b)).

   Die Vor- und Nachteile dieses Konzeptes ergeben sich aus den Nach- bzw. Vorteilen des zuerst genannten Konzeptes.

   Beispiele: Intel 80x86, IBM S/390

   Auch wenn Prozessoren separate I/O-Adressen unterstützen, müssen Rechnersysteme diese nicht unbedingt ausnutzen. Speicherbezogene Adressierung ist natürlich auch für solche Prozessoren möglich.

5.1.4 Synchrone und asynchrone Busse

Als nächstes widmen wir uns dem Zeitverhalten (dem "Timing") der Signale eines Busses. Man unterscheidet zwischen synchronen und asynchronen Bussen. Die Unterscheidung basiert auf der Unterscheidung zwischen dem **unidirektionalen** und dem **bidirektionalen** Timing [Hay79].

1. **Unidirektionales Timing bei synchronen Bussen**

   Beim unidirektionalen Timing verläßt sich derjenige, der eine Kommunikation initiiert, darauf, dass der Kommunikationspartner innerhalb einer festgelegten Zeitspanne passend reagiert, ohne dies in irgendeiner Weise zu überprüfen. Bei den Fällen Schreiben und Lesen wirkt sich dies wie folgt aus:

   - **Schreiben** (Initiierung durch Sender)

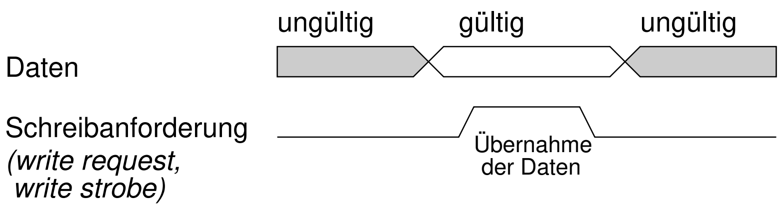


Abbildung 5.17: Schreiben beim unidirektionalen Timing

Die Anforderung wird vom Sender generiert (siehe Abb. 5.17) und zeigt an, dass geschrieben werden soll. Es kann sich z.B. um ein sog. *write-strobe*- oder *write-request*-Signal handeln.

Der Sender verläßt sich darauf, dass der Empfänger die Daten in einer vorgegebenen Zeit auch annimmt.

Bei Bussen werden meist zusätzlich Adressen, die das Ziel des Schreibvorgangs beschreiben, zu übertragen sein. Vielfach sind dafür separate Adressleitungen vorhanden (siehe Abb. 5.18). Ein *address strobe*-Signal zeigt an, dass gültige Adressen und eine gültige Busoperation vorhanden

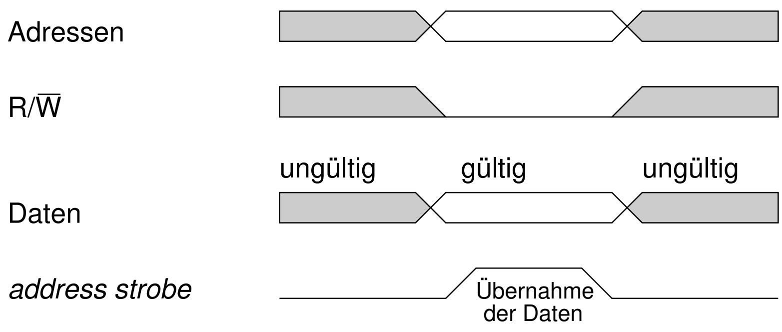


Abbildung 5.18: Schreiben beim synchronen Bus

sind. Das Signal $R/\overline{W}$ ($read/\overline{write}$) muss ebenfalls gültig sein während *address strobe*='1' ist, ansonsten ist es redundant (dargestellt durch die grauen Flächen). Vielfach wird es einen Bezug der Flanken zu Taktsignalen geben.

In der Praxis werden manche der Leitungen (insbesondere *address strobe*) invertiert sein, was aber am Prinzip nichts ändert.

- **Lesen** (Initiierung durch Empfänger)

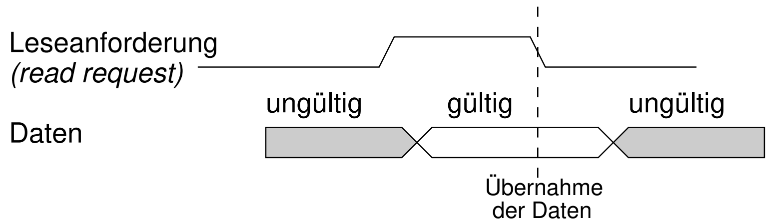


Abbildung 5.19: Lesen beim synchronen Bus

Die Anforderung wird vom Empfänger generiert und zeigt an, dass gelesen werden soll (siehe Abb. 5.19). Der Empfänger verläßt sich darauf, dass der Sender nach einer bestimmten Zeit gültige Daten geliefert hat. Im Falle eines Busses mit Adressleitungen sind die Verhältnisse analog.

2. **Bidirektionales Timing (*hand-shaking*) bei asynchronen Bussen**

   Beim bidirektionalen Timing wird durch ein vom Kommunikationspartner erzeugtes Kontrollsignal bestätigt (engl. *acknowledged*), dass er auf die Initiierung der Kommunikation in der erwarteten Weise reagiert hat.

   Bei den Fällen Schreiben und Lesen wirkt sich dies wie folgt aus:

   - Schreiben (Initiierung durch Sender)
     Der Sender hält Änderungen zurück, bis die Bestätigung eintrifft (Abb. 5.20).

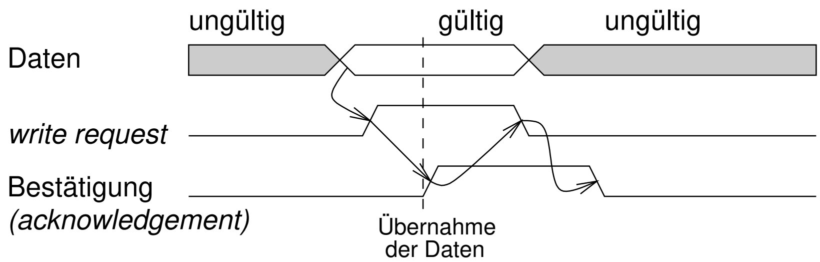


Abbildung 5.20: Schreiben beim birektionalen Timing

Bei Bussen werden wieder meist zusätzlich Adressenleitungen vorhanden sein (siehe Abb. 5.21). Adressen, Daten und das $R/\overline{W}$-Signal müssen gültig sein, solange *address strobe* ='1' ist.

Mit Hilfe der Signale *address strobe* und *data acknowledge* werden letztlich Automaten innerhalb des Senders und des Empfängers synchronisert. Die entsprechenden Zustandsdiagramme zeigt

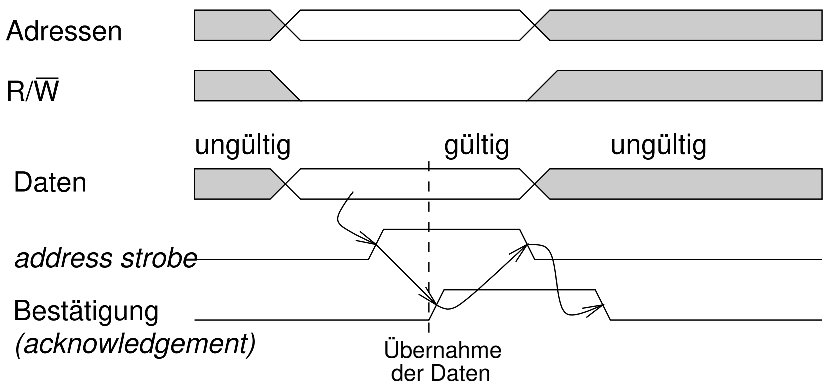


Abbildung 5.21: Schreiben beim asynchronen Bus

Abb. 5.22. Um die Reihenfolge des Anlegens von Adressen und dem zugehörigen *Strobe*-Signal deutlich werden zu lassen, könnte man den ersten Zustand des Empfängers noch in zwei Subzustände aufspalten. Analoges gilt auch für den Sender.

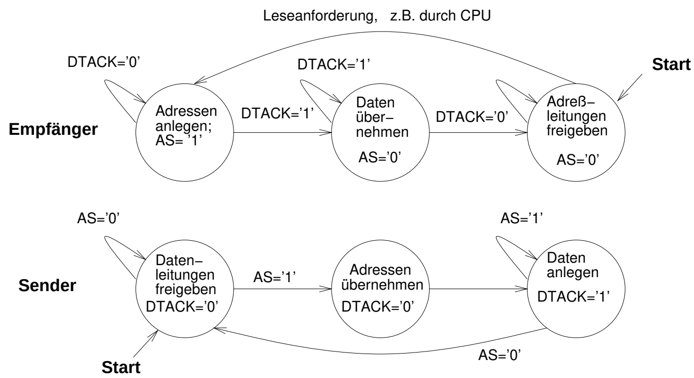


Abbildung 5.22: Zustandsdiagramm eines asynchronen Busses (AS=*address strobe*)

Derartige Zustandsdiagramme sind gut geeignet, um die Funktionen von Bus-Protokollen klar darzustellen. Sie sind daher auch der Ausgangspunkt für formale Überprüfungen der Korrektheit. Alternativ können auch reguläre Ausdrücke benutzt werden, die ja eine äquivalente Beschreibungsform von Automaten darstellen.

Bei Speichern muß die Bestätigung in der Regel außerhalb der eigentlichen Speicherbausteine erzeugt werden (siehe Abb. 5.23)

- **Lesen** (Initiierung durch Empfänger)

  Nach dem Senden der Anforderung kann noch eine große Zeit vergehen, bis der Sender die Bestätigung schickt (siehe Abb. 5.24).

  Die Daten müssen zumindest innerhalb des Zeitintervalls, in dem die Bestätigung = '1' ist, gültig sein.

  Bei Bussen werden meist zusätzlich Adressenleitungen vorhanden sein. Abb. 5.25 zeigt typische Signale in diesem Fall.

Das bidirektionale Timing wird in der Regel mit einer Zeitüberwachung kombiniert, da sonst evtl. das System vollständig blockiert werden könnte, z.B. beim versehentlichen Lesen aus nicht vorhandenem Speicherbereich.

Vor- und Nachteile der beiden Timing-Methoden zeigt Tabelle 5.1.

Die Kriterien für die Wahl des Bus-Timings zeigt auch die Abb. 5.26 [HP96].

Historisch hat man bei Mikroprozessoren aus Gründen der Einfachheit zunächst synchrone Busse benutzt, ist dann aus Gründen der Flexibilität meist auf asynchrone Busse übergegangen und benutzt neuerdings verstärkt wieder synchrone Busse, weil sonst nicht die geforderte Geschwindigkeit erreicht werden kann.

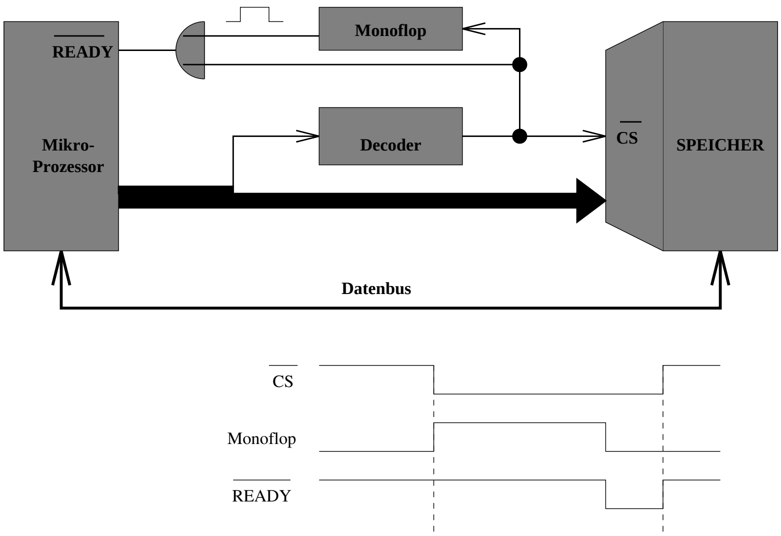


Abbildung 5.23: Erzeugung der Bestätigung bei Speicherbausteinen (READY=*acknowledge*)

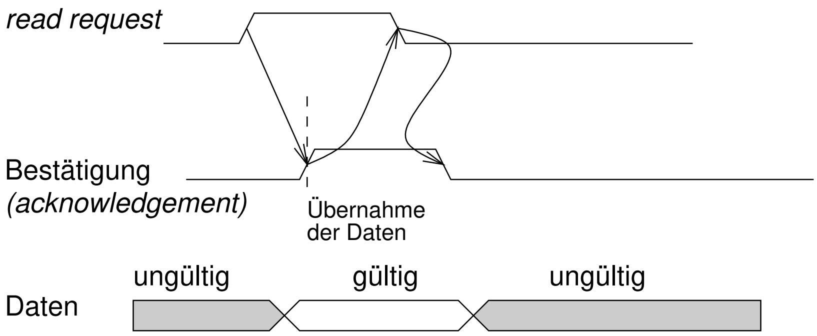


Abbildung 5.24: Lesen bei birektionalem Timing

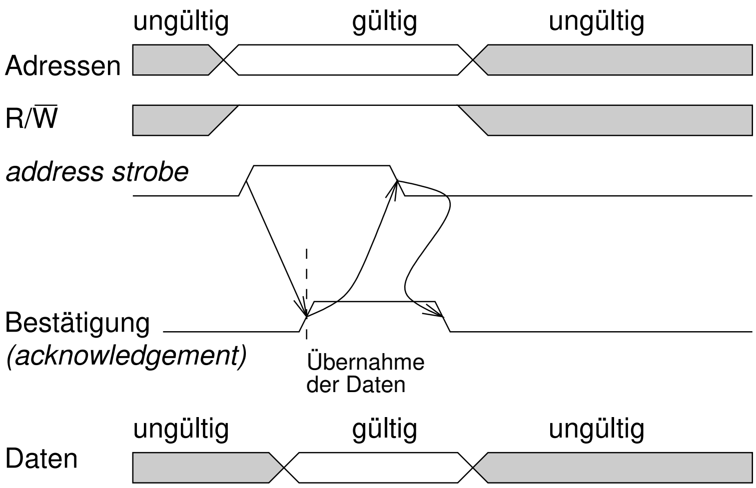


Abbildung 5.25: Lesen beim asynchronen Bus

Bähring [Bae94] betrachtet als Kompromiß **semisynchrone Busse**. Bei diesen muß der Kommunikationspartner eine negative Bestätigung senden, falls er nicht in der erwarteten Zeit antworten kann. Da immer auf negative Antworten gewartet werden muß, ist die Übertragungsgeschwindigkeit wie beim asynchronen Bus durch die Laufzeit der *request*- und *acknowledge*-Signale begrenzt.

	unidirektional	bidirektional
Vorteile	einfach; bei konstanten Antwortzeiten schnell	paßt sich unterschiedlichen Geschwindigkeiten an
Nachteile	Kommunikationspartner muß in bestimmter Zeit antworten	komplexer; Zeitüberwachung notwendig evtl. langsam (2 Leitungslaufzeiten!)
	− > synchrone Busse	− > asynchrone Busse

Tabelle 5.1: Unidirektionales und bidirektionales Timing

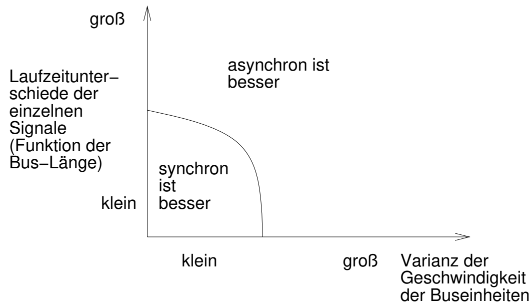


Abbildung 5.26: Kriterien für die Wahl des Bus-Timings

5.1.5 Ablauf der Kommunikation zwischen CPU und Gerätesteuerungen

Als nächstes wollen wir betrachten, wie die Leitungen eines Busses zum Zweck des geordneten Ablaufs der Kommunikation genutzt werden. Wir betrachten verschiedene Möglichkeiten, den Ablauf und die Synchronisation zu regeln.

5.1.5.1 Direkte Ein-/Ausgabe (*immediate devices*)

Die erste Methode der Ein-/Ausgabe wird hier nur der Vollständigkeit halber erwähnt, denn sie beinhaltet eigentlich überhaupt keine Methode der Synchronisation von CPU und Geräten bzw. Gerätesteuerungen: in Sonderfällen sind Geräte und Gerätesteuerungen so schnell, dass sie unter allen Umständen mit der Geschwindigkeit der CPU mithalten können. Da diese Geräte Ein- und Ausgaben “sofort” verarbeiten, heißen sie auch *immediate devices*. Eine explizite Synchronisation ist für diese nicht erforderlich.

Beispiele:

- Setzen von digitalen Schaltern/Lämpchen
- Ausgabe an schnelle Digital/Analog-Wandler
- Schreiben in Puffer
- Setzen von Kontroll-Registern

Das Programmieren beschränkt sich in diesem Fall auf die Erzeugung von LOAD- und STORE bzw. INPUT- und OUTPUT-Befehlen. Dementsprechend gibt es keine besonderen Anforderungen an die E/A-Organisation der Bus-Hardware.

5.1.5.2 Status-Methode (*Busy-Waiting*)

Bei der zweiten Methode wird in einer Warteschleife anhand der Abfrage eines “Bereit”-Bits geprüft, ob das Gerät (bzw. der *Controller*) Daten verarbeiten können.

Prinzip:

```
 REPEAT
  REPEAT
   lese Statuswort des Gerätes
  UNTIL Bereit-Bit im Statuswort ist gesetzt;
  lese Datenwort und speichere es im Datenblock ab;
  erhöhe Blockzeiger;
 UNTIL Ende des Blocks ist erreicht
```

Beispiel (in Anlehnung an Hayes [Hay79]) :

Gegeben sei eine Gerätesteuerung, bei der unter Adresse 1 der gegenwärtige Status und unter Adresse 2 die aktuelle Eingabe eingelesen werden kann (siehe Abb. 5.27).

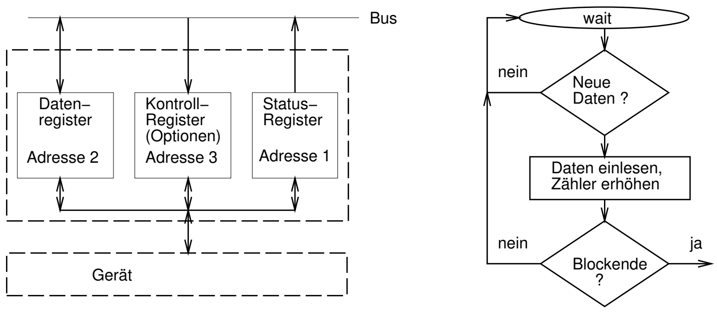


Abbildung 5.27: Realisierung von *busy waiting*

Das folgende Assembler-Programm für die Intel 80x86-Prozessorfamilie liest einen Datenblock ein:

```
      -- Register B enthalte die Blocklänge, (H&L) die Anfangsadresse
wait: IN 1                    -- A := Gerätestatus
      CPI  ready              -- Z := Bereit-Bit
      JNZ  wait               -- if nicht bereit then goto wait
      IN 2                    -- A := Datenregister der Steuerung
      MOV M,A                 -- MAIN[M + (H&L)]:= A
      INX H                   -- (H&L) := (H&L) +1
      DCR B                   -- B := B - 1; Z := (B=0)
      JNZ wait                -- if Z<>'0' then goto wait
ready ....                    -- block eingelesen
```

Nachteile dieser Methode sind:

- Keine Möglichkeit der verzahnten Bearbeitung anderer Aufgaben
- Geringe Übertragungsgeschwindigkeit (vgl. Bähring, Abb. 3-7.1)

Anwendung wird diese Methode in den folgenden Fällen:

- Wenn sowieso keine anderen Aufgaben vorliegen (Booten, Prozessoren in Controllern)
- Wenn das Gerät so schnell ist, dass die Schleife selten durchlaufen wird und für die Umschaltung auf andere Aufgaben keine Zeit bleibt
- Wenn das Gerät zu keiner intelligenteren Methode (DMA) fähig ist

Im Gerätetreiber müssen in diesem Fall v.a. die o.a. Schleifen vorkommen.

5.1.5.3 *Polling*

Bei der dritten Methode erfolgt ein gelegentliches Prüfen des Bereit-Bits verschiedener Geräte mit verzahnter Bearbeitung anderer Aufgaben.

Beispiel:

Diese Methode wird eingesetzt, um Übertragungswünsche einer großen Anzahl von Terminals, die an einem Rechner angeschlossen sind, zu erfassen. Das Verfahren erlaubt eine gerechte Bedienung aller Terminals und vermeidet Überlastungen durch eine große Anzahl praktsich gleichzeitig eintreffender Übertragungswünsche.

Ein Nachteil des Verfahrens ist v.a. die u.U. große Reaktionszeit.

Die Techniken aus den letzten drei Abschnitten heißen auch **programmierte Ein-/Ausgabe** (engl. *programmed I/O*).

5.1.5.4 Unterbrechungen

Bei der vierten Methode unterbricht die Geräte-Steuerung die CPU, falls Datentransport(e) erforderlich werden oder falls Fehler auftreten.

Der Ablauf ist der folgende (siehe auch Abb. 5.28):

- Prozess A überträgt E/A-Auftrag mittels SVC an Betriebssystem
- Dispatcher schaltet auf neuen Prozess um

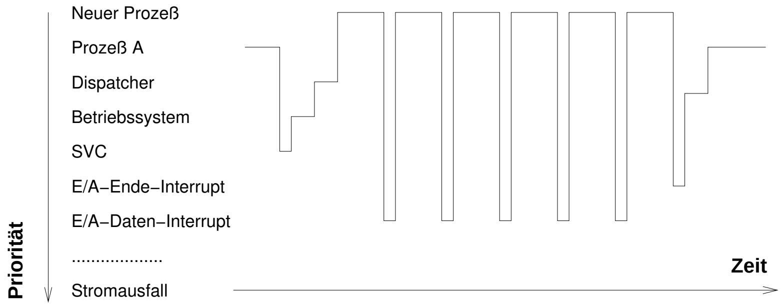


Abbildung 5.28: Ablauf bei der Interrupt-Verarbeitung

Im Falle eines Interrupts passiert Folgendes:

- Gerät sendet Interrupt.
- Falls Interrupts nicht gesperrt sind und die Priorität ausreichend hoch ist und der laufende Maschinenbefehl ist abgeschlossen: Sicherung des Zustandes (per Software oder μPgM)
- Feststellen der Interrupt-Ursache und Verzweigung (falls die hardwaremäßig geschieht, so heisst dies *vectored interrupt*)
- Datentransport: E/A, Lesen/Schreiben Speicher, Pufferzeiger erhöhen, Test auf Block-Ende
- Restaurieren des Kontexts, Return from Interrupt
- Prozess fährt in Bearbeitung fort
- Nach einer Reihe von Datenübertragungsinterrupts erfolgt ein Blockende-Interrupt, welcher die Beendigung des Datentransports signalisiert und den Dispatcher einschaltet

Wegen des Overheads ist diese Methode nicht für die schnelle Datenübertragung geeignet (<< 100 k Interrupts/sec)

Programmtechnisch muß sowohl für das geordnete Starten der Geräte vor der Übertragung als auch für eine passende Interrupt-Routine gesorgt werden.

Für Rechner ohne DMA (s.u.) und mit relativ langsamer Peripherie ist diese Methode erforderlich.

5.1.5.5 *Direct Memory Access* (DMA)

Die einfachen Operationen im Falle eines Datentransports können auch ohne Benutzung der CPU realisiert werden. **Auf diese Weise kann der Datentransport nebenläufig zur Bearbeitung eines (in der Regel anderen) Prozesses auf der CPU erfolgen**.

Im folgenden wollen wir den Ablauf von DMA-Transfers am Beispiel der Motorola 68000er-Prozessorn im Detail beschreiben. Die benutzte Hardware beschreibt Abb. 5.29.

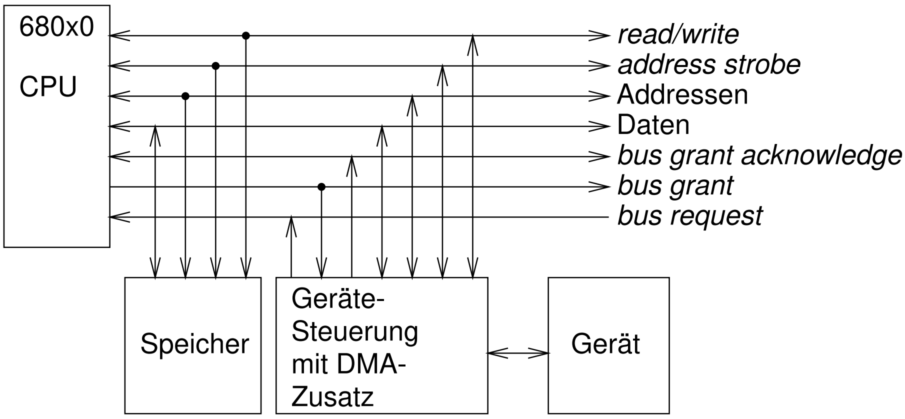


Abbildung 5.29: Bus des MC 680x0

Der Ablauf von DMA–Transfers ist wie folgt:

Von der Platte gelesenes Wort liegt vor, Gerätesteuerung setzt *bus request*;

→ Falls die CPU bereit ist, den Bus abzugeben (auch innerhalb von Befehlen): CPU setzt *bus grant*, Adressen hochohmig, kein Adreßstrobe (es wird nicht angezeigt, dass die Adressen gültig sind);

→ Controller setzt *bus grant acknowledge*;

→ *bus grant* geht auf 0;

→ Controller belegt Adreß- und Datenbus, setzt Adreßstrobe (Adresse gültig);

→ Speicher übernimmt Information von Datenbus in die adressierte Zelle (im Blockmode können die beiden letzten Schritte n-fach wiederholt werden, n meist nicht größer als 4);

→ Controller nimmt *bus grant acknowledge* zurück;

→ CPU kann den Bus ggf. wieder benutzen, Controller erhöht den Pufferzeiger und prüft auf Blockende.

Zur Erzielung einer möglichst höhen Datenrate werden eventuell mehrere Worte über den Bus übertragen, ohne dass der Bus neu zugeordnet werden kann. Diese Form der Übertragung heißt **Blockmode**.

Abgesehen von der Anfangs- und Ende-Behandlung wird die CPU nur durch **gestohlene Speicherzyklen** in der Geschwindigkeit beeinträchtigt. Diese Realisierung heißt daher *cycle-stealing DMA*.

Die Realisierung von *non-cycle stealing DMA* ist mittels Multiportspeichern und getrennten Bussen für E/A und CPU möglich. In diesem Fall Reduktion der CPU-Geschwindigkeit nur durch gelegentlichen Zugriff auf denselben Speicher. Für den 3-Port Speicher siehe Abb. 4.11.

Das Programmieren von DMA-Transfers ist, von *immediate devices* abgesehen, wohl am Einfachsten. Es sind lediglich vor der Übertragung die jeweiligen Parameter an die Steuerung zu übermitteln. Danach läuft alles Andere ohne Programmkontrolle ab. Lediglich am Ende der Bearbeitung, welches z.B. über einen Interrupt signalisiert werden kann, muß der beauftragende Prozeß noch ausführungsbereit erklärt und der Dispatcher gestartet werden. Per Interrupts können weiterhin Fehler signalisiert werden, zu deren Behandlung Software erstellt werden muß.

DMA-Einheiten können mit unterschiedlich viel Intelligenz ausgestattet sein. Zwei extreme Fälle sind die folgenden beiden:

- DMA-Controller

  Diese besitzen eine fest verdrahtete Folge einfacher Operationen wie z.B. "lese Wort", "erhöhe Zeiger" oder "prüfe Länge".

  Gelegentlich findet man bei Controllern eine Realisierung der **Daten-Kettung** (engl. *data-chaining*): Mit einem Befehl an Controller wird in diesem Fall eine Menge ggf. unzusammenhängender Speicherbereiche übertragen (notwendig beim Auslagern von Prozessen mit unzusammenhängendem realen Adreßbereich).

- E/A-Prozessor

  E/A-Prozessoren können vollständige Programm abarbeiten (sog. **Kanalprogramme**). Damit können Befehle wie Rückspulen, Kopf-Positionierung, Fehlerprüfung und -behandlung ohne CPU-Belastung durchgeführt werden.

  Extremfall: Das gesamte Betriebssystem läuft auf dem E/A-Rechner. Die CPU wird nur durch gelegentliche Prozeßwechsel unterbrochen. Beispiel: CDC 6600

Speziell bei IBM-kompatiblen Großrechnern der Serie S/370 und einem Teil der Nachfolger[3] wird zwischen drei Arten von Kanälen unterschieden, die jeweils eine unterschiedliche Flexibilität hinsichtlich ihres Kanalprogrammms besitzen:

- Der Selektor-Kanal

  Selektor-Kanäle beitzen einen exklusiven Weg zum Programm-gewählten Gerät. Der Weg (Kanal) ist für das Gerät reserviert, egal, ob das Gerät Daten überträgt oder z.B. nur den Kopf positioniert. Diese Kanäle bieten den schnellsten Anschluß von Geräten.

- Der Block-Multiplex Kanal

  Der Block-Multiplex Kanal besitzt zwei Betriebsarten:

  1. Betriebsart mit exklusivemr Weg analog zum Selektor-Kanal
  2. Betriebsart mit blockweiser Umschaltung zwischen Geräten

     Während der Suchphase eines Kopfes auf der Platte kann der Kanal auf ein zweites Gerät umschalten. Das Gerät meldet sich am Ende des Suchens mittels eines **Kanalanrufs**. Ist der Kanal gerade beschäftigt, muß das Gerät den Versuch wiederholen.

- Der Byte-Multiplex-Kanal

  Der Byte-Multiplex-Kanal besitzt zwei Betriebsarten:

  1. Betriebsart mit exklusivem Weg analog zum Selektor-Kanal
  2. Betriebsart mit byteweisem Umschalten zwischen Geräten (meist Terminals)

[3] Siehe www.s390.ibm.com.

5.1.6 Buszuteilung

Bei direktem Zugriff von der Gerätesteuerung (Controller) auf den Speicher benötigt der Controller die Kontrolle über den Bus (er muß **Bus-Master** werden).

Im Falle der 68000er-Prozessoren steuert die CPU die Zuteilung des Busses. Bei Multiprozessor-Systemen und bei Bussen, die nicht direkt an der CPU angeschlossen sind, muß diese Funktion von einer separaten Einheit übernommen werden. Die Komponente, die diese Funktion realisiert, heißt *arbiter* (deutsch: **Schiedsrichter**[4]).

Es gibt verschiedene Methoden der Arbitrierung ([HP95], Abschn. 8.4):

1. ***Daisy Chaining***[5]:

   In this scheme, the bus grant line in run through the devices from highest priority to lowest (the priorities are determined by the position on the bus). A high-priority device that desire bus access simply intercepts the bus grant signal, not allowing a lower priority device to see the signal. Figure[6] *... shows how a daisy chain is organized.. The advantage of a daisy is simplicity; the disadvantages are that it cannot assure fairness – a low priority request may be locked out indefinitely – and the use of the daisy chain grant signal also limits the bus speed. The VME bus, a standard backplane bus, uses multiple daisy chains for arbitration* [HP95].

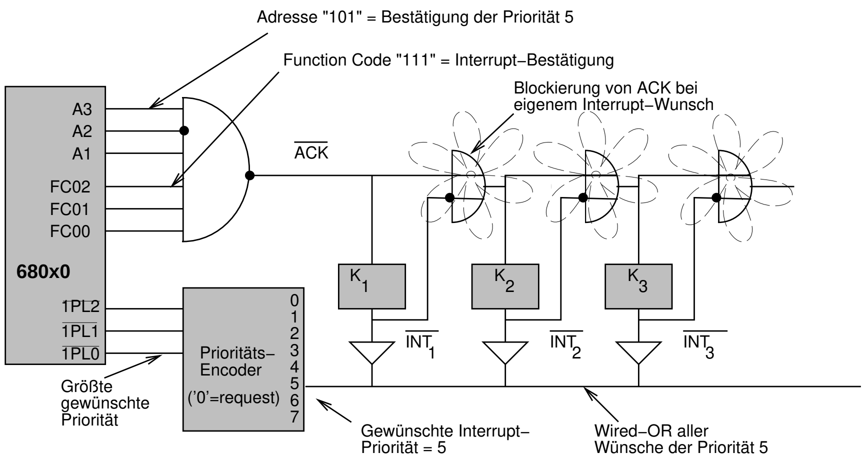


Abbildung 5.30: *daisy chaining* bei Motorola 68000er-Prozessoren

2. ***Centralized, parallel arbitration*** (siehe Abb. 5.31):

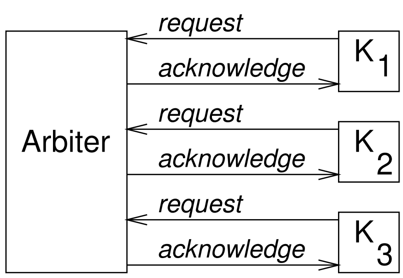


Abbildung 5.31: *Centralized parallel arbitration*

   These schemes use multiple request lines, and the devices independently request the bus. A centralized arbiter chooses from among the devices requesting bus access and notifies the selected device that it is now bus master. The disadvantage of this scheme is that it requires a central arbiter, which may become the bottleneck for bus usage [HP95].

[4] Siehe auch Abb. 5.2-1, 5.2-2, 5.2-3 des Buches von Bähring.
[5] Deutsch: Gänseblümchen-Ketten
[6] Siehe Abb. 5.30 [Bae94].

3. ***Distributed arbitration by self-selection*** (siehe Abb. 5.32):

   These schemes also use multiple request lines, but the devices requesting bus access determine who will be granted access. Each device wanting access places a code indicating its identity on the bus. By examining the bus, the devices can determine the highest priority device that has made a request. There is no need for a central arbiter; each device determines independently whether it is the highest priority requester. This scheme, however, does require more lines for request signals. The NuBus, which is the backplane bus in the Apple MAcintosh IIs, uses this scheme [HP95].

   Die Benutzung der Datenleitungen zur Arbitrierung ist der Grund dafür, dass bei manchen Bussen die Anzahl der Geräte auf die Anzahl der Datenleitungen begrenzt ist. Dies ist insbesondere beim SCSI-Bus der Fall. .

   Abb. 5.32 zeigt eine mögliche Realisierung.

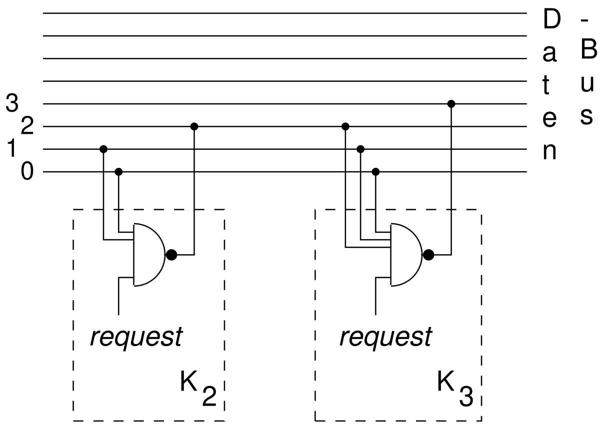


Abbildung 5.32: *Distributed arbitration by self-selection*

   Eine '0' auf einer Datenleitung i signalisiert, dass das Gerät i einen Arbitrierungswunsch hat. Jedes Gerät ist selbst dafür verantwortlich, vor dem Anmelden eines Wunsches die Datenleitungen der Geräte mit höherer Priorität zu prüfen. In der Abbildung geschieht dies, weil eine '0' auf den entsprechenden Datenleitungen die Weiterleitung von Busanforderungen an den Bus selbst verhindert.

4. ***Distributed arbitration by collision detection***:

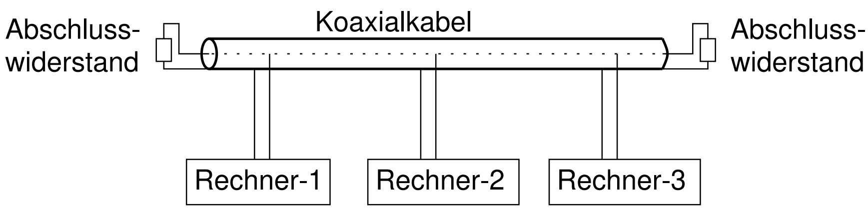


Abbildung 5.33: *Distributed arbitration by collision detection* beim Ethernet

   In this scheme, each device independently requests the bus. Multiple simultaneous requests result in a **collision**. *The collision is detected and a scheme for selecting among the colliding parties is used. Ethernets, which use this scheme, are further described in ...* [HP95] (siehe Seite 155).

5. ***Token ring, token bus, ...***

   Im Übrigen können alle Techniken zur Arbitrierung bei lokalen Netzwerken, wie z,B. *Token ring, token bus* usw. (siehe unten) auch zur Arbitrierung bei Bussen eingesetzt werden.

5.1.7 Interrupt-Controller

Im Falle der 68000er CPU übernimmt die CPU alle Funktionen der Interrupt-Verarbeitung. Für die Intel-Prozessoren der 80x86er-Linie existieren separate Interrupt-Controller (siehe Abb. 5.34).

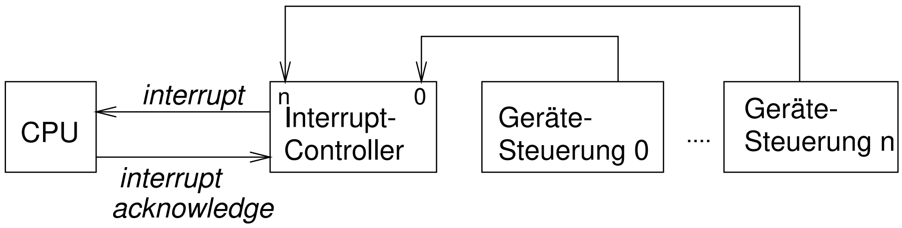


Abbildung 5.34: Prinzip der Verwendung separater Interrupt-Controller

5.1.8 Weitere Eigenschaften von Bussen

Einige spezielle Eigenschaften von Bussen sollen hier noch aufgeführt werden:

- **Multiplexing von Adressen und Daten**: Zur Einsparung von Leitungen werden bei manchen Bussen Adressen und Daten nacheinander übertragen. Man nennt dies **Multiplexing**.
- ***Split transaction busses***: Das Verstecken von Latenzzeit wird erleichtert, wenn insbesondere bei Leseaufträgen der Bus zwischen dem Senden der Adresse und dem Empfangen der Daten weitere Aufträge abgewickeln kann. Zu diesem Zweck können Aufträgen Identifier aus wenigen (z.B. 3) Bits mitgegeben werden. Anhand eines solchen Identifiers können die eintreffenden Daten dann wieder der Adresse zugeordnet werden.

5.1.9 Standardbusse

Es gibt eine Vielzahl von Standard-Bussen, die jeweils aufgrund bestimmter Zielvorstellungen und technischer Möglichkeiten definiert wurden [MSS91, HP95]. Wir werden hier zwischen Speicherbussen, lokalen Bussen und Peripheriebussen unterscheiden (vgl. Abb. 5.6). Tatsächlich kann ein bestimmter Standard mehreren Zwecken dienen und für einen bestimmten Zweck können aus Kompatibiltätsgründen mehrere Standards dienen.

5.1.9.1 Speicherbusse

Bei der Darstellung beginnen wir mit den Speicherbussen (vgl. Abb. 5.6). Speicherbusse sind die schnellsten Busse überhaupt. Sie sind vielfach Hersteller-spezifische Busse. In der Regel besteht keine Notwendigkeit der Kompatibilität mit Bussen anderer Hersteller. Tabelle 5.2 enthält charakteristische Daten der Speicherbusse einiger leistungsstarker Server (1994). Die hohe Leistung ist sehr klar zu erkennen.

	Hp Summit	SGI Challenge	Sun XDBus
Datenleitungen	128	256	144
Bustakt [Mhz]	60	48	66
Transfer-Rate (max) [MB/s]	960	1200	1056

Tabelle 5.2: Daten einiger Speicherbussse

5.1.9.2 Lokale Busse

Lokale Busse wurden ursprünglich vielfach "Systembusse" genannt. Eine Unterscheidung zwischen beiden wurde durch die unzureichende Geschwindigkeit mancher Systembusse notwendig. Dieser Mangel führte zur Einführung lokaler Busse unter Beibehaltung der bisherigen Systembusse aus Kompatibilitätsgründen. Wir werden daher lokale Busse und Systembusse als eine Klasse von Bussen behandeln.

Die nachfolgende Liste enthält einige bekannte lokale Busse.

- Multi-Bus I

  Dieser veraltete Intel-Bus hat eigentlich nur als Vorläufer des Multi-Bus II eine Bedeutung.

- Multi-Bus II

  Dies ist ein synchroner, sehr genau spezifizierter Bus für Systeme mit verteilter Kontrolle. Aufgrund der genauen Spezifikation dient er gelegentlich als Beispiel für die formale Verifikation von Bus-Protokollen. Seine Anwendung erfolgte u.a. in Apollo-Systemen etwa bis 1987, in einigen eingebetteten Systemen (z.B. der Fa. Siemens) bis heute (1996).

- NuBus

  Der NuBus ist der Bus des Apple Macintosh und TI-Explorer (ab 1983). Er erlaubte als einer der ersten Busse **Blocktransfer** von 2, 4, 8 oder 16 Worten ohne neue Busvergabe.

- AT-Bus, ISA-Bus

  Der AT-Bus ist der Speicher- und Peripheriebus für IBM-PCs ab 1981. Es ist ein einfacher, synchroner Bus. Er wurde von IBM selbst nicht publiziert, von einigen Firmen aber als ISA (industry standard architecture)-Bus spezifiziert. Für den ISA-Bus sind viele Karten verfügbar. Der ISA-Bus sollte von der Konzeption her mehrere Aufgaben übernehmen, wird aber heute nur noch aus Kompatibilitäts-Gründen realisiert, um nicht allzu schnelle Controller-Karten anzuschließen.

- EISA-Bus

  Dies ist eine zum ISA-Bus kompatible Erweiterung auf 32 Bit. Bislang wurden nur wenige Karten für diesen Bus entwickelt. Durch neuere Entwicklungen wird sich dies voraussichtlich auch nicht ändern.

- VME-Bus

  Der VME (Versa Module Europe)-Bus ist ein verbreiteter 32-Bit Standardbus. Ursprünglich wurde er für 32-Bit Datenbreite und 24-Bit-Adreßbreite definiert, später aber nicht immer ganz einheitlich erweitert. Die Normierung ist infolge der Erweiterungen problematisch.

- Micro-Kanal

  Der Micro-Kanal (MCA) wurde von IBM als Nachfolger für den relativ langsamen AT-Bus konzipiert. Er bietet 8-Bit-, 16-Bit- und 32-Bit-Übertragung sowie eine Integration von Video- und Analogsignalen. Das Hauptproblem MCA-basierter Systeme liegt darin, dass kein zusätzlicher ISA-Bus vorgesehen ist.

- SBus

  Von der Fa. Sun freigegebener Bus der SPARC-Systeme. Synchrones Protokoll mit einer Taktfrequenz von 16,66 bis 25 MHz. Adreßleitungen für virtuelle und reale 32-Bit-Adressen (weil MMU spät verfügbar war).

- Future-Bus

  Bus für hohe Geschwindigkeit. Adressen und zugehörige Daten können zeitlich verschränkt werden (es können für mehrere Adressen Leseanforderungen an den Bus gehen, bevor das erste Datum transportiert wird). Der Bus wurde von ANSI (American National Standardization Institute) genormt.

- PCI-Bus

  Der PCI-Bus wurde als CPU-unabhängiger Bus zur Benutzung vor allem auf der Systemplatine konzipiert. Obwohl er von der Fa. Intel für den Einsatz in Pentium-basierten Systemen gedacht war, wird er auch für andere Prozessoren angeboten. Der PCI-Bus wird über eine Brücke mit dem eigentlichen CPU-Bus gekoppelt und erlaubt auf beiden Seiten der Brücke die gleichzeitige Ausführung von unabhängigen Operationen und Transfers. Der PCI-Bus erlaubte ursprünglich maximal den Anschluß von drei PCI-Steckplätzen. Daten- und Adreßleitungen werden gemultiplext. Die sehr präzise Standardisierung erlaubt 32 und 64 Bit breite Daten- bzw. Adreßleitungen und die Benutzung von 3,3 statt der bislang 5 Volt Betriebsspannung. Er kann mit 33 und 66 MHz getaktet werden. Er unterstützt eine automatisierte Konfiguration [Sch93].

- AGP

  Wegen der Leistungsbegrenzung des PCI-Bus wurde für leistungsstarke Graphikkarten eine neue Schnittstelle eingeführt, ein *advanced graphics port* (AGP).

Die Tabelle 5.3 enthält charakteristische Daten einer Reihe von lokalen Bussen.

	ISA-Bus s	Future-Bus	S-Bus s	PCI-Bus	PCI-100	AGP (4x)
Einführungsjahr	1981	1984	1989			
Datenleitungen	8/16	32+1	32	32 (64)	32 (64)	
Adressenleitungen	20	32+1	28+32	32 (64)	32 (64)	
synchron/asynchron	synchron	asynchron	synchron	synchron	synchron	
Bustakt [Mhz]	4,77	-	16,7-25	33	100	266
Transfer-Rate, Einzelworte bei $T_{acc} = 0ns$, [MB/s]		37,0				
Transfer-Rate (Block-Mode) [MB/s]	9,54	95,2	100	111	333	1000
Bemerkung	8086-Bus	message-passing	SPARC-Bus			Punkt-zu Punkt-Verb.

Tabelle 5.3: Daten einiger Busse

5.1.9.3 Peripherie-Busse

Peripheriebusse sind für die Übertragung über größere Entfernungen (einige Meter) ausgelegt. Die folgende Liste enthält gebräuchliche Peripheriebusse:

- **SCSI**

  Als wichtigen Bus wollen wir hier den SCSI-Bus nennen. Der SCSI-Bus ist ein Bus, der ursprünglich v.a. für den Anschluß von Plattenlaufwerken gedacht war. Gegenüber anderen derartigen Anschluß-Standards zeichnet er sich v.a. durch eine weitgehende Abstraktion von der physikalischen Struktur der Platten aus. Durch die Definition von Lese- und Schreibbefehlen erlaubt er relativ intelligente Plattensteuerungen und sorgt so für eine Entlastung der CPU. Bis zu 8 Einheiten können an einen SCSI-Bus angeschlossen sein.

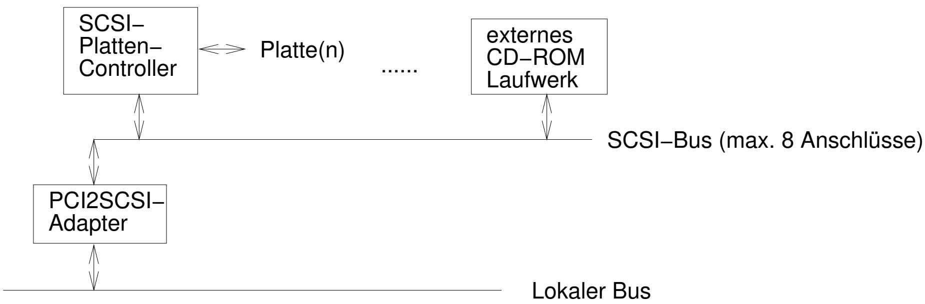


  Abbildung 5.35: Anschlußorganisation von SCSI-Controllern

  Von Prinzip her kann jede der am SCSI-Bus angeschlossenen Komponenten als Initiator einer Datenübertragung auftreten [Bae94]

  Es gibt folgende Unterscheidungsmerkmale von SCSI-Bussen [HS93, Tra96]:

  – SCSI-1

    Dies ist die ursprüngliche SCSI-Definition, die einen 8 Bit breiten Bus benutzte und deren Befehle v.a. Plattenlaufwerke unterstützen. Mit der überwiegend benutzten asynchronen Betriebsart läßt sich eine Übertragungsrate von bis zu 3,3 MByte/s erreichen. Die maximale Bus-Länge beträgt 6 Meter.

  – SCSI-2

    SCSI-2 verlangt synchronen Datentransfer und erlaubt Datenraten von bis zu 10 MByte/s. SCSI-2 definiert Befehle auch für andere Geräte, z.B. von CD-ROMs, optischen Speichern, Druckern und Bandlaufwerken.

  – Narrow/Wide SCSI

    SCSI-2 erlaubt auch die Option *Wide SCSI*. Diese Option ist nicht steckerkompatibel, erlaubt aber 16 und 32-Bit Datenleitungen. Genutzt werden v.a. 16-Bit-Datenleitungen.

- Fast/Ultra/Ultra2/Ultra160

  Diese Bezeichnungen beziehen sich auf eine Erhöhung der Anzahl der Datentransfers pro Sekunde durch höhere Taktraten. Fast-, Ultra- und Ultra2-SCSI erlauben 10, 20, bzw. 40 Millionen Transfers pro Sekunde. Ultra160 erlaubt 80 Mill. Transfers, was in Kombination mit (16-Bit-) Wide-SCSI 160 MB/s ergibt.
- differenzielle Signale

  Die schnelleren Taktsignale sind nur bei einer Verkürzung der Leitungslänge und/oder bei einer Reduktion der Anzahl der Geräte gegenüber SCSI-1 möglich (siehe Tabelle 5.4). Zur Verbesserung der Übertragungsqualität und im die Leitungslänge nicht unerträglich sinken zu lassen ist der Einsatz differenzieller Übertragung (*differential SCSI*) möglich. Diese Option wird v.a. bei Workstations eingesetzt.
- serielles SCSI

  Serielle SCSI-Varianten sind in Vorbereitung.

Tabelle 5.4 enthält gebräuchliche SCSI-Varianten.

	Transfer [MB/s]	Bits	Länge [m]	Geräte max.	Übertragung
SCSI-1	5	8	6	7	*single-ended*
Fast SCSI	10	8	3	7	*single-ended*
Fast/Wide SCSI	20	16	3	15	*single-ended*
Ultra/Narrow	20	8	3	4	*single-ended*
	20	8	1,5	7	*single-ended*
Ultra/Wide	40	16	3	4	*single-ended*
	40	16	1,5	7	*single-ended*
Ultra2/Wide	80	16	12		differenziell

Tabelle 5.4: Gebräuchliche SCSI-Varianten

- **USB-Bus (*Universal serial bus*)**

  Der USB-Bus hat folgende Eigenschaften:

  - Datenraten: 1,5 MBit/s oder 12 MBit/s, 500 MBit/s in Vorbereitung
  - Serieller Bus mit differenzieller Übertragung
  - Stromversorgung für angeschlossene Geräte
  - **3 Übertragungsmodi:**

    \- max 64 Byte, garantiert pro Zeitintervall

    \- *bulk transfer*, nur wenn Bandbreite vorhanden.

    \- isochroner Transfer, zeitgenau, keine Fehlerbehandlung
  - **NRZI-Kodierung der Daten:**

    '1' zu übertragen: Pegel bleibt erhalten

    '0' zu übertragen: Wechsel des Pegels

    Bit stuffing (Einfügen von Nullen, um Takt erzeugen zu können)
  - Hierarchisch aufgebautes *Hub*-System

- **FireWire (IEEE Std. 1394)**

  FireWire ist ein weiterer Standard für Peripherie-Busse.

5.2 Datenübertragung

Nach der Behandlung der Datenübertragung, v.a. der schnellen Datenübertragung innerhalb eines Rechners, wollen wir uns als nächstes mit der langsameren Datenübertragung (DÜ), insbesondere zu externen Netzen, Druckern, Modems, usw. beschäftigen.

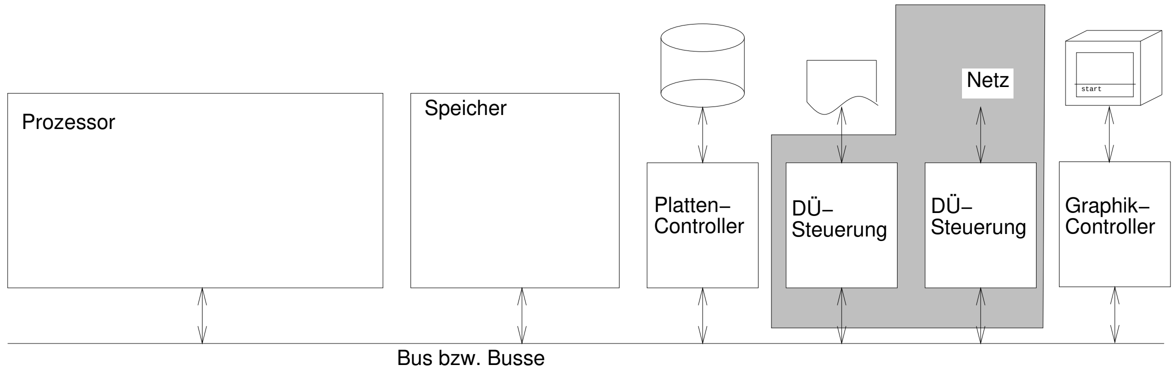


Abbildung 5.36: Datenübertragung von einem Rechner aus

5.2.1 Parallel-Schnittstellen

Zu weiter entfernt liegender Peripherie werden Daten in der Regel nicht als ganze Worte, sondern **Byte-parallel** oder sogar **bitseriell** übertragen. Betrachten wir zunächst die Byte-parallele Übertragung. Hierfür gibt es spezielle Hardware-Bausteine, die durch Konfigurierung auf unterschiedliche Spezifikationen der E/A-Signale angepaßt werden können. Handelsübliche Bausteine werden z.B. als PIA (*peripheral interface adapter*) bezeichnet und bieten 2 oder 3 Schnittstellen von jeweils 8 Bit. Meist kann die Übertragungsrichtung für jedes Bit einzeln konfiguriert werden. Diese Konfiguration geschieht, indem die Konfigurationsdaten an spezielle E/A-Adressen geschrieben werden. Bähring beschreibt als Beispiel den sog. PPI-Baustein 8255 von Intel. Dieser Baustein kann zur Realisierung unterschiedlicher Schnittstellen eingesetzt werden.

Eine standardisierte Schnittstelle ist die Centronics-Schnittstelle. Diese wurde v.a. als reine Drucker-Schnittstelle konzipiert. Aus diesem Grund ist es (zumindest in der ursprünglichen Form) eine reine Ausgabe-Schnittstelle. Es werden 8 Datenleitungen und einige Steuerleitungen benutzt (vgl. Abb. 5.37). Mittels der Steuerleitungen kann erreicht werden, dass keine Informationen aufgrund fehlender Empfangsbereitschaft verloren gehen (**Hardware-Flußkontrolle**) .

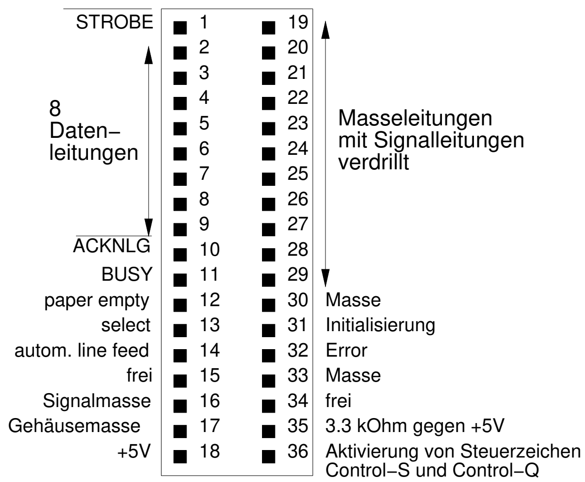


Abbildung 5.37: Signale der Centronics-Schnittstelle und deren Pinbelegung

Das Timing dieser Schnittstelle zeigt die Abb. 5.38.

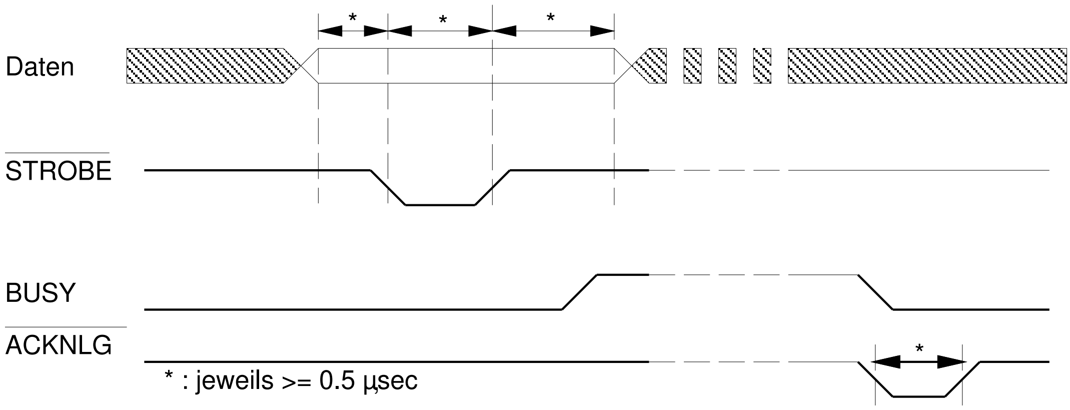


Abbildung 5.38: Timing der Centronics-Schnittstelle

Die Gültigkeit der Daten wird vom Sender das Strobe-Signal anzeigt, wobei jeweils gewisse Mindestzeiten einzuhalten sind. Gemäß diesem Timing müßten Datenübertragungsraten von knapp $1/2\mu$sec pro Zeichen = 500 kByte/sec erreichbar sein. Es handelt sich damit um eine recht schnelle Schnittstelle. Ein großer Vorteil der Centronics-Schnittstelle ist ihre weitgehende Normung und eine damit einhergehende Gerätekompatibilität.

Einschränkungen der ursprünglichen Centronics-Schnittstelle liegen v.a. in der Unidirektionalität der Datenübertragung sowie in fehlenden Möglichkeiten, Befehle zu übertragen und mehrere Geräte anzusprechen. Diese Nachteile sind mit den Erweiterungen ECP (*enhanced capability port*) und EPP ausgeräumt [Sti94].

Eigenschaften der beiden Erweiterungen sind die folgenden:

- **EPP** (*enhanced parallel port*):

  Die Bidirektionalität wird mit einer kleinen Hardware-Erweiterung realisiert. Ein Nachteil ist, dass *handshaking* in diesem Fall hardwaremäßiges aktives Warten (*busy waiting*) erfordert.

- **ECP** (*enhanced capability port*):

  Handshaking wird in diesem Fall über *strobe*, *busy*, *ack* und *autofd* realisiert. Ein 16-Byte FIFO-Puffer hilft, das Verlieren von Zeichen zu vermeiden. Die vorhandene Interrupt- und DMA-Fähigkeit sind einem Mehrprozessbetrieb angemessen. Die real erzielbare Übertragungsrate kann 2 MByte/s überschreiten.

5.2.2 Asynchrone serielle Schnittstellen

Asynchrone serielle Schnittstellen werden v.a. zum Anschluß von Terminals, aber auch zum Anschluß von Druckern, Fax-Geräten u.a.m. an einen Rechner benutzt. Grundlage der üblichen asynchronen Übertragung ist das in Abb. 5.39 dargestellte Format.

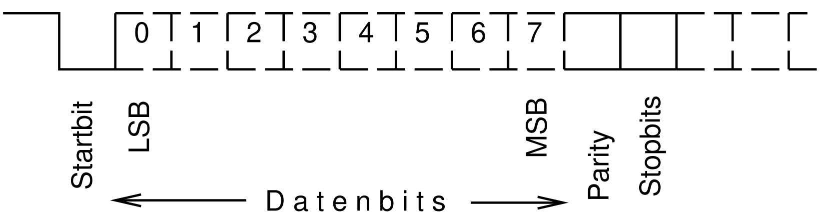


Abbildung 5.39: Zeichenrahmen bei der asynchronen Datenübertragung

Die einzelnen Bits werden mit unterschiedlichen Spannungen dargestellt. Vielfach sind dies -12 Volt und +12 Volt (mit Toleranzbereichen). Aber auch 0 Volt und 5 Volt (mit Toleranzbereichen) sind im Gebrauch. Die Übertragung jedes Zeichens beginnt mit einem Impuls (Startbit) zur Synchronisierung zwischen Sender

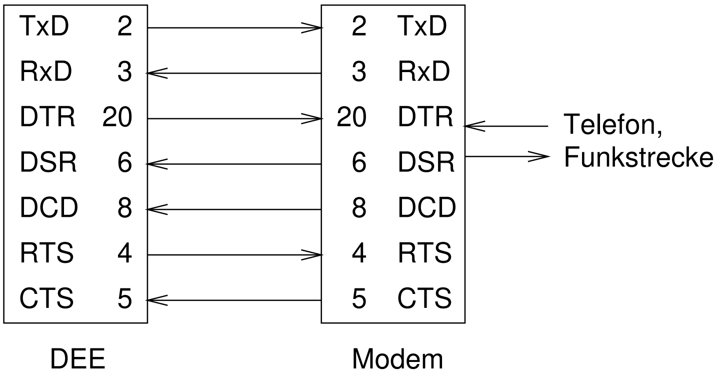


Abbildung 5.40: Anschluß einer DEE an ein Modem

und Empfänger. Anschließend folgt eine Anzahl von 5-8 Datenbits, wobei Sender und Empfänger auf gleiche Zahl von Bits eingestellt sein müssen. Das am wenigsten signifikante Bit wird zuerst gesendet (little endian-Konvention). Anschließend folgt ggf. ein Paritätsbit (gerade oder ungerade Parität). Auch über dieses müssen Sender und Empfänger die gleichen Annahmen machen. Schließlich folgen noch 1-2 Stopbits. Dies sind eigentlich keine Bits. Es wird lediglich spezifiziert, wie lang die Pause bis zum nächsten Zeichen mindestens sein muß.

Mit den üblichen seriellen Schnittstellen kann man auf kurze Entfernungen Übertragungsleistungen bis knapp oberhalb von 100 kbit/s erreichen, moderne Geräte vorausgesetzt (1996).

Im folgenden wollen wir die weitverbreitete V.24-Schnittstelle (auch bekannt als *RS 232-Standard*) besprechen.

Die ebenfalls benutzte Norm RS-422 unterscheidet sich von der hier dargestellten Norm RS-232 durch die Verwendung differenzieller Signale.

Gemäß dem V.24-Standard ist die Verwendung der asynchronen Übertragung zwischen einem Daten-Endgerät (DEE, Rechner oder Terminal) und einem Modem (Modulator/Demodulator zur Umsetzung in Telefonsignale) normiert. Danach werden u.a. die Signale nach Abb. 5.40 benötigt (Masseverbindungen nicht gezeigt).

In beide Richtungen können gleichzeitig über die Leitungen TxD und RxD Daten gesendet werden. Einen solchen Betrieb nennt man **Voll-Duplex-Betrieb** . Beim **Halb-Duplex-Betrieb** können in beide Richtungen Daten übertragen werden, aber nur in eine Richtung zur Zeit. Beim **Simplex-Betrieb** kann überhaupt nur in eine Richtung übertragen werden.

Die übrigen Signale haben die folgende Funktion:

DTR *data terminal ready*

Mit diesem Ausgangssignal informiert das DEE seinem Kommunikationspartner, dass es betriebsbereit ist.

DSR *data set ready*

Mit diesem Ausgangssignal informiert der Kommunikationspartner dem DEE, dass er betriebsbereit ist.

RTS *request to send*

Mit diesem Ausgangssignal informiert das DEE seinem Kommunikationspartner, dass es Daten senden möchte.

CTS *clear to send*

Mit diesem Ausgangssignal informiert der Kommunikationspartner dem DEE, dass er Daten empfangen kann.

DCD *data carrier detect*

Mit diesem Ausgangssignal informiert der Kommunikationspartner dem DEE, dass er eingangsseitig ein analoges Trägersignal (z.B. auf der Telefonleitung) empfängt.

Abb. 5.41 zeigt die Zuordnung dieser Signale zu den Pins der genormten 25- und 9-poligen Stecker. Für weitere Pins des 25-poligen Steckers wurden Signale normiert, diese werden aber in der Regel nicht benutzt.

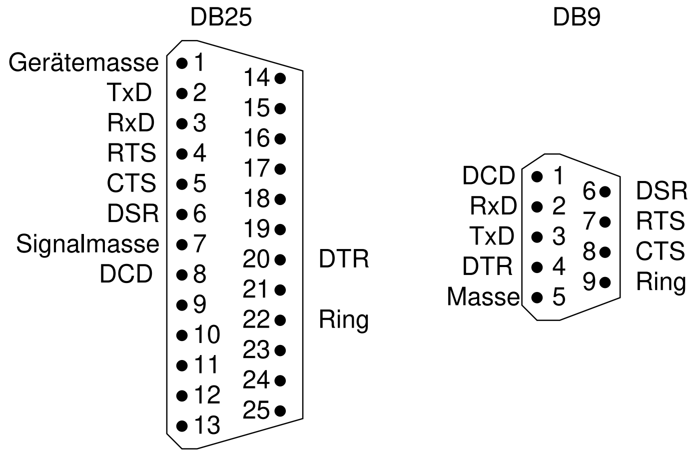


Abbildung 5.41: Steckerbelegung der V.24-Schnittstelle

Ein ganze Reihe von Problemen tritt dann auf, wenn die V.24-Schnittstelle für Anwendungen genutzt wird, für die sie eigentlich nicht gedacht war, insbesondere bei der Kommunikation zwischen Rechner und Terminal, die im Sinne der Datenfernübertragung (für die V.24 entwickelt wurde) beide DEEs darstellen.

Würde man bei den Datenleitungen Pins mit gleicher Nummer verbinden, so würde man Ausgang mit Ausgang und Eingang mit Eingang verbinden. Um dies zu vermeiden, werden in üblichen Kabeln die Pins 2 und 3 über Kreuz miteinander verbunden. Schließt man allerdings mehrere Kabel in Reihe, so muß bei einer geraden Anzahl von Kabeln natürlich gleiche Pins untereinander verbinden. Man braucht dann neben vertauschenden Kabeln reine **Verlängerungskabel**, bei denen (9 oder alle 25) korrespondierenden Pins miteinander verbunden sind.

Die oben angeführten 5 Kontrollsignale erlauben im Prinzip eine Hardware-**Flußkontrolle**, d.h. Sender und Empfänger können ihre jeweilige Bereitschaft über spezielle Signale mitteilen und so verhindern, dass Daten verloren gehen. Für diesen Zweck ist die folgende Verkabelung nach Abb. 5.42 notwendig.

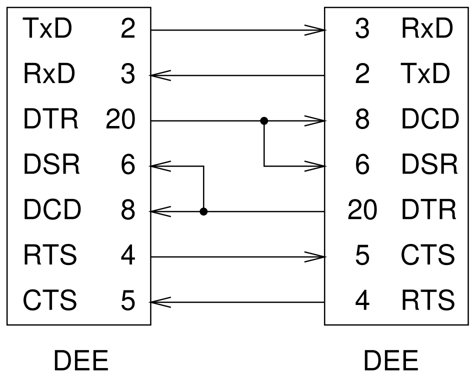


Abbildung 5.42: Verkabelung bei Anschluß zweier DEE einschließlich Hardware-Flußkontrolle

Über die Pins 6,8 und 20 signalisieren sich die Partner, dass sie jeweils (betriebsbereit) eingeschaltet sind. Die Brücke zum Pin 8 suggeriert einem DEE, dass ein Signalträger empfangen wird sobald der Kommunikationspartner auch nur eingeschaltet ist. Die Pins 4 und 5 erlauben es, Übertragungswunsch und Empfangsbereitschaft zu signalisieren. Leider fehlt eine exakte Definition der Unterscheidung zwischen "betriebsbereit" und "sende-" bzw. "empfangsbereit".

Die Verschaltung der Pins 4 und 5 wird gelegentlich fortgelassen (Apple?), denn die Betriebsbereitschaft kann schon über die Pins 6,8 und 20 angezeigt werden (→ eine weitere Kabelvariation).

Häufig will man mit einer geringeren Anzahl an Kabeln auskommen oder der Rechner stellt die Kontrollsignale nicht zur Verfügung. In diesen Fällen werden die Kontrollsignale direkt an den DEE miteinander so verbunden, dass eine ständige Bereitschaft des Kommunikationspartners suggeriert wird (vgl. Abb. 5.43).

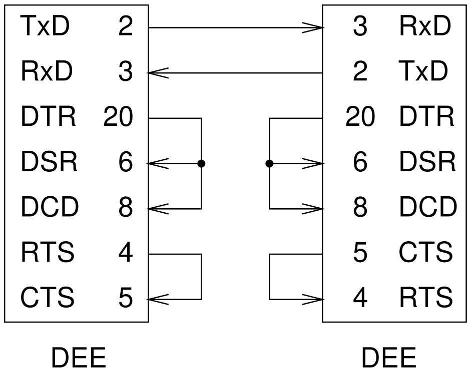


Abbildung 5.43: Verkabelung bei Anschluß zweier DEE ohne Hardware-Flußkontrolle

Die Flußkontrolle kann jetzt nicht mehr über die Hardware-Signale erfolgen. Eine Alternative bildet jetzt das Senden von speziellen Zeichen, die dem Partner melden sollen, dass man zur Kommunikation nicht mehr bzw. wieder bereit ist. Üblich sind hier die ASCII-Zeichen Control-S und Control-Q (auch als XOn und XOff bekannt). Diese Zeichen dürfen dann natürlich nicht mehr in den zu übertragenden Daten vorkommen. Insbesondere bei Binärinformationen bereitet das Schwierigkeiten.

5.2.3 Synchrone serielle Schnittstellen

Bei **synchronen seriellen Schnittstellen** wird die Synchronisation zwischen Sender und Empfänger nicht vor der Übertragung jedes <u>Zeichens</u>, sondern vor der Übertragung eines <u>Blocks</u> hergestellt. Dazu werden einem Block einige zwischen Sender und Empfänger verabredete Synchronisationszeichen vorangestellt. Abgeschlossen wird der Block ebenfalls durch einige verabredete Zeichen.

Wichtig ist jetzt die Frage: Dürfen diese verabredeten Zeichen selbst wieder in dem Datenblock vorkommen? Je nachdem, wie man diese Frage beantwortet, kommt man zu verschiedenen Methoden:

1. Die zeichenorientierte Übertragung

   Bei der zeichenorientierten Übertragung benutzt man das Format nach Abb. 5.44.

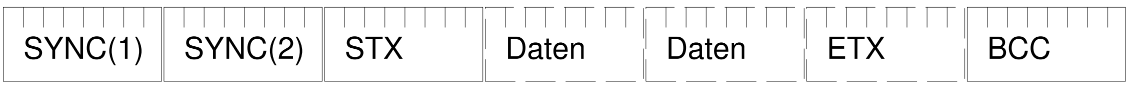
   

   Abbildung 5.44: Zeichenorientierte Übertragung

   Die Übertragung beginnt mit dem Aussenden von einem oder mehreren Synchronisationszeichen SYNC, die vom Empfänger benötigt werden, um sich auf den Sender taktmäßig einzustellen. Das Steuerzeichen STX (*start of text*) kennzeichnet dann den Anfang des eigentlichen Datenblocks. Dieser wird durch das Steuerzeichen ETX (*end of text*) abgeschlossen. Schließlich folgt meist noch ein Prüfzeichen BCC, welches z.B. eine Parität enthält. Zumindest das Zeichen ETX darf in dem Datenblock selbst nicht vorkommen. Dies bereitet Probleme bei der Übertragung von Binärinformation.

2. Die bitorientierte Übertragung

   Bei der bitorientierten Übertragung benutzt man das Format nach Abb. 5.45.

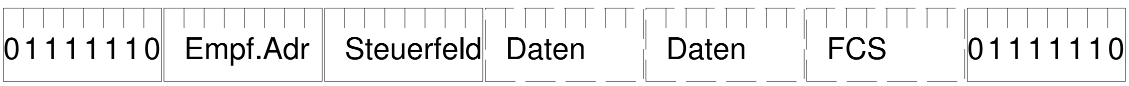
   

   Abbildung 5.45: Bitorientierte Übertragung

   Die Übertragung beginnt mit dem speziellen Flag "01111110". Auf dieses folgt die Empfangsadresse und ein Steuerfeld. Anschließend werden die Daten übertragen, dann ein Prüfzeichen und schließlich erneut das Flag "01111110". Hier könnte wieder das Problem entstehen, dass dieses Flag in den Daten nicht vorkommen dürfte. Hier hilft jetzt ein Trick: Innerhalb des Datenblocks wird vom Sender nach fünf Einsen stets eine Null eingefügt (engl. *bit stuffing*). Findet der Empfänger im Anschluß an fünf aufeinanderfolgende Einsen eine Null, so entfernt er diese bedingslos. Findet er eine weitere Eins, so handelt es sich um das Ende des Datenblocks. Dieses Verfahren heißt **bitorientiert**, weil die übertragene Nachricht durch das Einfügen von Nullen nicht mehr unbedingt eine ganze Anzahl von Bytes enthält. Bei einer ohnehin bitseriellen Übertragung und spezieller Hardware in Sendern und Empfängern ist das aber kein Problem.

Das beschriebene Format ist Teil der Normung des HDLC-Übertragungsprotokolls (*high-level data link control*), welches seinerseits wiederum Eingang in die ISDN-Norm gefunden hat.

5.2.4 Lokale Netzwerke

Die Vernetzung von Rechensystemen und anderen **Endgeräten** spielt z.Zt. eine stark wachsende Rolle. Eine gute Übersicht über das Gebiet gibt das Buch von Stallings [Sta94]. Eine detaillierte Behandlung dieses Themas erfolgt auch in der Stammvorlesung "Rechnernetze und verteilte Systeme". Wir geben hier nur einen kurzen Abriß elementarer Grundlagen.

5.2.4.1 Ethernet

Die Datenübertragung nach dem Ethernet-Schema bzw. nach dem weitgehend identischen Standard IEEE 802.3 ist weit verbreitet.

Das Grundprinzip, aus dem auch der Name *Ethernet* abgeleitet ist, ist das der verteilten Kontrolle: jeder Teilnehmer kann senden, solange ein kein anderes Signal auf der Leitung erkennt. Werden Zugriffskonflikte erkannt, so wird ggf. eine laufende Übertragung abgebrochen und nach einer Wartezeit wiederholt. Dieses Prinzip hat auch zu dem Namen *carrier sense multiple access/collision detect*–Verfahren (**(CSMA/CD)**) geführt. *Carrier sense* bedeutet: es wird geprüft, ob ein anderer Teilnehmer einen Träger für eine Übertragung sendet. Kollisionen werden erkannt (und nicht etwa vorab vermieden).

Ethernet-Varianten:

- **10 Mbits/s**

  Das physikalische Anschlußschema des 10 MBit-Ethernet basiert auf einem durchgehenden Koaxialkabel, auf das alle beteiligten Stationen zugreifen (siehe Abb. 5.46).

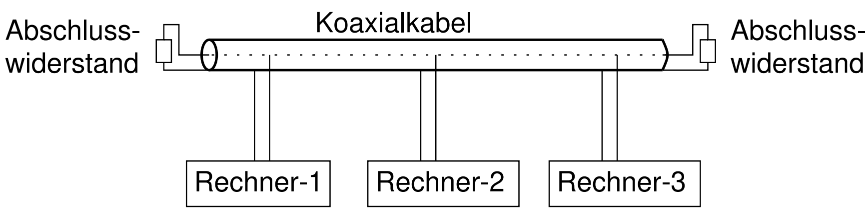


Abbildung 5.46: Physikalische Ethernet-Verbindung mit Abschlusswiderständen

  Der Abschluß mit dem Wellenwiderstand ist erforderlich, um Leitungsreflexionen zu vermeiden. Er erfolgt an den beiden Enden des Koaxialkabels. Die einzelnen Rechneranschlüsse dürfen ebenfalls keine Reflexionen erzeugen. Entsprechend muss der Anschluß erfolgen. Es gibt zwei Varianten:

  - **10BASE5, ThickWire**:

    Urtyp des Ethernet Kabels, für 10 MBit/s, max. 500 m, mind. 2,5m zwischen Anschlüssen; separate Transceiver an den Anschlüssen (siehe Abb. 5.47); Kabel wird an Anschlussstelle durchbohrt; Anschluss an AUI-Buchse des Rechners.

  - **10BASE2, ThinWire, Cheapernet:**

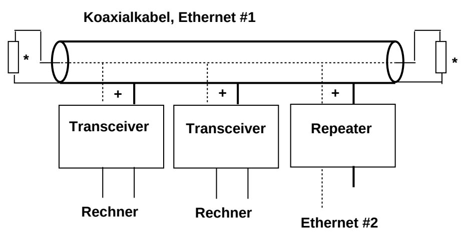


Abbildung 5.47: 10BASE5; +: kurze Verbindung zur Rechner; *: Abschlusswiderstand

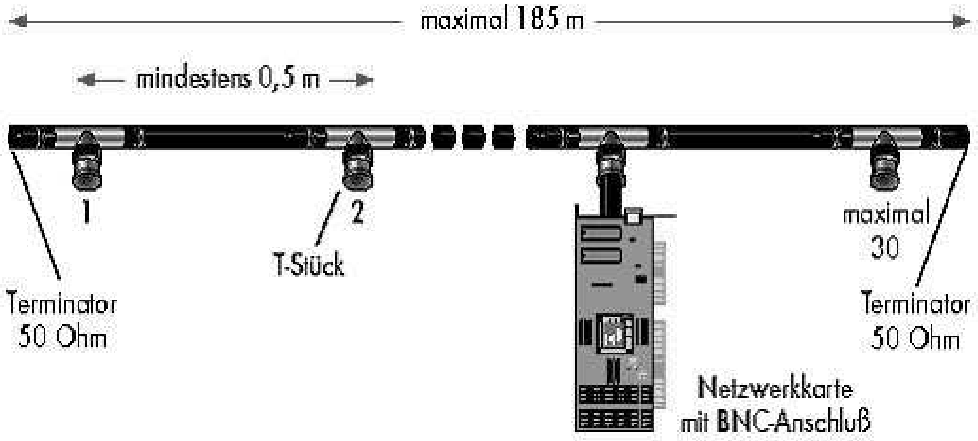


Abbildung 5.48: 10Base2: Detail des Rechneranschlusses (©c't)

Dünneres Kabel, gedacht für 10 MBit/s, max. 185 m lang, max. 30 Stationen, mind. 50cm zwischen den Anschlüssen; Kabel wird an Anschlüssen aufgetrennt; Kurze Verbindung zwischen T-Stück/Rechner (siehe Abb. 5.48).

Daher sind zum Rechner hin **zwei** Kabel erforderlich, falls dieser einen eingebauten Transceiver besitzt. Unbenutzte Anschlüsse werden mittels eines Verbindungskabels überbrückt.

Die maximale Datenrate (bei einem Sender) beträgt $\approx$ 10 MBit/sec. Die effektive Datenrate ist stark von der Zahl der Konflikte abhängig. Sie ist $\geq$ 100 kBit/s.

– **10BASE-T, TwistedPair:**
  Verdrillte Leitungen, max. 100m lang, i.d.R. Punkt-zu Punkt-Verkabelung zum *hub*. Gesamtkonzept: **strukturierte Verkabelung:** Primärverbindung zwischen Gebäuden, Sekundärverbindung zwischen Etagen und Tertiärverb. zwischen Etagenhubs und Endgeräten. UTP (*unshielded twisted pair*) & SUTP-Kabel, Anschluss über RJ-45-Stecker (siehe Abb. 5.49).

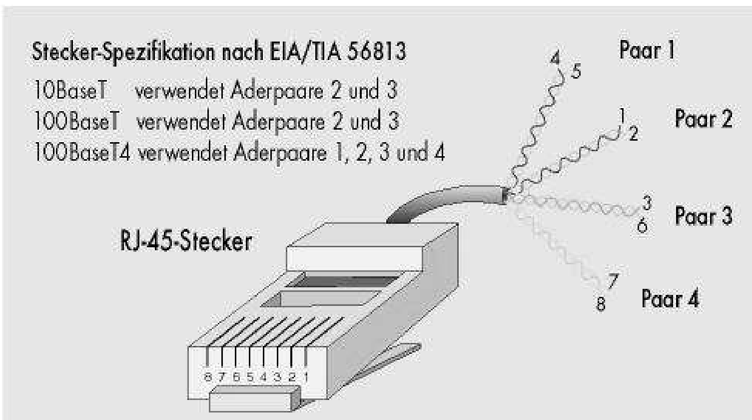


Abbildung 5.49: RJ-45-Stecker (©c't, 1999)

– **10BASE-F** (Ethernet-Lichtleiter-Norm):

Dies ist eine (überholte) Ethernet-Lichtleiter-Norm.

- **100 Mbit/s Ethernet (*Fast Ethernet*)**
  - **100BASE-T4:**
    Kabel der Kategorie 3 (8-polige Telefonkabel); Aufteilung der Daten auf 4 Aderpaare zur Reduktion der Übertragungsfrequenzen (Störstrahlung!); Halbduplex-Modus
  - **100BASE-Tx:**
    Punkt-zu Punkt-Verkabelung; Kabel der Kategorie 5 (4-UTP/ STP-Kabel);
    Aufteilung der Daten auf Aderpaare und 3-wertige Logik zur Reduktion der Übertragungsfrequenzen (Störstrahlung!);
  - **100BASE-Fx:**
    Sternförmige Glasfaserverkabelung
- **1000 Mbit/s, *(Gigabit Ethernet)***
  CSMA/CD erfordert das die Übertragungszeit eines Paketes größer ist als die Laufzeit der Pakete auf den Leitungen. Andernfalls könnte der Sendevorgang abgeschlossen sein, bevor ein möglicher Konflikt erkannt wird. Daher muss es eine minimale Größe der Pakete geben. Aus diesem Grund wird die zulässige Mindestlänge beim Gigabit-Ethernet von 64 Byte auf 512 Byte erhöht. Notfalls müssen Pakete durch Füllbytes ergänzt werden.

Abbildung 5.50: Paketerweiterung (*extension*) beim Gigabit-Ethernet

  - **1000BaseLx** Lichtleiter, Segmentlänge 2-550 m. Glasfaserkabel zum Arbeitsplatz recht teuer.
  - **1000BaseCx** (IEEE 802.3z): *Twisted pair*-Kabel mit $Z_0 = 150\Omega$. Max. 25 m. 2 *Twinax*- oder 1 *Quad*-Kabel (siehe Abb. 5.51) pro Verbindung.

Abbildung 5.51: *Twinax*- und*Quad*-Kabel (©c't)

  - **IEEE 802.ab**:
    1000 Mbit/s über Kategorie 5-Kabel bis $\leq$ 100 m. Dies ist die kostengünstigste Variante, erlaubt sie doch den Übergang von 1000 MBit- auf Gigabit-Ethernet unter Verwendung der vielfach installierten Kategorie-5 Kabel.

5.2.4.2 Token-Ring-Netze

Token-Ring-Netze basieren auf der Idee, dass innerhalb eines Rings eine spezielle Signalkombination zur Anzeige der Verfügbarkeit für Datenübertragungen genutzt wird. Diese Signalkombination, *token* genannt, läuft im Ruhezustand auf dem Ring um (siehe Abb. 5.52).

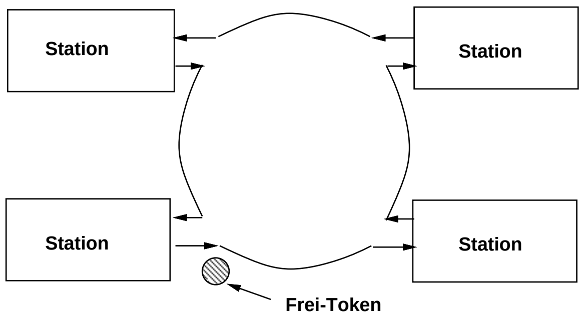


Abbildung 5.52: Ruhezustand

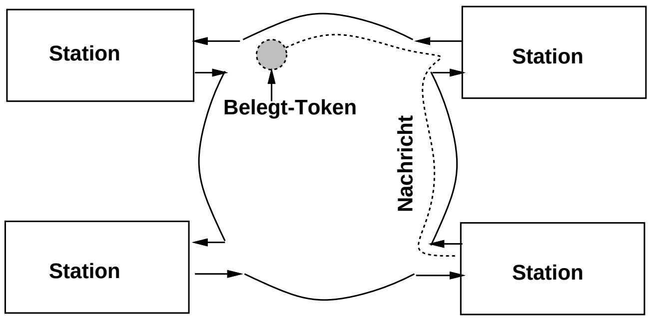


Abbildung 5.53: Erzeugung einer Nachricht

Stationen mit Sendewunsch wandeln ein "Frei-Token" in ein "Belegt-Token" um und fügen die Empfängeradresse sowie die zu übertragende Nachricht an (Abb. 5.53).

Die Nachricht wird von allen Stationen, die nicht selbst übertragen, weitergeleitet. Die Verzögerung der Nachricht pro Station ist minimal (z.B. nur 1 Bit/Station). Der durch die Adresse bestimmte Empfänger übernimmt eine Kopie der Nachricht. Eine beim Sender unverfälscht eintreffende Nachricht dient als Bestätigung der korrekten Übertragung (siehe Abb. 5.54).

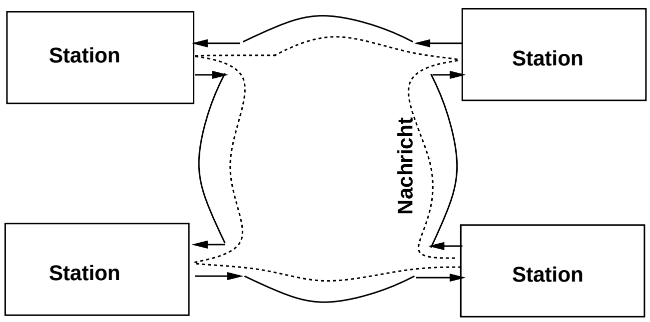


Abbildung 5.54: Vollständiger Umlauf

Nach Beendigung der Übertragung generiert der Absender ein "Frei-Token". Die physikalisch nächste Station besitzt die höchste Priorität bezüglich weiterer Übertragungen.

Vorteile des Konzepts sind die folgenden:

- Selbst bei starker Belastung bleibt bei n Stationen für jede Station 1/n-tel der Übertragungsrate verfügbar. Es tritt kein sog. *thrashing*, d.h. eine Reduktion der effektiven Übertragungsleistung bei zunehmender Belastung auf.
- An jeder Station erfolgt ein Auffrischen der Signale. Daher ist nur die Ausdehnung zwischen **benachbarten** Stationen begrenzt, nicht die Gesamtausdehnung.
- Das Quittungsprinzip ist einfach.

Nachteile gibt es auch:

- Die Installation ist etwas aufwendiger als beim Ethernet.
- Beim Ausfall eines Transceivers ist das Netz vollständig unterbrochen. → besonders zuverlässige Transceiver, Überbrückung durch Relais falls Rechner ausgeschaltet oder Defekt erkannt wurde, Doppelring.
- Die Verbreitung ist im Verhältnis zu Ethernet geringer.

In der bislang erläuterten Form wurde das Konzept beim Token-Ring der Apollo-Workstations zuerst realisiert (seit ≈ 1980; 12 MBit/s ; max. Abstand der Rechner 2 km). Beim jüngeren IBM-Token Ring, der auch als IEEE-Standard 802.5 definiert wurde, hat man die Möglichkeit vorgesehen, Nachrichten mit unterschiedlichen Prioritäten auszustatten. Eine Übertragung durch eine Station darf damit nur dann erfolgen, wenn die Priorität der Nachricht größer ist als die des momentanen Frei-Tokens. Die Priorität des Frei-Tokens bestimmt sich u.a. aus Informationen, die zusammen mit der vorherigen Nachricht übertragen werden und diese wiederum können durch Stationen mit Sendewünschen hoher Priorität beeinflußt werden [Sta94]

Weiterhin gibt es in der IBM-Version eine *early token release*-Option, die es gestattet, von der sendenden Station bereits am Ende der gesendeten Nachricht ein neues Frei-Token erzeugen zu lassen. Auf diese Weise kann es insbesondere bei geographisch weit verteilten Stationen gleichzeitig mehrere Übertragungen geben.

Schließlich gibt es in dieser Version einen zentralen Rangier-Verteiler, zu dem der Ring immer wieder zurückgeführt wird. Bei Unterbrechung des Rings kann der unterbrochene Teil so überbrückt werden.

5.2.4.3 FDDI

FDDI (Fiber Distributed Data Interface) ist von der ANSI (American National Standards Institute) als Standard für optische Netze vorgeschlagen worden. Die Datenrate beträgt 100 MBaud [Abe91]. Es werden zwei unabhängige Ringe verwendet. Nicht alle Stationen haben Zugang zu beiden Ringen. Fällt ein Ring aus, wird über den anderen rekonfiguriert,

FDDI unterscheidet zwischen verschiedenen Prioritäten von Nachrichten. Damit ist es möglich, wichtige Nachrichten rasch über ein Netz zu transportieren, welches gerade große Mengen von Daten niedrigerer Priorität übermittelt (File-Backup).

5.2.5 Weitverkehrs-Netze (WANs)

Interessant ist es, sich für LANs die Verhältnisse zwischen der Sendezeit einer Nachricht und ihrer Laufzeit auf dem Kabel zu überlegen. Aufgrund der üblichen Gegebenheiten (nicht zu kurze Nachrichten, nicht zu lange Kabel) sind die Kabel **bei lokalen Netzwerken** während der Übertragung zu einem großen Teil der Zeit tatsächlich ausgelastet. Bei nicht-lokalen Netzwerken liegen die Verhältnisse anders. Hier möchte man bei einem Ring z.B. mehrere Frei-Token erlauben, um die Kapazität auszunutzen. Auch kann die Ethernet-Technik nicht mehr genutzt werden, weil es auf Grund der Leitungslaufzeiten zu lange dauert, bis Übertragungskonflikte erkannt werden. Dieses sind die Ursachen dafür, dass für *wide-area-networks* (WANs) andere Techniken entwickelt werden müssen. Der z.Zt. (1993) aktuelle Stand dieser Techniken ist z.B. in [DSTG93] und [Ebe93] beschrieben worden. Eine derzeit im Aufbau befindliche Technik ist die ATM (asynchronous transfer mode)-Technik, die eine Vermittlung von Paketen mit unterschiedlichen Prioritäten gestattet (siehe z.B. [Fis93]).

5.2.6 Das ISO-Referenzmodell

Das ISO-Referenzmodell bildet den Standard für die Entwicklung und die Klassifikation von DÜ-Protokollen. Stichwortartig können die Ebenen wie folgt charakterisiert werden:

1. Die physikalische Ebene

   Die Normung umfaßt:

- Die Übertragung einzelner Bits
- Benutzte Spannungen und Ströme
- Benutzte Stecker
- Benutzte Daten- und Kontroll-Leitungen
- Benutzte Datenraten

Normen z.B. V.24, X.21, RS 432, RS 422, Satellitenverb.

2. Die Leitungsebene (Sicherungsschicht)

   Aufgabe der Leitungsebene ist die Bereitstellung eines fehlerfreien logischen Kanals zwischen benachbarten Kommunikations-Einheiten

   Die Normung umfaßt:

   - Die Fehlerkorrektur (z.B. mittels zyklischer Codes)
   - Die Flußkontrolle (Abstimmung von Lese- und Schreibgeschwindigkeiten, siehe z.B. X-ON/X-OFF-Protokoll)
   - Ausstattung der Blöcke mit Folgenummern
   - Sortieren der Blöcke gemäß Folgenummern
   - Synchronisation bei Übertragung verschiedener Daten über eine Leitung

   Normung z.B. HDLC (In den Daten wird im Sender nach 5 Einsen eine 0 eingefügt und beim Empfänger wieder entfernt).

3. Die Netzwerkebene

   Aufgabe der Netzwerkebene ist die Kommunikation zwischen Endsystemen.

   Die Ebene enthält die Auswahl der notwendigen Zwischenstationen (das sog. Routing) sowie die Absicherung gegen ausfallende Zwischenstationen und die Überlastung von Zwischenstationen. Schließt Verbindungsaufbau und -abbau ein.

   Norm: X.25 - umfaßt Ebenen 1-3

4. Die Transportebene

   Aufgabe der Transportebene ist die Kommunikation zwischen Prozessen.

   Vergleichbar mit der Interprozeßkommunikation. Die Funktionen umfassen:

   - Die Adressierung von Hosts und Prozessen
   - Auf- und Abbau von Verbindungen
   - Behandlung von Host-Abstürzen
   - Multiplexen von Daten verschiedener Prozesse
   - Gateway-Funktion zwischen verschiedenen Netzen

5. Die Sitzungsebene

   Aufgabe ist die Bereitstellung benutzerorientierter Dienste wie z.B. die Definition von Wiederaufsetzpunkten, Bericht über Protokoll-Fehler, Abstimmung der Partner über benutzte Funktionen.

6. Die Darstellungsebene

   Diese Ebene umfaßt Verabredungen über den Zeichensatz, die Codierung, die Darstellung von Daten auf dem Bildschirm oder Drucker, Umwandlung von File-Formaten, Verschlüsselung

7. Die Anwendungsebene

   Diese Ebene umfaßt direkt sichtbare Dienste wie: Electronic Mail, Remote Login, Remote Job Entry, File Transfer

5.3 Sicherung von Daten mit zyklischen Codes

5.3.1 Bildung von CRC-Zeichen

Zur Absicherung von zu übertragenden oder abzuspeichernden Informationen werden meist bestimmte Codes eingesetzt, bei denen man eine bestimmte Menge von Fehlern erkennen kann [PB61, HQ89]. Wir werden diese hier zunächst allgemein, unabhängig von Plattenspeichersystemen, betrachten.

Zu übertragen sei eine Nachricht, die in k Bits kodiert sei. Durch Übertragung mit einem Code von mehr als k Bits ist ein Schutz gegen fehlerhafte Übertragung möglich. Wir wollen annehmen, dass insgesamt n Bits übertragen werden. Falls der gewählte Code in den k Bits mit der Ausgangsnachricht übereinstimmt, sprechen wir von einem **systematischen Code**:

$$\begin{array}{cc} \text{Nachricht} & \text{Prüfbits} \\ \leftarrow - - - - - \rightarrow & \leftarrow - - - - - \rightarrow \\ k \text{ Bits} & n-k \text{ Bits} \end{array}$$

Für die weitere Theorie betrachten wir die einzelnen Bits a_i der Nachricht und der Prüfbits als Koeffizienten eines Polynoms G bzw. R. Wir betrachten also das **Nachrichtenpolynom**:

$$G = \sum_{i=0}^{k-1} a_i * 2^i \tag{5.1}$$

Statt der speziellen Basis 2 betrachten wir im folgenden das Polynom zur allgemeinen Basis x:

$$G(x) = \sum_{i=0}^{k-1} a_i * x^i \tag{5.2}$$

Den übertragenen Code können wir, falls es ein systematischer Code ist, durch ein **Codepolynom** der Form

$$F(x) = G(x) * x^{n-k} + R(x) \tag{5.3}$$

darstellen. Darin stellt $R(x)$ die Prüfbits dar.

Wir fordern nunmehr, dass für alle benutzten Codierungen der Nachricht $F(x)$ durch ein **Generator-Polynom** $P(x)$ teilbar sein soll, d.h. es soll gelten:

$$F(x) = P(x) * Q(x) = G(x) * x^{n-k} + R(x) \tag{5.4}$$

Wir nennen derartige Codes **zyklische Codes**.

Wir wollen im folgenden weiter voraussetzen, daß wir alle Operationen modulo 2 durchführen. Als Folge davon brauchen wir zwischen + und − nicht zu unterscheiden. Aus der letzten Gleichung folgt also:

$$R(x) = G(x) * x^{n-k} - P(x) * Q(x) \tag{5.5}$$

Damit sind die Prüfbits $R(x)$ gleich dem Rest der Division von $G(x)$ (nach links verschoben um x^{n-k}) durch $P(x)$ (der Rest r der Division von g durch p ist allgemein definiert durch $r = g - p * q$ mit $r < q$).

Wir wollen den Rechengang an einem praktischen Beispiel deutlich machen:

$$
\begin{array}{l}
(1x^6 + 0x^5 + 1x^4 + 0x^3 + 1x^2 + 0x^1 + 1x^0) : (1x^3 + 1x^2 + 0x^1 + 1) = \\
\;1x^6 + 1x^5 + 0x^4 + 1x^3 \qquad\qquad\qquad\qquad\qquad\qquad 1x^3 + 1x^2 + 0x + 1 \\
\;-------------- \\
\;0x^6 + 1x^5 + 1x^4 + 1x^3 + 1x^2 \\
\qquad\;\; 1x^5 + 1x^4 + 0x^3 + 1x^2 \\
\qquad\;\; -------------- \\
\qquad\;\; 0x^5 + 0x^4 + 1x^3 + 0x^2 + 0x^1 \\
\qquad\qquad\quad 0x^4 + 0x^3 + 0x^2 + 0x^1 \\
\qquad\qquad\quad -------------- \\
\qquad\qquad\qquad\quad 1x^3 + 0x^2 + 0x^1 + 1x^0 \\
\qquad\qquad\qquad\quad 1x^3 + 1x^2 + 0x^1 + 1x^0 \\
\qquad\qquad\qquad\quad -------------- \\
\qquad\qquad\qquad\qquad\quad 1x^2 + 0x^1 + 0x^0 \quad \rightarrow \texttt{Rest = 100}
\end{array}
$$

Auf die Koeffizienten reduziert, sieht der Rechengang wie folgt aus:

```
(1 + 0 + 1 + 0 + 1 + 0 + 1) : (1 + 1 + 0 + 1) =
 1 + 1 + 0 + 1                         1 + 1 + 0 + 1
 -------------
 0 + 1 + 1 + 1 + 1
     1 + 1 + 0 + 1
     -------------
     0 + 0 + 1 + 0 + 0
         0 + 0 + 0 + 0
         -------------
             1 + 0 + 0 + 1
             1 + 1 + 0 + 1
             -------------
                 1 + 0 + 0   → Rest = 100
```

Diese Operationen können wir durch die Schaltung nach Abb. 5.55 realisieren. Die Positionen der Modulo-2-Addierer (XOR-Gatter) sind dabei durch das Polynom bestimmt, durch das wir teilen.

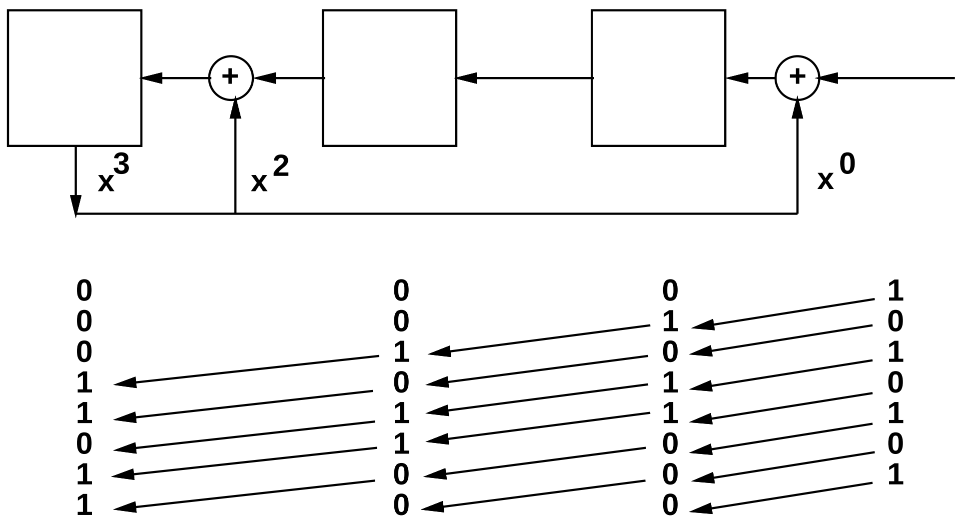


Abbildung 5.55: Division durch ein Polynom mit Registern und XOR-Gattern

5.3.2 Eigenschaften von zyklischen Codes

Vom Empfänger wird im allgemeinen nicht die Nachricht $F(x)$, sondern irgendeine, möglicherweise davon verschiedene Nachricht $H(x)$ empfangen werden. Sei $E(x)$ der durch Störungen verursachte Unterschied zwischen beiden, sei also

$$H(x) = F(x) + E(x).$$

$E(x)$ heißt **Fehlerpolynom** Jede 1 dieses Polynoms kennzeichnet eine Fehlerstelle. Für die Erkennung von Fehlern gilt das folgende:

- Ist $H(x)$ nicht durch $P(x)$ teilbar, so wird ein Fehler erkannt
- Ist $H(x)$ durch $P(x)$ teilbar, so ist die Übertragung fehlerfrei oder es liegt ein nicht erkennbarer Fehler vor.

Satz 1: Ein zyklischer Code, der durch ein Polynom mit mehr als einem Term erzeugt wird, entdeckt alle Einzelfehler.

Bew.: Einzelfehler besitzen ein Fehlerpolynom der Form $E(x) = x^i$. Hat $P(x)$ mehr als einen Term, so teilt es x^i nicht.

Satz 2: Jedes durch $1 + x$ teilbare Polynom hat eine gerade Zahl von Termen.

Bew.-Idee: Vielfache von $(1 + x)$ enthalten Paare.

$\Rightarrow$: Mit $(1 + x)$ als Faktor findet man eine ungerade Anzahl von Fehlern (Parity-Prüfung).

Def.: Ein Burstfehler der Länge b ist ein Fehler, bei dem die falschen Symbole den Abstand b haben.

Beispiel: $E(x) = x^7 + x^4 + x^3 = 0010011000$. Per Definition zählt man dies als Abstand 5, d.h. man zählt die Randsymbole mit.

Satz 3: Ein durch ein Polynom vom Grad $n - k$ erzeugter zyklischer Code entdeckt alle Burstfehler der Länge $b \leq n - k$, wenn $P(x)$ x nicht als Faktor enthält. **und $E(x)$ den Term x^0 nicht enthält** [7].

Bew.: Sei x^i der Term mit dem kleinsten Exponenten, der in $E(x)$ vorkommt. Wir können dann $E(x)$ darstellen als $E(x) = x^i * E_1(x)$. Das Produkt ist nicht durch $P(x)$ teilbar, wenn die beiden Terme nicht durch $P(x)$ teilbar sind.

1. x^i ist nicht durch $P(x)$ teilbar, wenn $P(x)$ nicht x als Faktor enthält.
2. $E_1(x)$ ist höchstens vom Grad $b - 1$, d.h. vom Grad $n - k - 1$. Daraus folgt: $P(x)$ teilt $E_1(x)$ nicht.

Wenn $E(x)$ den Term x^0 enthält (also $i = 0$ ist), dann kann kein x^i mit $i > 0$ ausgeklammert werden. Die o.a. Argumente bezüglich $E_1(x)$ gelten dann aber direkt bezüglich $E(x)$.

Satz 4: Die Anzahl der nicht erkannten Burstfehler der Länge $b > n - k$ ist, bezogen auf die Anzahl möglicher Burstfehler:

(5.6) $$2^{-(n-k-1)} \quad , \quad \text{wenn } b = n - k + 1$$

(5.7) $$2^{-(n-k)} \quad , \quad \text{wenn } b > n - k + 1$$

Beispiel: $P(x) = (1 + x^2 + x^{11})(1 + x^{11})$ erkennt:

- 1 Burst der Länge ≤ 22
- 2 Bursts der $\sum$-Länge ≤ 12 wenn $k \leq 22495$
- Jede ungerade Anzahl von Fehlern
- 99,99996 % der Bursts der Länge 23
- 99,99998 % der Bursts der Länge > 23

Burst-Fehler sind gerade deshalb so interessant, weil sie bei einer kurzzeitigen Störung der Übertragung oder bei einem Staubkorn auf dem Band oder der Platte auftreten.

[7] Fettgedrucktes fehlt im Originalartikel.

5.3.3 Standard-Generatorpolynome

Die Prüfbits $R(x)$ werden bei der Übertragung oder Speicherung von Nachrichten als sog. **CRC-Zeichen** (von engl. *cyclic redundancy check*) zugesetzt. Generatorpolynome sind laut DIN:

Magnetband 800bpi	$1 + x^3 + x^5 + x^6 + x^9$
Magnetbandkassette	$1 + x^2 + x^{15} + x^{16}$
Floppy disc	$1 + x^2 + x^{15} + x^{16}$
GCR ECC-Zeichen	$1 + x^2 + x^{15} + x^{16}$
HDLC	$1 + x^5 + x^{12} + x^{16}$

Dabei wird das CRC-Zeichen aus dem bitseriellen Datenstrom erzeugt. Bei byteparalleler Übertragung müssen die Daten also zusätzlich in einen bitseriellen Strom gewandelt werden.

Im Falle eines erkannten Lesefehlers wird in der Regel zunächst eine Wiederholung des Lesens durchgeführt. Erst nachdem eine gewisse Anzahl von Wiederholungen erfolglos blieb, wird auf einen permanten Fehler geschlossen.

5.4 Massenspeicher

5.4.1 Allgemeines

Als nächstes wollen wir Massenspeicher behandeln.

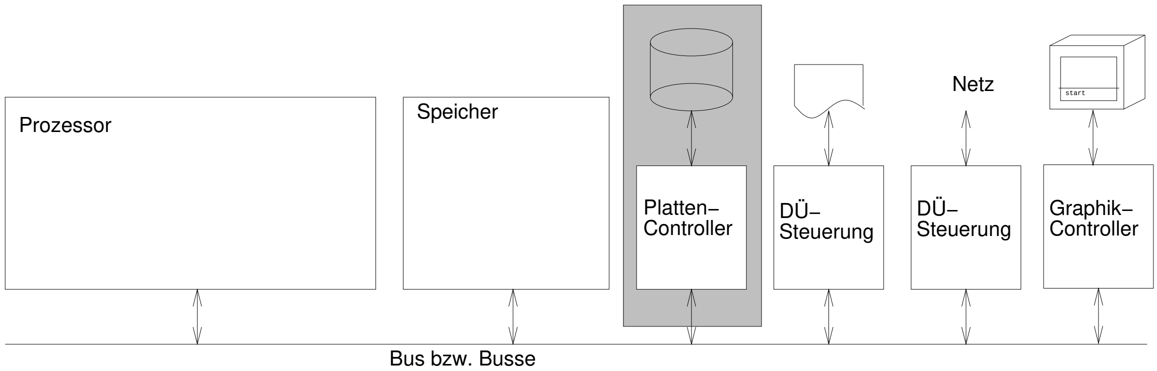


Abbildung 5.56: Massenspeicher eines Rechensystems

5.4.2 Controller-Schnittstellen

Neben den bei PCs üblichen EIDE-Schnittststellen kann der SCSI-Bus für den Anschluss von Massenspeichern genutzt werden.

Tabelle 5.5 enthält einige SCSI-Befehle der Befehlsgruppe 0.

5.4.3 Plattenspeicher

Abb. 5.57 zeigt den schematischen Aufbau eines Plattenlaufwerks.

Die wesentlichen Eigenschaften von Plattenlaufwerken (Aufteilung in Spuren und Sektoren usw.) wird hier als bekannt vorausgesetzt.

Befehls-Code	Befehl	Bedeutung
0	Test Unit Ready	Abfrage auf Betriebsbereitschaft
1	Recalibrate	Köpfe auf Spur 0 bewegen
3	Request Sense	Fehlerstatus einlesen
4	Format Unit	Festplatte formatieren
5	Read Block Limits	Abfragen der Festplattengröße
8	Read	Lesen eines Datenblocks
10	Write	Schreiben eines Datenblocks
11	Seek	Suchen einer Spur

Tabelle 5.5: SCSI-Befehle der Gruppe 0 (Festplatten-Systeme)

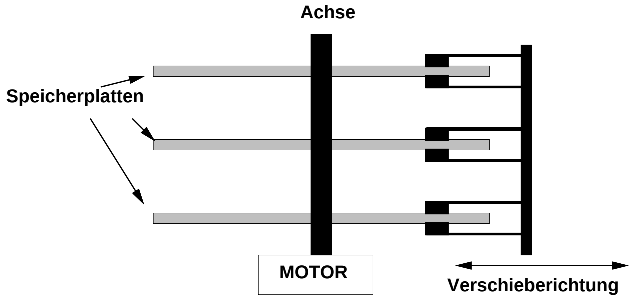


Abbildung 5.57: Plattenlaufwerk

5.4.3.1 Disc-Arrays

Eine spezielle Organisationsform von Plattenlaufwerken bilden die sog. **Plattenverbunde**. Aufgrund der stark gewachsenen Prozessorleistung ist man gerade in den letzten Jahren auch an einem Zuwachs der Geschwindigkeit der Schreib- und Lese-Operationen interessiert gewesen. Die Zugriffsgeschwindigkeit auf einzelne Platten konnte man leider nur noch sehr begrenzt erhöhen. Man ist daher dazu übergegangen, die Existenz mehrerer Platten zu einem parallelen Zugriff auszunutzen. Während vieler Jahre hat man die Existenz mehrerer Platten lediglich dazu ausgenutzt, von verschiedenen Platten verschiedene Blöcke zeitlich überlappt zu lesen. So konnten z.B. einige Platten mit der Suche nach der richtigen Spur und dem richtigen Sektor beschäftigt sein, während (evtl. von mehreren anderen Platten) Übertragungsvorgänge liefen. Diese Vorgehensweise kann aber meist nur für überlappte E/A-Operationen verschiedener Prozesse ausgenutzt werden. Primär wird so der Durchsatz (in E/A-Operationen pro Sekunde) erhöht, aber weniger die Wartezeit (engl. *latency*) eines einzelnen Prozesses reduziert. Um auch diese zu reduzieren, ist man dazu übergegangen, die einzelnen Bits der Datenblöcke auf verschiedenen Platten abzuspeichern. So angeordnete Platten nennt man *Drive Arrays* oder *Disc Arrays*. Eine Normung in diesem Bereich bietet die Spezifikation *RAID* (*redundant array of inexpensive discs*) . Folgende Verfahren sind normiert [HP96, WZ93, CLG$^+$94, Tra96]:

- RAID 0

  Die Informationen einer Datei werden auf mehreren Platten abgelegt (siehe Abb. 5.58). Bei diesem Verfahren werden mehrere kleine Laufwerke zu einem großen logischen Laufwerk zusammengefaßt. Der *Striping*- Parameter kennzeichnet dabei die Länge der jeweils auf eine Platte geschriebenen Stücke. Bei kleinem Striping-Faktor (z.B ein Byte) erreicht man eine hohe Übertragungsrate, verwendet aber evtl. auf jeder der Platten nur kleine Blöcke. Da keine Redundanz vorhanden ist, ist das Verfahren störanfällig und dürfte eigentlich nur AID-0 heißen.

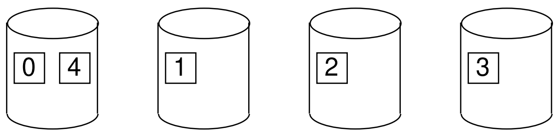


Abbildung 5.58: Verteilung von Daten auf verschiedenen Platten bei RAID 0

- RAID 1

  Bei diesem Verfahren wird mit **gespiegelten Platten** (engl. *mirrored discs*) gearbeitet, d.h. dieselben Daten werden auf zwei Platten geschrieben (siehe Abb. 5.58). Die Lesegeschwindigkeit kann so erhöht werden. Fällt eine aus, so kann mit den Daten der anderen Platte weitergearbeitet werden. Nachteil ist der doppelte Plattenaufwand.

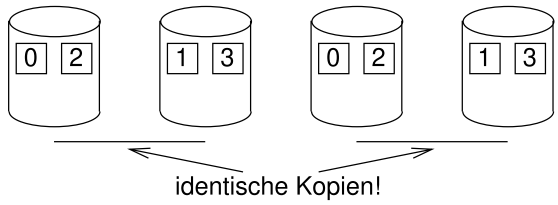


Abbildung 5.59: Verteilung von Daten auf verschiedenen Platten bei RAID 1

- RAID 2

  Die **Bits** eines Datenblocks werden wie auf mehreren Platten verteilt (engl. *bit interleaving*, *striped discs*). Zusätzlich werden Prüfbits auf spezielle Prüfbit-Platten geschrieben. Die Anzahl der Prüfbits erlaubt dabei die Verwendung von fehlerkorrigierenden Codes (ECC). So können trotz des Ausfalls von Platten die Daten regeneriert werden. RAID-2 besitzt geringere Redundanz als RAID 1, ist aber wegen der Prüfbit-Erzeugung und der Last auf den Prüfbit-Platten langsamer als RAID 1.

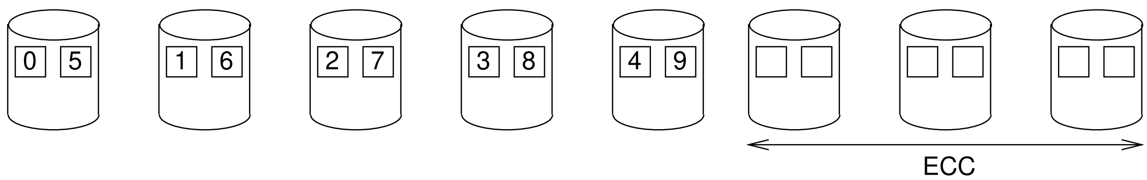


Abbildung 5.60: Verteilung von Daten auf verschiedenen Platten bei RAID 2

- RAID 3

  Wie RAID 2, jedoch wird nur ein einzelnes Prüfbit auf einer Parity-Platte abgelegt. RAID-3 besitzt eine schlechte Schreibperformance und besitzt nur beim Lesen von langen Blöcken eine gute Performance.

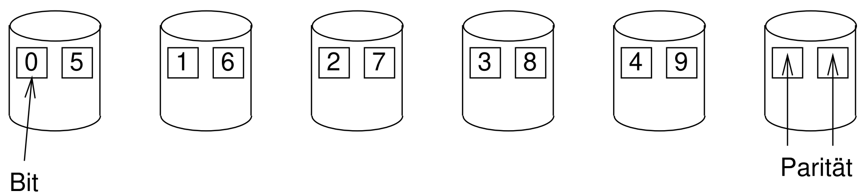


Abbildung 5.61: Verteilung von Daten auf verschiedenen Platten bei RAID 3

- RAID 4

  Wie RAID 3, jedoch mit einem Striping-Faktor von einem Block oder mehr. Dadurch bietet RAID-4 eine bessere Performance bei kurzen Zugriffen.

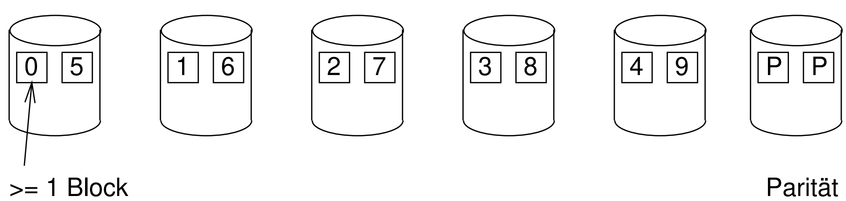


Abbildung 5.62: Verteilung von Daten auf verschiedenen Platten bei RAID 4

- RAID 5

  Bei RAID 5 wird die Paritätsinformation über verschiedene Platten verteilt. So wird die Paritätsplatte als Performance-Engpaß vermieden.

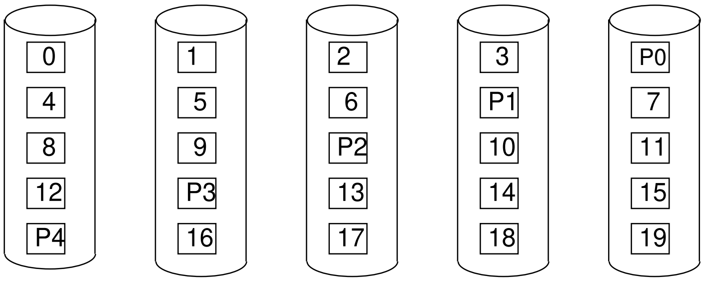


Abbildung 5.63: Verteilung von Daten auf verschiedenen Platten bei RAID 5

Eine Übersicht über die Eigenschaften der verschiedenen RAID-Ebenen bietet die Tabelle 5.6.

RAID level	Technik	Fehler toler.	Daten-Platten	Prüf-platten
0	nonredundant	0	8	0
1	mirrored	1	8	8
2	Memory-style ECC	1	8	4
3	bit-interleaved parity	1	8	1
4	block-interleaved parity	1	8	1
5	- " -, distrib. parity	1	8	1
6	P+Q redundancy [HP96]	2	8	2

Tabelle 5.6: Vergleich der RAID-Ebenen

RAID-Systeme dienen v.a. der verbesserten Verfügbarkeit von Daten; die Transfer-Rate wird im wesentlichen nur beim Transfer von großen Datenmengen deutlich verbessert [Tra96].

Über die Realisierung von Dateisystemen auf Plattenspeichern findet man in Büchern über Betriebssysteme (siehe z.B. Tanenbaum [Tan76]) ausreichend Information.

5.4.3.2 Weitere Plattenspeicher

- **CD-ROM/CD-R/CD-RW**

  Die CD-ROM, entsprechend "akustischen" CD's, erreicht eine Speicherkapazität von 650 MByte. Zugriffszeiten liegen in der Größenordnung von ca. 100 ms.

  Den grundsätzlichen Aufbau zeigt die Abb. 5.64[8].

  Die Intensität des reflektierten Lichts ist dabei abhängig von den Vertiefungen auf den Datenträgern.

- **DVD**

  Bei der DVD wird die Kapazität von CDROMs mit verschiedenen Mitteln erhöht. Zunächst einmal werden die Abstände der Vertiefungen und deren Größe gegenüber der CDROM reduziert (siehe Abb. 5.65.

  Durch diese Maßnahmen ergibt sch eine Kapazitätssteigerung etwa um den Faktor 4. Durch eine bessere Fehlerkorrektur kann außerdem die Redundanz reduziert werden. Es ergibt sich so eine Kapazität von 4,7 GB.

  Zusätzlich kann man durch Umfokussieren zwei Ebenen je Seite einer DVD nutzen (siehe Abb. 5.66). Dadurch kommt man auf eine Kapazität von 8,5 GB je Seite einer DVD.

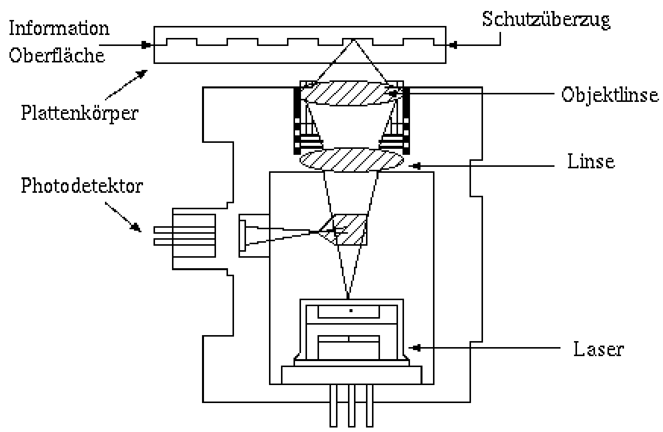


Abbildung 5.64: Grundsätzlicher Aufbau eines CDROM-Laufwerkes

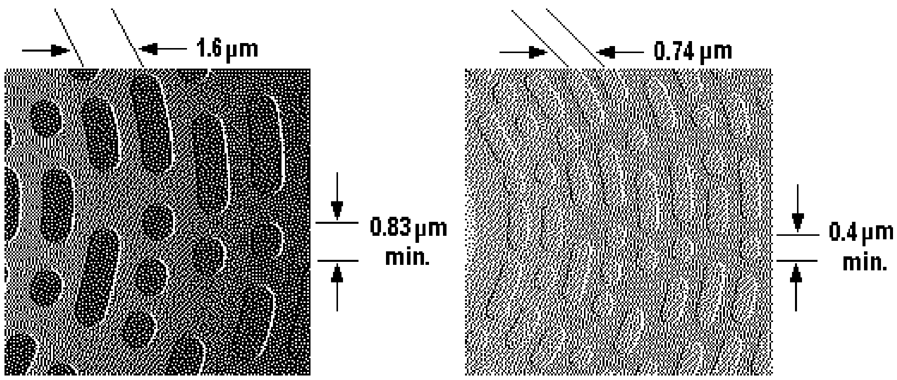


Abbildung 5.65: Vergleich der Vertiefungen bei der DVD und der CDROM

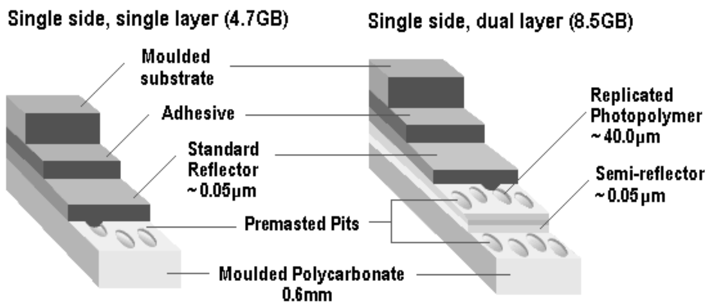


Abbildung 5.66: Ausnutzung zweier Ebenen je DVD-Seite

Schließlich kann man noch beide Seiten einer DVD nutzen und kommt so auf bis zu 17 GB je DVD (siehe Abb. 5.67).

Für die DVD wurde von Anfang an ein Dateiformat berücksichtigt: UDF (erweitertes ISO 9660).

Bei der DVD erfolgt die Speicherung in der Mitte der Schichtdicke; zur Kompatibilität mit den CDROMs werden in DVD-Laufwerke daher zwei separate Laser eingebaut. Die Datenrate bei DVDs "einfacher" Geschwindigkeit beträgt 1,25 MB/s.

Zur Speicherung von Filmen auf DVDs wird Kompression eingesetzt. Mit MPEG-2 reichen 4,7 GB für 133 min. Video mit verschiedenen Tonspuren.

- **Magneto-optische (MO-) Platten**

[8] Quelle: [aia.wu-wien.ac.at/inf_wirt/03.04.01.html]

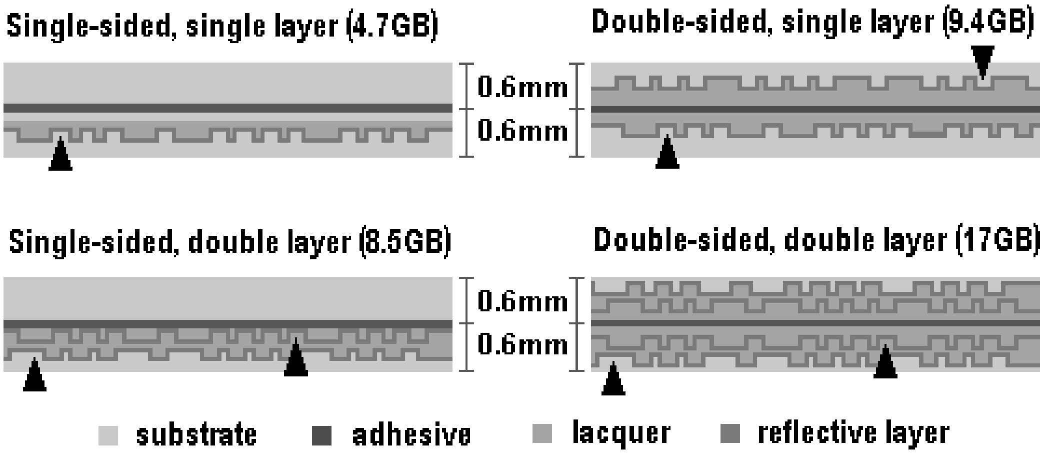


Abbildung 5.67: Speicherkapazitäten von DVDs

Diese Platten werden mit einem Laser beschrieben. Aufgrund des **Kerr-Effekts** kann mit dem Schreibvorgang eine Drehung der Polarisationsebene reflektierten Lichts ausgelöst werden. Die Platten werden vor für sicherheitskritische Anwendungen eingesetzt, bei denen die Sicherheit der Speicherung wichtiger ist als die gegenüber normalen Platten reduzierte Geschwindigkeit.

5.4.3.3 Platten-Caches

Analog zu den Caches als Pufferspeicher zur Beschleunigung der Hauptspeicher-Zugriffe realisieren die meisten Betriebssysteme heute **Plattencaches** zur Beschleunigung des Platten-Zugriffs .

Ein Problem bildet die Gewährleistung einer Sicherung gegen Systemabstürze. Es muß, insbesondere bei Datenbankanwendungen, sicher sein, mit welchem Zustand man nach einem Absturz wieder beginnen könnte. Aus diesem Grund werden Platten-Caches z.B. bei dem neuen SPARC 10-System in einem nichtflüchtigen Teil des Hauptspeichers realisert (NVRAM).

5.4.4 Bandlaufwerke

Für die Datensicherung sind weiterhin Bandlaufwerke mit sequenziellem Zugriff auf die Daten interessant, v.a. wegen ihres niedrigen Preises pro Bit und ihrer Speicherkapazität.

Verbreitete Formate sind die folgenden [Beh99]:

- **QIC-Kassetten**

  QIC steht für *quarter inch cartridge*. QIC-Kassetten (siehe Abb. 5.68) arbeiten, wie der Name schon sagt, mit 1/4"-Kassetten. Die Aufzeichnung erfolgt in mehreren Spuren nacheinander, wodurch mehrfaches Vor- und Zurückspulen erforderlich ist. Das Schreibverfahren ist darauf optimiert, dass vom Rechner mit der vollen Schreibgeschwindigkeit große Datenmengen bereitgestellt werden können. Fast das gesamte Band erhält tatsächlich Informationen; nur wenig Platz wird für Lücken zwischen den Blöcken verwandt. Mit Kompression sind bis zu 5 GByte speicherbar.

  Das Travan-5-Laufwerk stellt eine Weiterentwicklung dar (10 GByte, 1 MB/s, 9 DM/GB).

- **DAT-Bänder**

  DAT-Bänder (siehe Abb. 5.69) sind Bänder im Format der *digital audio tapes* (4 mm breit). Sie verwenden ebenfalls Schrägspuraufzeichnung und erreichen eine Speicherkapazität von 20 GByte und eine Transferrate von 2,4 MB/s (Stand 2000).

  Das DDS-4-Laufwerk stellt eine Weiterentwicklung dar.

- **DLT-Kasetten**

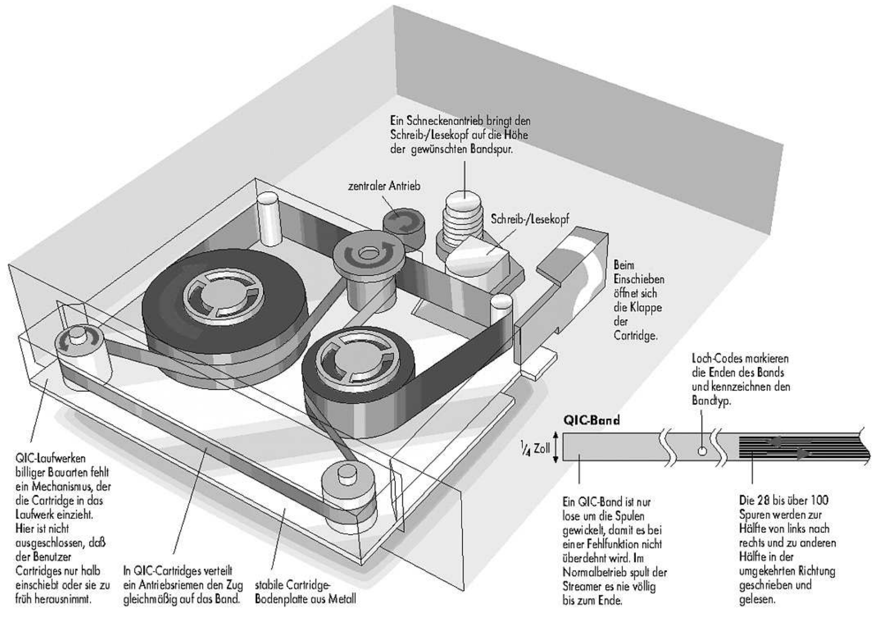


Abbildung 5.68: QIC-Laufwerk (©c't)

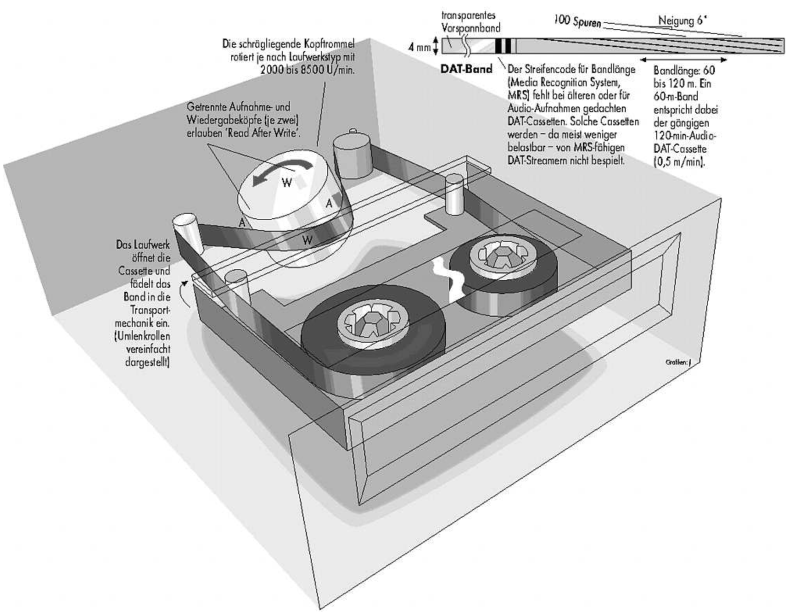


Abbildung 5.69: DAT-Laufwerk (©c't)

DLT-Kassetten enthalten 1/4"-Bänder. Die Aufzeichnung erfolgt wie bei QIC-Bändern parallel zur Kante des Bandes in mehreren Spuren nacheinander. Die Kassetten sind für den ständigen Gebrauch geeignet und erreichen eine Kapazität von 40 GByte (1999).

- **EXABYTE-Bänder**

  Bei EXABYTE-Bändern handelt es sich um Bänder im Format von 8mm Videobändern, die im Schrägaufzeichnungsverfahren beschrieben werden. Kapazitäten bis zu 60 GByte und Transferraten bis 12 MBit/s sind erreichbar (2000).

- **AIT-2**

  8 mm-Band mit Schrägspuraufzeichnung und integriertem Chip mit Formatinformation. Kapazität: 50 GB (2000)

- **WORM (write once read many)-Platten**

Dies sind nur einmalig beschreibbare, optische Platten, die für Archivierungszwecke eingesetzt werden. Kapazität: bis 3 GByte/Platte.

- **1/2"-Bänder**

  1/2"-Bänder werden im Start/Stop-Betrieb (mit relativ großen Lücken zwischen Blöcken) mit 9 Spuren bei bis zu 6250 Byte/Inch beschrieben. Diese Geräte waren an Großrechnern populär und wurden häufig als Inbegriff von Datenverarbeitung in den Medien gezeigt wurden. Sie sind mit einer Speicherkapazität von maximal 180 MByte und einem großen Volumen heute technisch veraltet.

5.5 Graphische Systeme

Als nächstes wollen wir uns mit den Graphik-Systemen[9], von Rechnern beschäftigen. Diese Systeme bestehen in der Regel aus dem Graphik-Controller und dem eigentlichen Bildschirm (siehe Abb. 5.70).

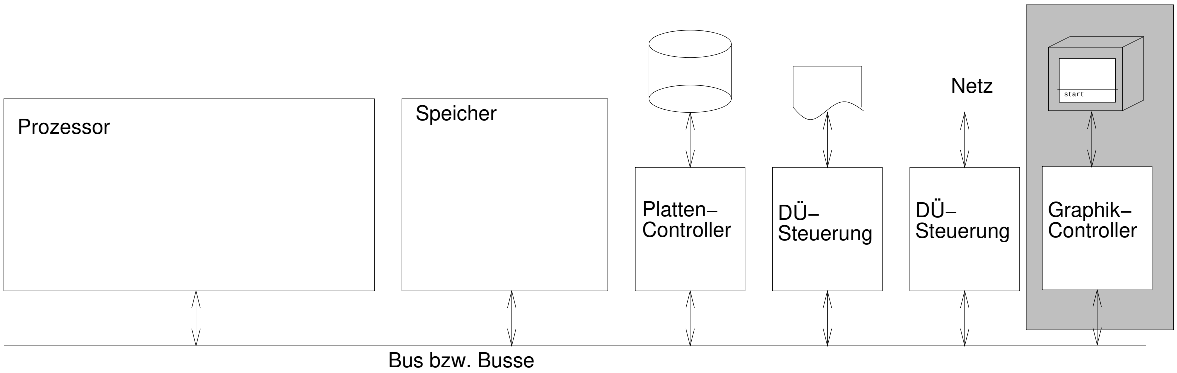


Abbildung 5.70: Graphik-Subsystem

5.5.1 Fernsehtechnik

Standard-Fernsehbildschirme bilden den historischen Ausgangspunkt für die Darstellung von Graphik. Daher sollen hier zunächst einige grundlegende Eigenschaften der Fernsehtechnik vorgestellt werden.

Die Darstellung von Fernsehbildern erfolgt mittels Kathodenstrahlröhren. Abb. 5.71 zeigt den Aufbau einer Kathodenstrahlröhre für die Erzeugung einfarbiger Bilder.

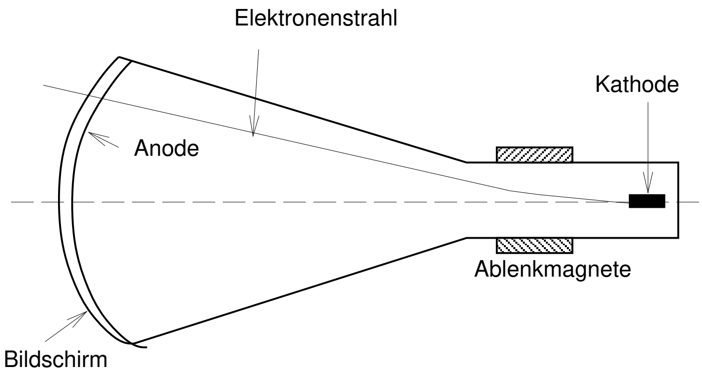


Abbildung 5.71: Kathodenstrahl-Röhre

Die Kathode dieser Röhre wird mittels eines elektrischen Stroms geheizt. Aus der Kathode treten dadurch Elektronen aus, die zur Anode hin angezogen werden. Auf diesem Weg zur Anode können die Elektronen

[9]Dieser Stoff wird in der nächsten Version des Skripts völlig überarbeitet sein und ist in der hier beschriebenen Darstellung nicht mehr prüfungsrelevant.

durch elektromagnetische Felder in ihrer Richtung beeinflußt werden. In der Fernsehtechnik werden heute ausschließlich Magnetspulen zur Felderzeugung eingesetzt.

Bei Farbbildröhren werden für den grünen, roten und blauen Anteil des Bildes eigene Kathoden benutzt. Mittels einer Lochmaske (siehe Abb. 5.72) wird dafür gesorgt, dass die aus diesen Kathoden austretenden Ströme auf dem Bildschirm jeweils nur auf Flächen entsprechender Farbe auftreffen.

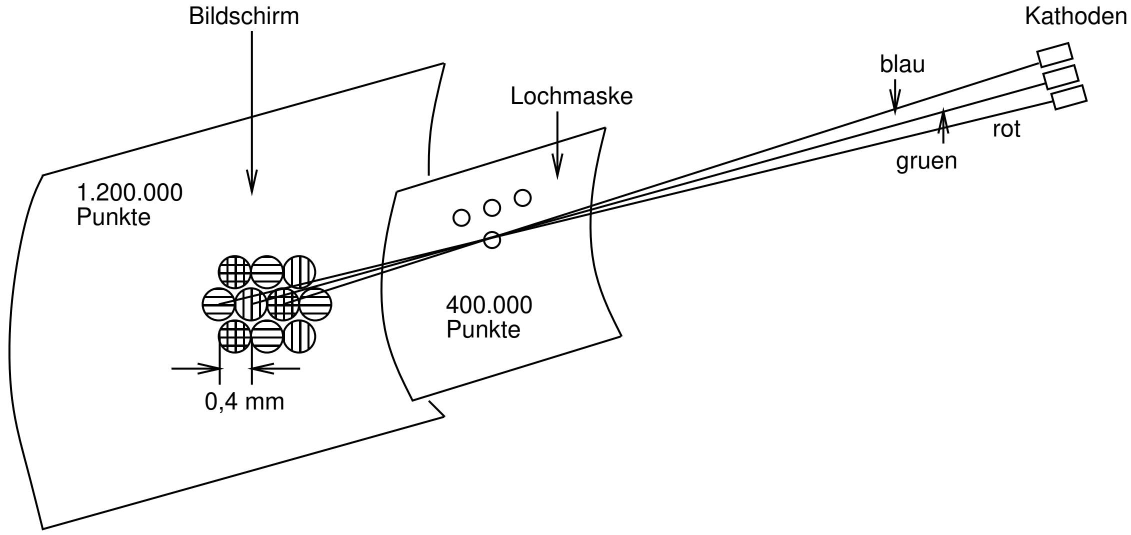


Abbildung 5.72: Prinzip der Erzeugung von Farbbildern

Da diese Flächen klein und eng benachbart sind, kann das Auge bei hinreichendem Betrachtungsabstand diese Flächen nicht mehr einzeln auflösen. Bei der üblichen Fernsehnorm besitzt die Lochmaske ca. 400.000 Löcher und der Bildschirm entsprechend 1,2 Mill. leuchtende kleine Flächen bzw. “Punkte”.

Die einzelnen Bildpunkte werden zeilenweise mit Elektronen “beschrieben”. Beim üblichen europäischen Fernsehbild stehen 52 μs für das Beschreiben einer Zeile und 12 μs für den Rücklauf, d.h. für das Ändern des Magnetfeldes, zur Verfügung. Während des Rücklaufs wird der Kathodenstrom (hoffentlich!) unterdrückt. Aus der Gesamtzeit von 64 μs pro Zeile läßt sich schließen, dass 15625 Zeilen pro Sekunde beschrieben werden. Dies ist die gelegentlich als Pfeifen hörbare **Zeilenfrequenz**.

Für jede Zeile gibt es laut Norm eine Helligkeitsauflösung von 200 Punkten. Während des Schreibens der Zeile wird also eine Übertragungsfrequenz von $200 * 1/52\ \mu\text{s} = 3{,}8$ MHz benötigt. Dies ist die für das Bild benötigte Übertragungs-**Bandbreite**.

Laut Norm besteht das Bild aus 625 Zeilen. Aus der Zeit für das Beschreiben einer Zeile kann wegen $1/(625 * 64 * 10^{-6}) = 25$ geschlossen werden, dass das Bild pro Sekunde 25× geschrieben wird. Diese Frequenz ist so niedrig, dass das Auge bereits die einzelnen Durchläufe als starkes Flimmern erkennen würde. Um mit der Übertragungskapazität auszukommen, wird das Auge überlistet, indem abwechselnd jeweils die geraden bzw. die ungeraden Zeilen beschrieben werden. Dadurch läuft der Elektronenstrahl 50× pro Sekunde vom oberen zum unteren Rand und es entsteht praktisch der Eindruck einer **Bildfrequenz** von 50 Hz. Wegen des alternierenden Beschreibens der geraden und ungeraden Zeilen heißt dieses Verfahren auch **Zeilensprungverfahren** (engl. *interlace mode*, vgl. Abb. 5.73).

Das Zeilensprungverfahren macht eine Strahlsteuerung erforderlich, die bei Rechner-Sichtgeräten wegen der vorhandenen Bildspeicher unnötig ist. Außerdem verbleibt beim Fernsehbild, insbesondere bei Betrachtung aus kurzer Entfernung, ein Restflimmern. Man geht daher zu höheren Bildfrequenzen über, z.B. zu 72 Hz.

Übliche Schnittstellen für Rechner-Sichtgeräte (**Monitore**), wie z.B. vertikale und horizontale Synchronisationssignale, BAS, FBAS und RGB-Signale und deren jeweilige Steckverbindungen beschreibt Bähring in seinem Buch.

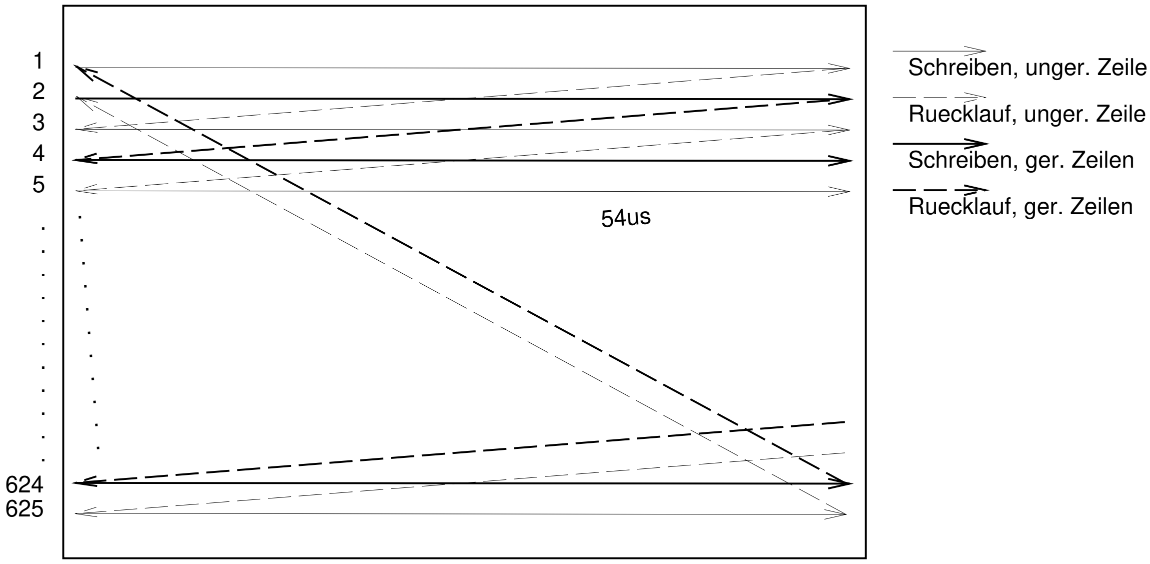


Abbildung 5.73: Zeilensprungverfahren

5.5.2 Alphanumerische Sichtgeräte

Ältere Sichtgeräte sind v.a. alphanumerische Sichtgeräte, d.h. nur zur Darstellung standardisierter Zeichensätze geeignet. Auf diese Weise gelingt eine erhebliche Reduktion des benötigten Speicherbedarfs. Ein Beispiel zeigt die Abb. 5.74.

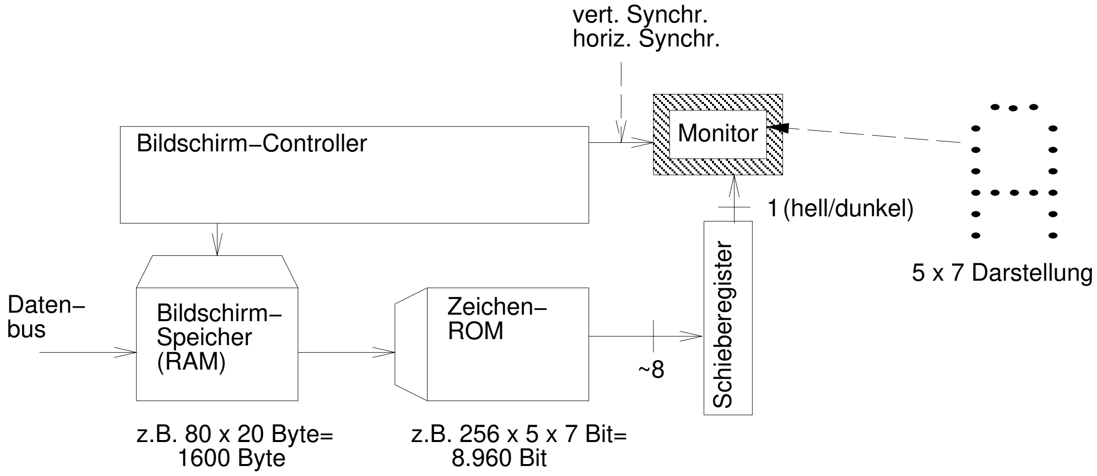


Abbildung 5.74: Bildspeicher alphanumerischer Sichtgeräte

Statt Speicher für alle vorhandenen Bildpunkte vorzusehen, werden alle darstellbaren Zeichen als Menge von Rasterpunkten im Zeichen-ROM abgelegt und dort über den Code des darzustellenden Zeichens adressiert. Dadurch läßt sich erheblich Speicher sparen, allerdings zu Lasten der Darstellbarkeit von anderen Zeichen oder Graphik. Der Ausgang des Zeichen-ROMs liefert jeweils alle Bits, die zur Darstellung einer Zeile eines Zeichens dienen. Diese Bits werden dann mit einer für das Zeichen-ROM nicht erreichbaren Geschwindigkeit seriell zum Monitor ausgegeben.

5.5.3 Graphische Sichtgeräte

Graphische Sichtgeräte enthalten für jeden Punkt des Bildes (**Pixel**) eigenen Speicher. Bei Farbbild- und Graustufen-Monitoren sind dies mehrere Bit pro Pixel. Hierzu werden spezielle Speicherbausteine, sog. **Video-RAMs** verwendet. Abb. 5.75 zeigt ein einzelnes Video-RAM.

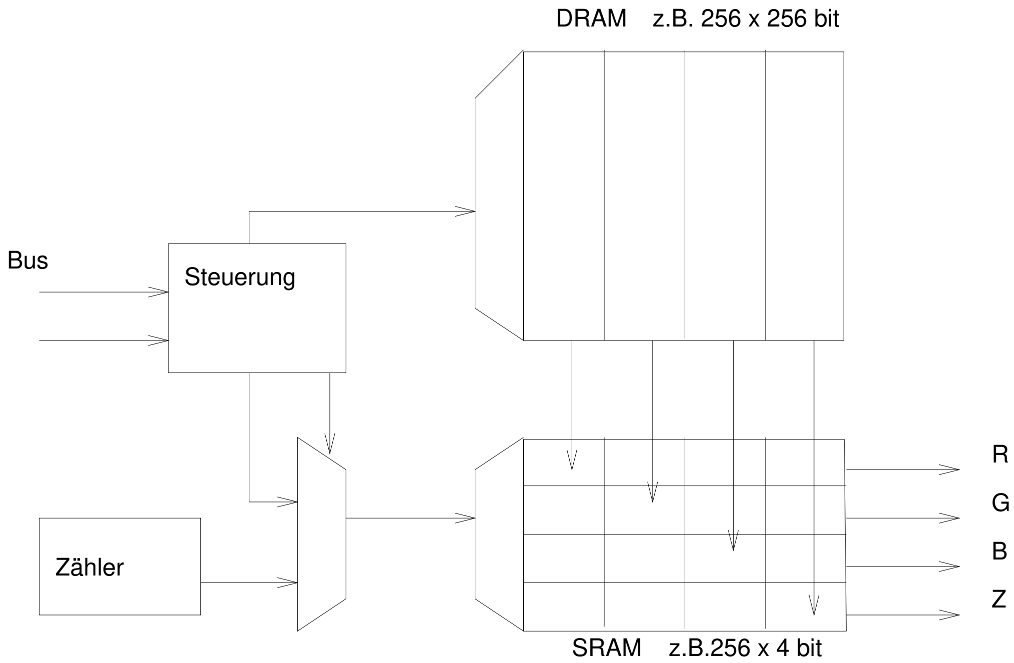


Abbildung 5.75: Video-RAM

Teile des Bildes werden im Video-RAM in DRAMs gespeichert. Diese sind jedoch nicht schnell genug, um mit der notwendigen Geschwindigkeit die Farbsignale erzeugen zu können. Daher werden Zeilen des DRAMs in ein SRAM kopiert, welches auch als Schieberegister verwendet werden kann. Der Schiebetakt kann wesentlich schneller als der Lesetakt des DRAMs sein. Das Video-RAM wird in zwei Phasen genutzt. In der ersten Phase wird das DRAM um Information über das nächste Bild ergänzt. Gleichzeitig kann unabhängig über das SRAM der Monitor versorgt werden. In der zweiten Phase, z.B. während des Strahl-Rücklaufs des Monitors, wird eine neue Zeile in das SRAM übernommen. Die hierfür benötigte Datenwegbreite (im Beispiel 256 Bit) kann nur innerhalb eines Chips realisiert werden. Deshalb lassen sich Video-RAMs auch nicht aus DRAM- und SRAM-Chips aufbauen, es sei denn, man sieht eine sehr aufwendige Verdrahtung vor. Praktische Graphik-Controller enthalten mehrere solcher Video-RAMs sowie einen Prozessor, der die CPU von der Erzeugung von Bildern im Video-RAM entlastet. Z.B. kann der Prozessor Rechtecke, Kreise, Polygone oder andere geometrische Objekte anhand einer Beschreibung der Parameter selbständig in Pixel-Informationen übersetzen.

Wichtig ist es, sich die Größe des benötigten Bildspeichers auszurechnen. Zur Darstellung von Bildern mit Fotoqualität benötigt man 16 Millionen Farben, also 24 Bit/Pixel. Bei einer Auflösung von 1280×1024 Punkten kommt man auf einen Speicherbedarf von ca. 30 MBit=3,75 MByte. Obwohl es spezielle Graphikstationen mit z.B. 40 Bit/Pixel gibt, versucht man, mit weniger Speicher auszukommen.

Eine Möglichkeit dazu sind die **Farbtafeln** (engl. *color look-up table*). Da man häufig nicht alle 16 Mill. Farben gleichzeitig benötigt, speichert man die benötigten Farben in einer Farbtafel und adressiert diese Farbtafel mit dem Code der gerade benötigten Farbe (siehe z.B. Abb. 5.76). Workstations erlauben so z.B., mit 8-12 Bit/Pixel 256-4096 Farben aus insgesamt 16 Mill. Farben darzustellen.

Verfügbare Graphik-Controller sind von unterschiedlicher Komplexität. Am oberen Ende anzusiedeln sind spezielle Hardware-Beschleuniger. Für PCs gibt es Beschleuniger für die Fensterverwaltung, die 3D-Darstellung und neuerdings auch für die Video-Dekodierung, z.B. nach dem MPEG-Verfahren. Ein Beispiel für die Hardware einer Hochleistungs-Workstation zeigt die Abbildung 5.77.

Das System besitzt eine 5-stufige feste Pipeline (siehe Abb. 5.77). Diese Abb. vermittelt einen Eindruck davon, dass Rechner nicht unbedingt immer nur eine ALU haben müssen.

Die Polygone werden von der CPU der Workstation über den Systembus an den Kommandoprozessor geschickt. Dieser verteilt die graphischen Primitiva gleichmäßig auf die **Geometrieprozessoren***, die die Koordinatentransformation und das Clipping durchführen. Nach einer Umorganisation der Polygonkanten werden in der* **Polygon-Engine** *die Farbwerte an den Ecken und an der Umrandung der Polygone berechnet.*

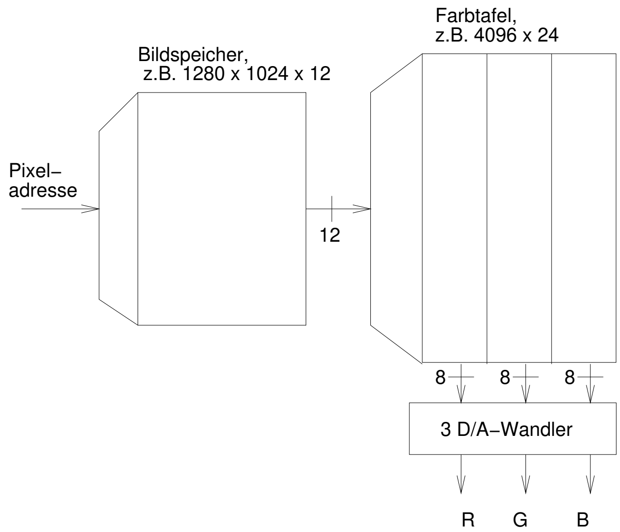


Abbildung 5.76: Farbtafel

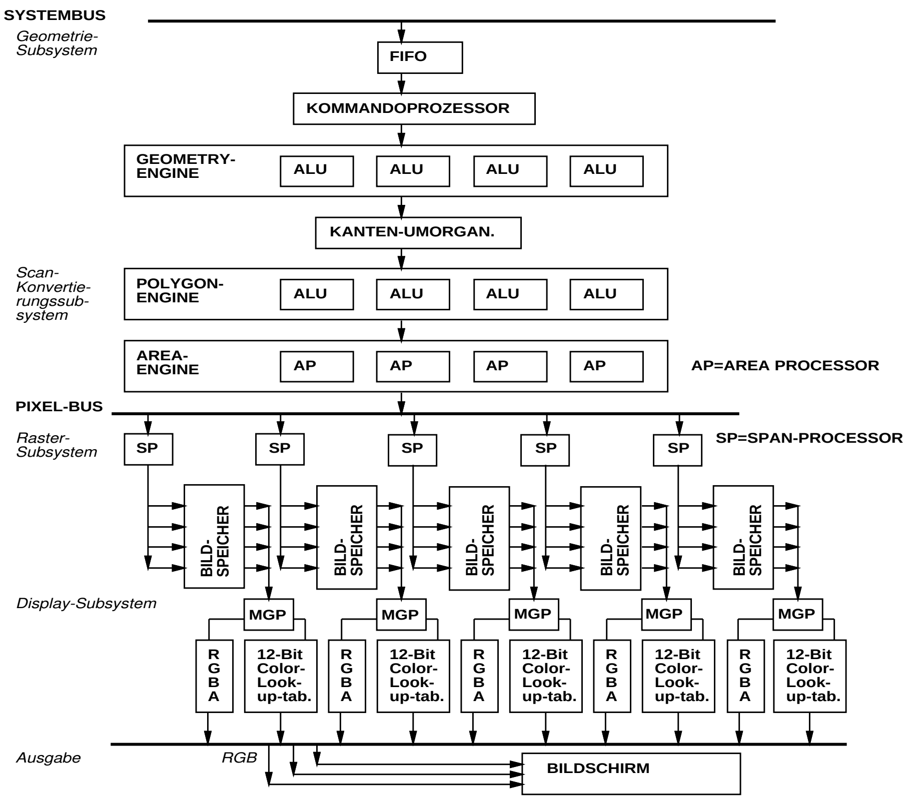


Abbildung 5.77: Graphikbeschleuniger der Serie *Silicon Graphics GTX*

Die **Area-Engine** *berechnet dann die Farbwerte der Pixel innerhalb der Polygone und führt gleichzeitig die Scan-Konvertierung durch. Die fertigen Pixel werden im Bildspeicher abgelegt. Von dort gelangen sie über den* **Multi Mode Graphics Processor (MGP)** *und einen D/A-Wandler auf den Bildschirm. Der MGP steuert die Farbzuordnungen.* (zitiert nach [MSS91], S. 335).

Bei anderen Workstations findet man häufig eine größere Flexibilität hinsichtlich der benutzbaren Algorithmen.

Kapitel 6

Multiprozessorsysteme

There is no reason anyone would want a computer in their home

[Ken Olson, Präsident und Gründer der Fa. Digital Equipment, 1977]

6.1 Einführung

In den vergangenen Jahren konnte die Leistungsfähigkeit einzelner Prozessoren enorm gesteigert werden. Es ist jetzt allerdings zu sehen, dass superskalare Prozessoren bereits extrem komplex sind. Eine gewisse weitere Leistungssteigerung, von Fortschritten in der Technologie einmal abgesehen, mag noch mit EPICs möglich sein. Es wird aber immer schwieriger, für Einzelprozessoren die Leistungssteigerung des Moore'schen Gesetzes beizubehalten.

Seit vielen Jahren versucht man, Leistungssteigerung durch Parallelarbeit in vielen Prozessoren zu erreichen. Problematisch an der Nutzung der Parallelität ist jedoch in der Regel der zusätzliche Aufwand bei der Programmierung von Parallelrechnern[1]. Die Programmierschnittstellen von Parallelrechnern werden aber langsam komfortabler und die Nutzung von Parallelrechnern dürfte in den nächsten Jahren zunehmen.

Wir wollen uns in diesem Kapitel[2] mit den Grundlagen von Parallelrechnern beschäftigen.

6.1.1 Klassifikation von Rechnern nach Flynn

Von Flynn stammt eine Klassifikation von Parallelrechnern, die zwar nicht sehr aussagekräftigt ist, die aber mangels wesentlich besserer Kriterien immer noch benutzt wird. Danach werden Rechensysteme nach der Anzahl ihrer Befehls- und Datenströme eingeteilt:

- **SISD:** *single instruction stream, single data stream*

  Dies sind die bislang besprochenen Mono-Prozessorsysteme.

- **SIMD:** *single instruction stream, multiple data streams*

  Zu jedem Zeitpunkt wird derselbe Befehl auf unterschiedlichen Daten ausgeführt. Ein Beispiel dafür haben wir bereits mit den Multimedia-Befehlen kennengelernt.

[1] Programmiersprachen, die eine automatische Parallelisierung erlauben (wie z.B. funktionale Sprachen oder Datenflusssprachen) haben sich bislang nicht durchgesetzt.

[2] Grundlage für dieses Kapitel ist überwiegend das Kapitel 8 in [HP96]. Ergänzendes Material ist im Kapitel 8 in [HP95] sowie den Büchern von Culler et al. [CS99] und Lindemann [Lin98] zu finden.

Weiter gehören in diese Klasse die sog. **Feldrechner**. Ein ganzes Feld von Prozessoren führt dabei denselben Befehl aus.

Beispiele:

- ILLIAC IV (Illinois Array Processor): Vorläufer aller Feldrechner, von D. Kuck und Mitarbeitern an der Universität von Urbana-Champaign entwickelt. Der ILLIAC IV war technisch und v.a. finanziell offenbar ein Fehlschlag. Die Arbeiten in Illinois sind aber grundlegend für alle Arbeiten zur Parallelisierung gewesen.
- ICL Distributed Array Processor (DAP), 64×64 1-Bit Prozessoren, 370 Mill. 32-Bit Gleitkomma-Additionen/Sek.

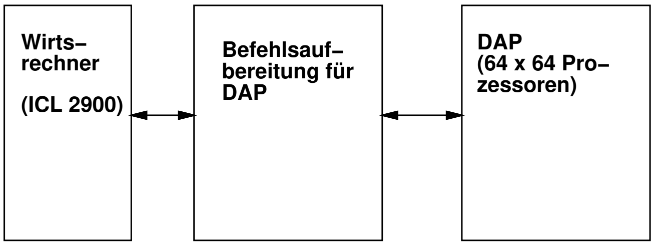


Abbildung 6.1: DAP-System

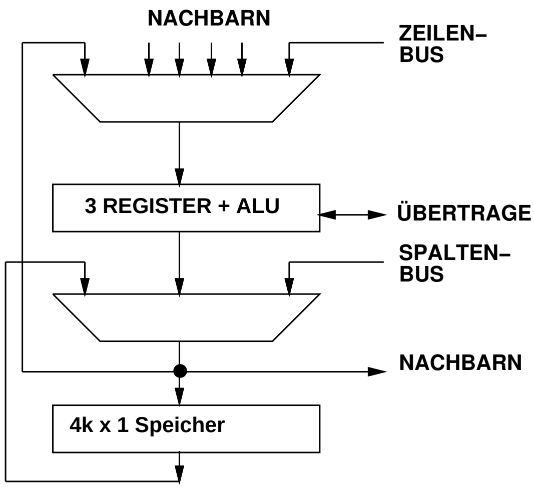


Abbildung 6.2: Struktur eines einzelnen Prozessors des Prozessorfeldes

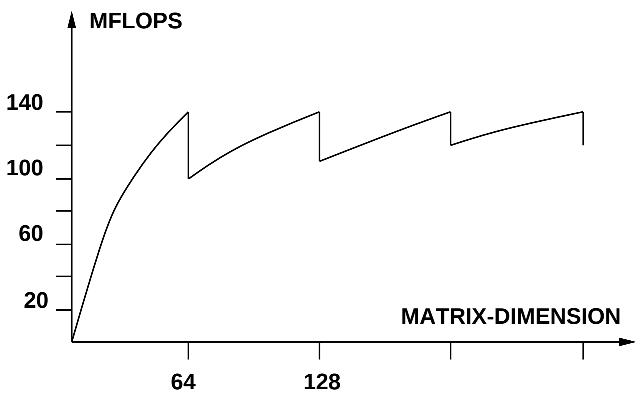


Abbildung 6.3: Leistung des DAP in Abhängigkeit von der Vektorgröße

Der DAP wurde z.B. an der Universität Erlangen als Hardware-Plattform für die Entwicklung paralleler Algorithmen benutzt.

– NASA/Goodyear Massively Parallel Processor (MPP), Auslieferung 1982 128 × 128 1-Bit Prozessoren. Bis zu 6.533 MIPS (8-Bit Integer-Additionen).

Literatur: [Sch]

Bei den genannten Beispielen handelt es sich durchgehend um ältere, da diese Systeme wenig flexibel sind und das Interesse an diesen Systemen deutlich abgenommen hat.

Eine Variante der SIMD-Maschinen sind *systolische Arrays*. Bei systolischen Arrays führen alle Prozessoren nur einen einzigen festen Befehl aus und die Daten werden durch das Array hindurchgeschoben. Ein Beispiel dafür ist das Systolische Array *odd even transposition sort* (OETS)[3]. OETS dient dem parallelen Sortieren von Zahlen. Für das Sortieren von n Zahlen werden $O(n^2)$ Vergleicher benötigt, die entsprechend Abb. 6.4 angeordnet werden. Nach $O(n)$ Schritten stehen eingegebene Zahlentupel am Ausgang sortiert zur Verfügung. Die ersten Schritte zeigt Abb. 6.5. In jedem Schritt kann ein neues Zahlentupel eingegeben werden.

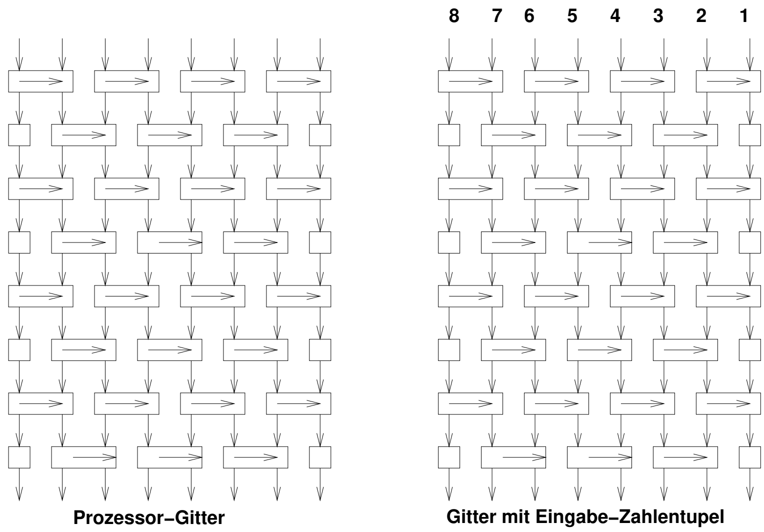


Abbildung 6.4: Struktur des systolischen Arrays OETS für $n = 8$

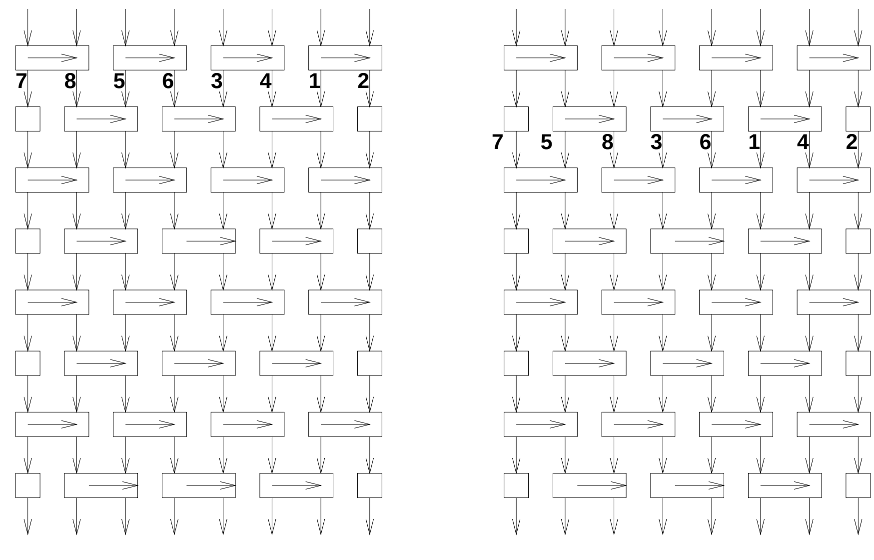


Abbildung 6.5: Die ersten beiden Sortierschritte von OETS

Systolische Arrays sind v.a. für spezielle Anwendungen, z.B. bei der schnellen Videosignalverarbeitung interessant. Sie können auch mit Hilfe von konfigurierbaren Logikschaltungen (FPGAs) in Form von sog. *custom computing machines* realisiert werden.

[3] Von Kennern der Szene liebevoll "Ötz" genannt.

- **MISD:** *multiple instruction streams, single data stream*

  Diese Klasse ist nur mit Mühe sinnvoll zu füllen[4]. Pipeline-Rechner können in diese Klasse aufgenommen werden, weil ein bestimmtes Datum in den verschiedenen Stufen der Pipeline von unterschiedlichen Befehlen bearbeitet wird.

  Beispiele für Pipelines:

  - Reine arithmetische Pipelines: Gleitkomma-Rechenwerk mit überlapptem Exponentenvergleich, Exponentenanpassen, Mantissenoperation, Normalisieren.
  - *Vektorrechner:*

    Vektorrechner besitzen spezielle Maschinenbefehle für die Verarbeitung von Vektoren. Sie enthalten eine oder mehrere Pipelines, die auf die schnelle Verarbeitung von Vektorelementen ausgelegt sind.

    Beispiele: CDC-205 (80er Jahre), IBM-Rechner mit VF-Zusatz, CRAY-Rechner.

    Leistung: CDC-205: 100 Mill. 32-Bit Gleitkomma-Additionen pro Sek. und Vektor-Pipeline.

  Diese Beispiele sind aber zugegebenermaßen etwas zwanghaft konstruiert, um überhaupt Beispiele für diese Klasse zu haben.

- **MIMD:**, *multiple instruction streams, multiple data streams*

  In diese Klasse fallen gekoppelte SISD-Maschinen. Davon gibt es wieder verschiedene Varianten:

  - Eng gekoppelte SISD-Maschinen, Kopplung über Hauptspeicher oder Platten: *Mehrprozessor-Anlagen.*
  - Lose gekoppelte SISD-Maschinen, Kopplung über E/A-Schnittstellen: *Rechnernetze.* Diese kann man weiter unterteilen in *lokale Rechnernetze* (Übertragungsgeschwindigkeit $>$ 100 kBaud) und *nicht-lokale Rechnernetze* (Übertragungsgeschwindigkeit $\leq$100 kBaud).
  - Der *Transputer* als Sonderfall. Die Transputer CPU-Chips besitzen bereits mehrere schnelle E/A-Schnittstellen, die sog. *Känale.*

    Ausgangsbasis für die Transputer-Hardware ist die Sprache CSP (*concurrent sequential processes*) von Hoare. Speziell für diese Sprache wurde die Transputer-Hardware von der Fa. `INMOS` entworfen. Die Hardware unterstützt die Kommunikation über Nachrichtenkanäle.

    Beispiel (Prinzip):

    ```
    PROCESS A                           PROCESS B
    .....                               ......
    VAR a ..                            VAR b ..
     a := 3;                            ...
     c!a   ;  --schreibe a auf Kanal c  c?b   -- lese b über Kanal c
    END;                                END;
    ```

    Beide Prozesse warten ggf. darauf, dass der andere Prozeß ebenfalls am Treffpunkt ankommt (*rendez-vous*-Konzept).

    CSP ist die Grundlage für die von der Herstellerfirma `INMOS` entwickelte Sprache `OCCAM`.

    Die ersten Transputer besaßen 4 Hardware-Kanäle. Die Kommunikation war mit den jeweils daran angeschlossenen Transputern möglich. Innerhalb eines Prozessors konnten (vom Betriebssystem simulierte) Kanäle zur Interprozeßkommunikation benutzt werden.

    Neuere Transputer benutzten die Kommunikation über **virtuelle Kanäle**. Nachrichten über diese Kanäle wurden für den Benutzer unsichtbar über Zwischenstationen geschickt. Damit war die Kommunikation zwischen beliebigen Transputern leicht programmierbar. Transputer konnten langfristig leistungsmäßig nicht mit den Standardprozessoren amerikanischer Hersteller konkurrieren. Damit ist das CSP-Konzept natürlich nicht ungültig. Es wird für die CSP-Realisierung von Realzeit-Betriebssystemen eingebetteter Systeme genutzt (siehe www.eonic.com).

Wir werden uns in diesem Kapitel v.a. mit MIMD-Systemen beschäftigen. Der große Vorteil dieser Systeme ist ihre Flexibilität.

[4] In vielen Büchern wird sie auch als leer bezeichnet.

6.1.2 UMA- und NUMA-Architekturen

MIMD-Systeme können anhand ihrer Speicherarchitektur klassifiziert werden. Die erste Klasse von Systemen enthält einen zentralen Hauptspeicher (siehe Abb. 6.6).

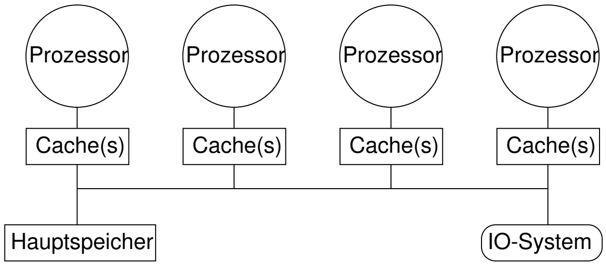


Abbildung 6.6: MIMD-System mit zentralem Hauptspeicher

Bei diesen Systemen benötigen alle Zugriffe auf den Hauptspeicher dieselbe Zugriffszeit. Sie heißen daher *uniform memory access* (UMA) - Architekturen. Diese Architekturen können genutzt werden, solange die Anzahl der Prozessoren nicht zu groß sind und solange eine große Trefferzahl in den Caches erreicht wird. Ansonsten wird der Zugriff auf den gemeinsamen Speicher schnell zu einem Engpass. Viele der heute gebräuchlichen Multiprozessor-Systeme fallen in die UMA-Kategorie.

Bei einer zweiten Gruppe von Rechnern ist der Hauptspeicher verteilt aufgebaut (siehe Abb. 6.7).

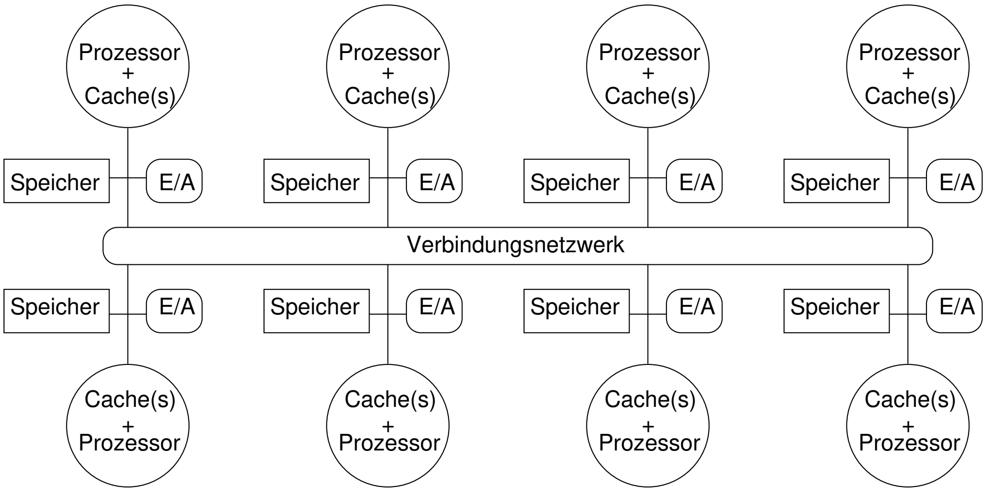


Abbildung 6.7: MIMD-System mit verteiltem Hauptspeicher

Ein wesentlicher Vorteil dieser Architektur ist die Skalierbarkeit: es können relativ leicht größere Prozessorzahlen erreicht werden, ohne dass der Speicher zu einem Engpass wird. Die Zugriffszeit auf den Speicher hängt allerdings davon ab, ob auf den lokalen Speicher eines Prozessors zugegriffen wird oder auf den Speicher eines anderen Prozessors. Entsprechend heißen diese Architekturen *non-uniform memory access* (NUMA)-Architekturen. .

An jeden einzelnen Speicher kann tatsächlich statt eines Prozessors auch eine kleine Zahl von Prozessoren angeschlossen sein.

6.1.3 Modelle der Kommunikation

Man kann Speicher eines Multiprozessorsystems entweder über einen gemeinsamen Adressraum oder über separate Adressräume der einzelnen Speicher adressieren. Im Falle eines gemeinsamen Adressraums sprechen wir von einer Architektur mit gemeinsamen Speicher (engl. *shared memory*). Eine NUMA-Architektur mit gemeinsamen Speicher bezeichnen wir als *distributed shared memory* (DSM)-Architektur. Im Falle separater

Adressräume besteht das System möglicherweise aus separaten Rechensystemen, die nur über ein Netzwerk verbunden sind.

Wir können weiterhin zwischen zwei Arten der Kommunikation unterscheiden: der Kommunikation über gemeinsamen Speicher und der Kommunikation per Nachrichtenaustausch (engl. *message passing*). Im ersten Fall muss den Kommunikationspartnern lediglich die Adresse der Nachricht bekannt sein; unnötiges Kopieren und Übertragen entfällt. Bei Nachrichtenaustausch wird eine Kopie aus dem Speicher des Absenders übertragen, im Speicher des Emfängers hinterlegt und der Empfänger dann passend benachrichtigt.

Vorteile des gemeinsamen Speichers sind die folgenden:

- Kompatibilität mit dem gut verstandenen Programmiermodell für Systeme mit zentralem Speicher,
- einfache Realisierung unterschiedlicher Arten der Kommunikation,
- geringer Overhead bei der Kommunikation, die sogar ohne die Hilfe des Betriebssystems zu realisieren ist,
- die Möglichkeit, automatisches Caching der Daten zu nutzen.

Vorteile des Nachrichtenaustauschs sind:

- die Hardware kann einfacher sein und sie wird skalierbar,
- Kommunikation muss explizit beschrieben werden, was für viele Anwendungen ein heilsamer Zwang ist.

Im Prinzip ist es möglich, Kommunikation nach einem Modell zu realisieren auch wenn die Hardware nach dem anderen aufgebaut ist. So kann man leicht auf einer DSM-Architektur eine Kommunikation per Nachrichtenaustausch realisieren. Kommunikation nach dem Modell des gemeinsamen Speichers auf Hardware ohne gemeinsamen Speicher zu realisieren ist schwieriger. Man könnte aber beispielsweise die Seitentabellen beim *paging* so modifizieren, dass der Zugriff auf gewisse Seiten das Betriebsystem veranlasst, diese Seite von einem anderen Rechner zu holen. Die Konsistenz ist allerdings dann schwierig zu realisieren.

6.2 Speicherkohärenz

6.2.1 Architekturen mit zentralem gemeinsamen Speicher

Wir können bei den UMA-Architekturen unterscheiden zwischen Daten, die ein Prozessor exklusiv benutzt (**private Daten**) und Daten, auf die mehrere Prozessoren zugreifen (**gemeinsame Daten**). Beide Sorten von Daten können in Caches zwischengespeichert werden. Werden Daten in Caches dupliziert, so kann sich Inkohärenz zwischen den Kopien ergeben. Genauer unterscheiden wir zwischen der Daten-**Kohärenz** und der Daten-**Konsistenz**. **Kohärenz** sagt etwas darüber aus, welcher Wert beim Lesen abgeliefert wird. Kohärenz bezieht sich stets auf das Lesen und Schreiben derselben Speicherzelle.

Def.: Ein Speichersystem heisst **kohärent**, wenn

1. bei einem Schreiben einer Zelle x durch einen Prozessor, welches von einem Lesen derselben Zelle gefolgt wird, das Lesen immer den geschriebenen Wert abliefert, sofern zwischen beiden Operationen kein Schreiben eines anderen Prozessor erfolgt.
2. bei einem Schreiben einer Zelle x durch einen Prozessor P, welches von einem Lesen derselben Zelle durch einen Prozessor P' gefolgt wird, das Lesen immer den geschriebenen Wert abliefert, sofern zwischen beiden Operationen kein Schreiben eines anderen Prozessor erfolgt und sofern zwischen beiden Operation hinreichend viel Zeit vergeht.
3. Schreibvorgänge in dieselbe Zelle serialisiert werden, d.h., zwei Schreibvorgänge durch zwei Prozessoren werden durch die übrigen Prozessoren in derselben Reihenfolge gesehen.

Die erste Eigenschaft bedeutet, dass wir die Operationsreihenfolge des sequenziellen Programms (engl. *program order*) erhalten wollen, eine Eigenschaft, die wir auch für Einzelprozessoren fordern. Die zweite Eigenschaft trifft den Kern der Kohärenz: könnten wir beliebig lange alte Werte lesen, so wäre das Speichersystem ganz klar inkohärent. Die dritte Eigenschaft wird benötigt, damit alle Prozessoren, die jeweils den zuletzt gesehenen Wert lokal speichern, auch denselben Wert speichern.

Konsistenz sagt etwas darüber aus, wie lange es dauert, bis ein geschriebener Wert x von anderen Prozessoren gelesen wird. Dabei geht es nicht um absolute Zeiten, sondern um zeitliche Ordnungen zwischen den Schreib- und Lesevorgängen auf andere Zellen und dem Lesen des neuen Wertes x.

Das Kohärenz-Problem tritt sowohl bei Ein- und Ausgaben wie auch bei Mehrprozessorsystemen auf. Die Lösungen sind allerdings unterschiedlich. Bei Ein- und Ausgaben ist es eine Ausnahme, wenn mehrfache Kopien von Daten existieren (eine Ausnahme, die möglichst ganz zu vermeiden ist). Bei Mehrprozessorsystemen sind Kopien von Daten entscheidend für die Leistung des Systems. Es ist wichtig, sowohl die **Migration** von Daten wie auch deren **Duplizierung** zu unterstützen.

Verfahren, welche die Kohärenz von Caches garantieren, heissen **Cache-Kohärenz**-Protokolle. Der Schlüssel zur Kohärenz ist die Verfolgung der Zustände (*ungültig, gültig, shared, private* usw.) gemeinsamer Datenblöcke. Hierfür gibt es zwei Klassen von Verfahren:

- **Verzeichnisbasiert** (engl. *directory based*): Der Zustand wird in einem Verzeichnis protokolliert.
- **Bus-Lauschen** (engl. *snooping*): Jeder Cache besitzt Informationen über den Zustand der in ihm enthaltenen Blöcke. Die Caches sind an einem zentralen Bus angeschlossen, beobachten die Vorgänge auf dem Bus und reagieren entsprechend.

Im Falle von mehreren Cache-Ebenen (L1-Cache, L2-Cache) wird die Gewährleistung der Konsistenz vereinfacht, wenn die sog. *inclusion property* vorliegt, die wie folgt definiert ist:

Def.: In einem Mehrebenen-Cache-System gilt die *inclusion property* genau dann, wenn ein Datenblock zu einer Adresse x nur dann in einem *level i*-Cache vorliegen darf, wenn auch der zugehörige *level i+1*-Cache dieselbe Adresse gepuffert hat.

Dies bedeutet nicht, dass beide Cache-Ebene zu der Adresse x notwendigerweise denselben Wert gespeichert haben (was bei einem *write back*-L1-Cache auch nicht der Fall ist). Wesentlich ist, dass die jeweils langsameren Caches 'wissen´, zu welchen Speicherzellen Daten in schnelleren Caches vorliegen könnten und sich so um Konsistenz-Maßnahmen kümmern können. Weiterhin wäre sonst auch die Aktualisierung des Hauptspeichers von einem L1-Cache aus sehr schwierig.

Es gibt zwei Methoden, um die geforderte Kohärenz sicherzustellen:

- Das *write update*-Protokoll:

  Jedes Beschreiben einer gemeinsamen Zelle führt zu einer automatischen Aktualisierung aller Kopien. Tabelle 6.1 zeigt ein Beispiel (Hennessy, Abb. 8.8):

Prozessor-Aktivität	Bus-Aktivität	Inhalt Cache A	Inhalt Cache B	Inhalt Speicher
				0
CPU A liest x	Miss für x	0		0
CPU B liest x	Miss für x	0	0	0
A: x:= 1	broadcast	1	1	1
CPU B liest x		1	1	1

Tabelle 6.1: Beispiel für ein *write update* Protokoll

  Dieses Protokoll führt zu einer starken Belastung des Busses. Aus diesem Grund wird in der Regel das nachfolgend beschriebene *write invalidate*-Protokoll realisiert, das zu einer geringeren Bus-Belastung führt. Wir werden das *write update*-Protokoll im Folgenden nicht weiter betrachten.

- Das *write invalidate*-Protokoll:

  Jeder Prozessor, der eine Zelle beschreiben möchte, lässt zunächst alle Kopien des Zelleninhaltes für ungültig erklären. Tabelle 6.2 zeigt ein Beispiel (Hennessy, Abb. 8.7; Annahme: x ist initial 0).

Prozessor Aktivität	Bus Aktivität	Inhalt Cache A	Inhalt Cache B	Inhalt Speicher
				0
CPU A liest x	Miss für x	0		0
CPU B liest x	Miss für x	0	0	0
A: x:= 1	x ungültig	1		0
CPU B liest x	Miss für x	1	1	1*

Tabelle 6.2: Beispiel für ein *write update* Protokoll (*: vereinfacht das Protokoll)

Ein Schreibvorgang beim *write invalidate*-Protokoll erfordert, dass sich der zu beschreibende Cache zunächst um den Bus bewirbt (siehe Bus-Arbitrierung, Kapitel 5). Nach der Zuteilung des Busses löst er ein *write invalidate*-Signal aus. Erst danach kann das Schreiben in den Cache abgeschlossen werden. Dieses Verfahren stellt die Kohärenz sicher: Eigenschaft 2 der Definition ist erfüllt, weil alle Kopien von Daten vor dem Schreiben vernichtet werden und ein Lesevorgang nach dem Schreiben zu einer Neuanforderung des Inhalts der Zelle führt. Das Protokoll gewährleistet auch die Eigenschaft 3: von zwei schreibwilligen Prozessoren wird einer den Wettlauf um den exklusiven Zugriff gewinnen und als erster schreiben. Der zweite Prozessor wird danach den exklusiven Zugriff bekommen und ebenfalls schreiben. Es ist ausgeschlossen, dass lesende Prozessoren noch den vom ersten Prozessor geschriebenen Wert erhalten während der zweite Prozessor bereits geschrieben hat.

Es ist sinnvoll, anhand des Cache-Zustandes zu vermerken, ob Kopien eines Datums in anderen Caches existieren, denn so kann eine unnötige Belastung des Busses bei privaten Daten vermieden werden.

Gilt die *inclusion property*, dann können die Blöcke in schnelleren Caches ungültig erklärt werden, indem sie *invalidate*-Signale von den langsameren, am Bus lauschenden Caches erhalten.

Das *write invalidate*-Protokoll erfordert auch, dass die aktuellen Daten jeweils lokalisiert werden können. Bei einem *write-through* Cache ist dies einfach: der Hauptspeicher enthält jeweils eine aktuelle Kopie. Beim *write-back* Cache enthält eventuell nur ein Cache das aktuelle Datum. Bei Anforderung eines Zelleninhaltes über den Bus muss sich also ggf. ein Cache melden und mitteilen, dass ein Datum nicht aus dem Hauptspeicher, sondern aus dem Cache zu lesen ist. Zu diesem Zweck wird auch das Bus-Lauschen eingesetzt.

Der Zugriff auf die *tags* eines Caches kann leicht zu einem Engpass werden, da sowohl Prozessor-seitig wie auch Bus-seitig häufig auf die *tags* zugegriffen werden muss. Hierfür gibt es zwei Lösungen: duplizierte *tags* und Multilevel-Caches. Im Falle eines Zwei-Ebenen-Caches kann der Bus auf den langsameren (L2-) Cache zugreifen, während der Prozessor überwiegend auf dem schnelleren (L1-) Cache arbeitet.

Die *write invalidate*-Methode ist sehr verbreitet.

Beispiel eines *write invalidate*-Protokolls:

Bei dem nachfolgend beschriebenen Protokoll nehmen wir zur Vereinfachung an, das *write misses* wie *write hits* behandelt werden. Dies reduziert die Anzahl der Fallunterscheidungen und vereinfacht den Steuerautomaten der Caches.

Abb. 6.8 zeigt den Steuerautomaten, der für jeden Cache-Block zu realisieren ist (tatsächlich reicht es aus, die Zustandsinformation durch Kontrollbits darzustellen, die jedem Block zugeordnet sind und nur einen einzigen Steuerautomaten zur Aktualisierung der Kontrollbits vorzusehen).

Aufgrund des vorhandenen Zustände kann man sa Protokoll als **MSI**-Protokoll bezeichnen. Aktionen sind in dem Diagramm mit einem '/' von den Bedingungen getrennt. Die Zustandsübergänge mit nicht-kursiver Beschriftung werden durch Vorgänge im Prozessor ausgelöst. Sie würden auch bei einem Mono-Prozessorsystem notwendig sein. Shared entspräche in diesem Fall dem sonst Clean genannten Zustand, in dem sich im Hauptspeicher und im Cache dieselben Informationen befinden, der Hauptspeicher also aktuell ist. Modified entspräche dem Zustand dirty, in dem nur der Cache die aktuellen Daten des Blocks enthält.

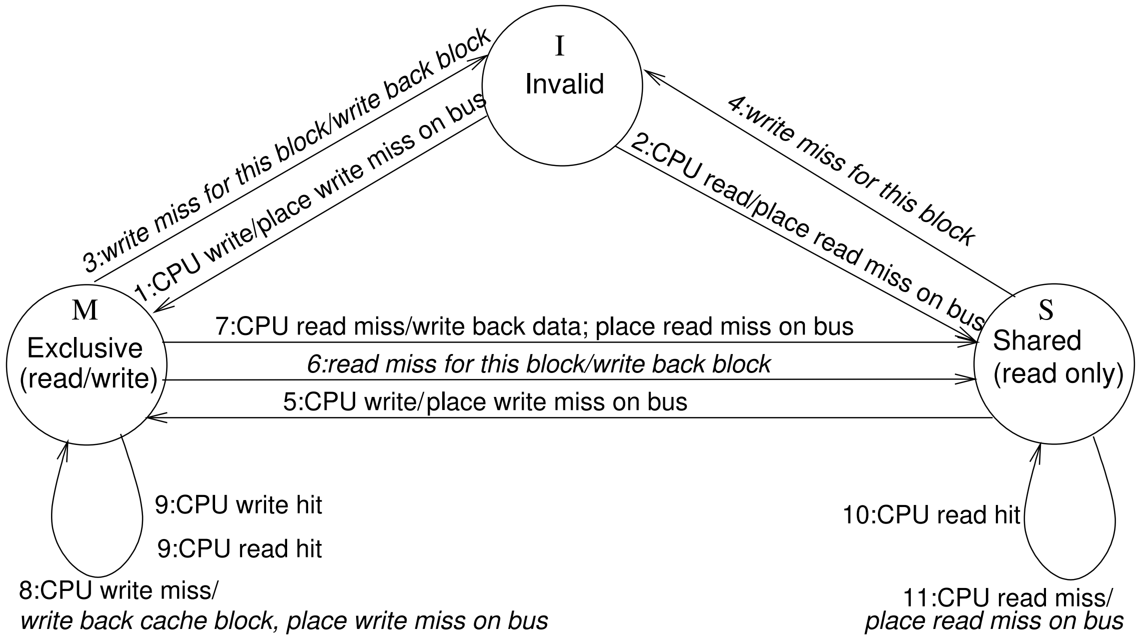


Abbildung 6.8: Zustandsdiagramm des Cache-Controllers für einen Cache-Block

Es wird angenommen, dass die Aktionen place write miss on bus und place read miss on bus für ein Auffüllen der jeweiligen Cache-Blöcke aus dem Speicher sorgen.

Erläuterung der einzelnen Übergänge:

1. **CPU will schreiben; Cacheblock ungültig;**

   Erzeugtes *write miss* bewirkt: *write back* aus Cache-Kopien (Kante 3); Kopien werden ungültig (Kanten 3 und 4); Laden aus dem Speicher in den lokalen Cache.

2. **CPU will lesen; Cacheblock ungültig;**

   Erzeugtes *read miss* bewirkt: Laden aus dem Speicher in den lokalen Cache.

3. **Andere CPU schreibt; R/W-Kopie lokal**

   Aktueller Block wird in Speicher kopiert. (und danach von anderer CPU gelesen)

4. **Andere CPU schreibt; RO-Kopie(n)**

   Lokaler Block wird ungültig.

5. **CPU will schreiben; lokale RO Kopie;**

   Erzeugtes *write miss* bewirkt: Kopien werden ungültig (4:); (unnötiges) Laden; Lokale Kopie geht in R/W-Modus

6. **Andere CPU liest; R/W-Kopie lokal**

   Aktueller Block wird in Speicher kopiert. *read miss* füllt Cache der anderer CPU

7. **Lokale CPU muss Block verdrängen**

   Aktueller Block wird in Speicher kopiert. *read miss* dieser CPU füllt Cache.

8. **Lokale CPU muss Block verdrängen**

   Aktueller Block wird in Speicher kopiert. *write miss* lässt ggf. Daten aus anderen Caches in Speicher kopieren und holt sie von dort. Kopien in anderen Caches werden ungültig.

9. **CPU hat Treffer; R/W-Kopie lokal** CPU schreibt oder liest.

10. **CPU hat Lesetreffer; lokale RO Kopie;** CPU liest

11. **CPU muss RO-Kopie verdrängen** *read miss* dieser CPU füllt Cache.

Beim vorgestellten Protokoll ist jeder gültige Block entweder im Zustand shared in einem oder mehreren Caches oder im Zustand modified in genau einem Cache. Jede Transition in den Zustand modified generiert write miss, was diesen Block in allen anderen Caches in den Zustand invalid überführt. Falls ein anderer Cache den Block im Zustand modified hatte, so generiert dieser ein Zurückschreiben des Blocks (Kante 3). Falls sich ein Block im Zustand modified befindet und für diesen ein read miss auf dem Bus erzeugt wird, so wird der Inhalt des Blocks auf den Bus gelegt und der Controller geht in den Zustand shared. Dieses Protokoll sorgt dafür, dass Blöcke im Zustand shared stets auch im Speicher aktuell sind.

Das beschriebene Protokoll enthält noch einige Vereinfachungen:

- Es wird angenommen, dass die Operationen **atomar** sind, d.h. nicht durch andere Operationen unterbrochen werden. Für write misses wird dies in der Praxis nicht stimmen. Bei einem *split transaction bus* wären auch die read misses nicht atomar zu realisieren. Entsprechende Änderungen am Protokoll lassen *deadlocks* möglich werden, die vermieden werden müssen (siehe [HP96], Anhang E).
- Alle Übergänge in den Zustand modified führen zu einem write miss. Wir nehmen an, dass write misses den Block in den Cache übertragen. Dies ist überflüssig, wenn eigentlich ein write hit vorlag, d.h. wir für diesen Block einen Übergang von shared nach modified hatten. Mit einem weiteren Bus-Signal invalidate können wir dafür sorgen, dass Blöcke für ungültig erklärt werden, aber trotzdem kein Auffüllen des auslösenden Caches mit aktuellen Daten erfolgt. wir annehmen,
- Weiter wäre es sinnvoll, bei Blöcken im Zustand shared zu unterscheiden, ob es sich um gemeinsame oder private Blöcke handelt. So könnte unnötiger Busverkehr vermieden werden. Hierzu könnte man einen Zustand exclusive einführen und käme so zu einem MESI-Protokoll. MESI-Protokolle kommen in vielen kommerziellen Rechnern vor.

6.2.2 Architekturen mit verteiltem gemeinsamen Speicher

Der wesentliche Vorteil von Architekturen mit verteiltem gemeinsamen Speicher ist die Skalierbarkeit. Man muss sich bemühen, diese auch durch die Maßnahmen zur Kohärenzerhaltung nicht zu zerstören. Es gibt dafür mehrere Fälle.

Erstens kann man auf Hardware-unterstützte Kohärenz-Erhaltung völlig verzichten. Der Compiler sorgt in diesem Fall für die Zuordnung von gemeinsamen Variablen zu speziellen Segmenten oder Speicherbereichen, die vom Caching ausgenommen werden. Dies erfordert spezielle Maßnahmen im Compiler und führt bei häufigem Zugriff auf gemeinsame Variablen zu erheblichen Leistungseinbußen. Diese Lösung wird zwar kommerziell benutzt (Cray T3D), sie gilt aber als im Allgemeinen inakzeptabel.

Im Weiteren kann man ausnutzen, dass verzeichnisbasierte Protokolle bekannt sind, deren Aufwand beim Produkt der Anzahl der Blöcke und der Anzahl der Prozessoren liegt. Für nicht zu große Systeme ist dieser Aufwand akzeptabel.

Ein verzeichnisbasiertes Protokoll muss im Wesentlichen zwei Operationen realisieren: die Behandling von *read misses* und Schreiben in einen lokal vorhandenen, gemeinsamen Cache-Block. *Write misses* sind einfach eine Kombination dieser beiden Operationen. Zur Realisierung dieser Operationen muss im Verzeichnis der jeweilige Zustand der Cache-Blöcke protokolliert werden. Mögliche Zustände sind:

- *Shared*: Mindestens ein Prozessor hält den Block im Cache und die Caches wie auch der Speicher enthalten den aktuellen Wert,
- *Uncached:* Kein Prozessor hat eine Kopie,
- *Modified:* Genau ein Prozessor hat den Block geschrieben, dieser Prozessor heißt **Eigner** des Blocks. Der Speicher ist nicht aktuell.

Es muss auch protokolliert werden, welche Prozessoren eine Kopie eines Blocks haben. Die einfachste Lösung hierfür besteht aus einem Bitvektor, in dem für jeden Prozessor eingetragen wird, on er eine Kopie des Blocks enthält.

Die Zustände und die Übergänge entsprechen im Wesentlichen denen eines *snooping caches*. Allerdings können *invalidate*-Nachrichten jetzt nicht mehr einfach über den Bus an alle Caches versandt werden. Vielmehr müssen diese Nachrichten über ein Verbindungsnetzwerk an genau die Caches geschickt werden, die auch eine Kopie enthalten. Aufgrund der Nachrichten-Orientierung müssen explizite Bestätigungen erfolgen, damit der Abschluss von Operationen bekannt ist. Eine weitere Schwierigkeit ergibt sich durch den Fortfall der Arbitrierung von konkurrierenden Schreibvorgängen mit Hilfe des Bus-Arbiters. Im Übrigen entsprechen die Vorgänge weitgehend denen in *snooping caches*.

6.3 Synchronisation

Programmen kann die Benutzung von Ressourcen (wie Speicherzellen, Geräten, Dateien, Verzeichnissen usw.) gestattet werden. In einer Reihe von Fällen muss für gewisse Programmabschnitte garantiert werden, dass den Programmen die Ressourcen **exklusiv** zugeteilt werden, d.h. keine anderen Programme dürfen in der Zeit vom Eintritt in diesen Abschnitt bis zum Verlassen dieses Abschnittes auf diese Ressource zugreifen. Weder darf ein anderes Programm nach einem Prozesswechsel auf demselben Prozessor auf die Ressource zugreifen noch darf in einem Mehrprozessorsystem ein anderer Prozessor (auch kein E/A-Prozessor) auf diese Ressource zugreifen. Beispielsweise muss die Manipulation von Verzeichnissen oder Dateien im Allgemeinen durch ein einziges Programm erfolgen; sonst könnten sich inkonsistente Zustände ergeben.

Def. Ein **kritischer Abschnitt** ist ein Programmabschnitt, in dem einem Programm exklusiver Zugriff auf eine Ressource garantiert sein muss.

Der exklusiver Zugriff kann über eine **Sperre** (*lock*) realisiert werden. Beim Setzen der Sperre muss geprüft werden, ob die Sperre bereits durch ein anderes Programm (einen anderen Prozess) gesetzt ist und fall nicht, dann kann die Sperre gesetzt werden. Zwischen den beiden Operationen für ein Programm darf keine andere Operation auf derselben Sperre durch ein anderes Programm durchgeführt werden; sonst könnten mehrere Programme wieder Zugang zur Ressource erhalten. Die beiden Operationen müssen also **ununterbrechbar** oder **atomar** ausgeführt werden. Das Setzen der Sperre erfordert i.d.R. Hardware-Unterstützung zur Realisierung der **Unteilbarkeit** (oder **Atomizität**) der Operationen.

Eine mögliche Unterstützung ist der *test-and-set*-Befehl tsl. Ununterbrechbar kann mit ihm der Inhalt einer Speicherzelle einem Register zugewiesen werden und der Inhalt der Speicherzelle auf 1 gesetzt werden:

< Reg := Speicher[x]; Speicher[x]:=1 > (unteilbar)

Findet man nach Ausführung des Befehls im Register bereits eine 1 vor, so war die Sperre bereits gesetzt und der exklusive Zugriff wurde nicht erreicht. Man muss dann den Versuch wiederholen (sog. *spin lock* oder *busy waiting*[5]). Eine Anwendung auf drei Prozesse P1-P3 könnte wie folgt aussehen:

```
 P1                P2                P3
try:              try:              try:
 tsl $r,x          tsl $r,x          tsl $r,x
 bne $r,0,try      bne r,0,try       bne $r,0,try
 . krit.Abschn.    . krit.Abschn.    . krit.Abschn.
 li $r,$0          li $r,0           li $r,0
 st $r,x           st $r,x           st $r,x
```

Diese Realisierung ist in dieser Form nur für kurze Abschnitte, die sicher wieder verlassen werden, geeignet.

Eine ähnliche Funktionalität bietet der *swap*- oder *atomic exchange*-Befehl, bei dem atomar eine Speicherzelle und ein Register vertauscht werden.

Beide Befehle kann man bei Monoprozessoren durch eine besondere Form des Speicherzugriffs realisieren, bei der zwischen beiden Operationen der Speicherbus nicht wieder freigegeben wird (so wird dies beim Motorola 68000 gemacht).

[5] In der Praxis wird man versuchen, durch Betriebssystemsfunktionen *busy waiting* zu vermeiden, um den Prozessor nicht unnötig zu belasten.

Die erwähnten Befehle sind im Fall von Multiprozessorsystemen mit Caches schwierig zu implementieren, denn sie verlangen die Realisierung eines Speicherlesens und eines Speicherschreibens in einer ununterbrechbaren (atomaren) Sequenz. In Kombination mit der geforderten Cache-Kohärenz ist dieses schwierig. Ein weiteres Problem ist, dass tsl auch im erfolglosen Fall in den Speicher schreibt, im Fall von mehreren sich bewerbenden Prozessoren also die *ownership* zwischen den Caches ständig wechselt.

Eine Alternative bietet die Realisierung eines Paars von Befehlen, für die aus dem Ergebnis der Ausführung des zweiten Befehls geschlossen werden kann, ob die Sequenz ohne Unterbrechung ausgeführt wurde. Benutzt wird ein Paar aus den Befehlen *load linked* und *store conditional*:

ll $r,x ; load linked r,x
....
sc $r,x ; store conditional
(r=1 im erfolgreichen Fall, =0 sonst)

Wenn die Speicherzelle x vor dem Lesen durch den *store conditional*-Befehl benutzt wurde, dann liefert der *store conditional*-Befehle eine 0. Wenn zwischen beiden Befehlen ein Kontextwechsel erfolgt ist, so wird ebenfalls eine 0 abgeliefert. Ansonsten liefert der *store conditional*-Befehl eine 1. Es stellt sich sofort die Frage, wie man ein solches Verhalten realisiert. Man kann sich ja nicht für jede Speicherzelle merken, welcher Prozessor diese referenziert hat. Lösen lässt sich dies Problem durch den Zustand exclusive eines MESI-Protokolls. Im Wesentlichen muss der ll-Befehl den Zustand exclusive herstellen und sc muss diesen vorfinden (hierbei gibt es allerdings noch einige Details zu beachten). Die Befehle, die zwischen beiden Befehlen erlaubt sind, sind prozessorspezifisch, auf jeden Fall aber einfach. Wichtig ist: es gibt keine *write invalidates* im erfolglosen Fall.

Eine atomare Vertauschung des Registers 4 und der durch Register 5 spezifizierten Speicherzelle wird durch folgende Assemblersequenz bewirkt:

```
try: add $3,$0,$4          ; move-Befehl: R3:= (0+) R4
     ll  $2,0($5)          ; load linked; R2:=Speicher[R5]
     sc  $3,0($5)          ; store conditional
     beq $3,$0,try         ; springe wenn 0
     add $4,$0,$2          ; kopiere gelesenen Wert nach R4
```

Auf der Basis dieser Befehle kann auch aktives Warten auf die Freigabe einer Ressource (einer Sperre) realisiert werden. Dabei sollte allerdings vermieden werden, dass auch dann in den Speicher geschrieben wird, wenn keine 0 gelesen wurde. Besser ist es, im Falle einer gelesenen 0 nur den Leseversuch zu wiederholen. Das folgende Programmstück zeigt ein derart realisiertes aktives Warten auf die Freigabe einer Ressource bzw. einer Sperre.

Annahme: Adresse der Sperre in Register 3.

```
lockit:
 ll   $2,0($3)    ; load linked
 bne  $2,$0,lockit; gesperrt: springe zurück
 li   $2,1        ; generiere eine 1
 sc   $2,0($3)    ; store conditional
 beq  $2,$0,lockit; zurück, wenn Fehlschlag
```

Ein Vorteil ist, dass der ll-Befehl eine lokale Kopie der Sperre im Cache liest, solange diese =1 ist.

Ein Problem ist, dass die Freigabe einer Sperre (Speichern einer 0) zu einem enormen Bus-Verkehr führt, der die Leistung des Systems erheblich herabsetzen kann. Zur Abhilfe wird in [HP96] vorgeschlagen, nach jedem Fehlschlag eine Zeit zu warten, die exponentiell mit der Anzahl der Fehlschläge zunimmt:

```
        li   $3,1              ; initiale Verzögerung
lockit: ll   $2,0($4)          ; load linked
        bne  $2,$0,lockit      ; springe zurück
        addi $2,$2,1           ; generiere eine 1
```

```
        sc   $2,0($4)          ; store conditional
        bne  $2,$0,gotit       ; springe, wenn Befehl erfolgreich
        sll  $3,$3,1           ; verdopple Verzögerung
        pause $3               ; BS-Aufruf
        j    lockit            ; noch einmal
gotit   Benutze die gesicherte Ressource
```

Diese Lösung bietet eine kurze Verzögerung bei wenig Wettbewerb um gemeinsame Ressourcen und vermeidet Kommunikationsengpässe bei starkem Wettbewerb. Diese 'Lösung' ist für Realzeit-Systeme aber überhaupt nicht zu gebrauchen.

Eine (wohl bessere) Lösung ist der Aufbau einer Warteschlange von Prozessoren, die auf die Freigabe einer Ressource warten. Eine solche Warteschlange kann sowohl in Hardware wie auch in Software realisiert werden [HP96].

6.4 Speicherkonsistenz

Speicherkohärenz allein gibt noch keine Aussage darüber, wann geschriebene Werte von den beteiligten Prozessoren gelesen werden. Das folgende Beispiel zweier Prozesse P1 und P2 belegt, dass dies zu unerwartetem Verhalten von parallelen Programmen führen kann.

```
P1:     a:=0;            P2:    b:=0;
        ....                    ....
        a:=1;                   b:=1;
L1:     if (b=0) ...     L2:    if (a=0) ...
```

`a` und `b` seien darin gemeinsame Variable der Prozesse `P1` und `P2`. Die Frage ist: kann die `if`-Abfrage für beide Prozesse positiv ausfallen? Intuitiv würden wir sagen: nein. Denn wenn `P2` zur Ausführung seines `if`-Statements kommt, dann sollte `b` bereits 1 sein und der Test des `if`-Statements von `P1` sollte fehlschlagen.

Tatsächlich aber haben wir bislang nicht gefordert, dass zwischen dem Schreiben und dem Lesen von verschiedenen Variablen irgendwelche zeitlichen Ordnungen eingehalten werden müssen. Es könnte die Schreiboperation b:=1 so stark verzögert sein (z.B. durch Schreibpuffer), dass der Test in P1 noch positiv ausfällt, obwohl P2 inzwischen bei der Ausführung seines if-Statements angekommen ist.

Das einfachste Forderung, welche diese Möglichkeit ausschließt, heißt **sequenzielle Konsistenz** (engl. *sequential consistency*):

Def.: *Sequential consistency requires that the result of any execution be the same as if the accesses executed by each processor were kept in order and the accesses among different processors were interleaved* [HP96].

Sequentielle Konsistenz verlangt also, dass die Reihenfolge der Speicherzugriffe der Einzelprozessoren der Reihenfolge in deren Programmen entspricht (*program order*, siehe auch [TM93]). Der o.a. Fall eines unerwarteten Programmverhaltens kann damit nicht auftreten, da für jeden Prozess erst die Zuweisungen abgeschlossen werden, bevor die Tests ausgeführt werden.

Sequentielle Konsistenz vermeidet also das unerwartete Programmverhalten, ist aber so restriktiv, dass es die Leistung eines Rechners sehr negativ beeinflusst. So dürfen beispielsweise schon nicht einmal mehr Schreibpuffer oder *prefetching* realisiert werden.

Es ist daher notwendig, Konsistenzmodelle zu entwickeln, welche die Leistung eines Systems nicht zu sehr herabsetzen und die gleichzeitig Programmierern einfach zu erklären sind. Ein solches Modell ist das der **synchronisierten Programme**.

Def.: Ein Programm heißt **synchronisiert** oder *data race free*, wenn alle Zugriffe auf gemeinsame Daten mittels Synchronisation geordnet sind.

Def.: Ein Zugriff auf Daten heißt **durch Synchronisation geordnet**, wenn zwischen dem Schreiben einer (gemeinsamen) Variablen durch einen Prozessor und einem Zugriff auf dieselbe Variable durch einen anderen

Prozessor stets ein Paar von Synchronisationsoperationen liegt, von denen eine nach dem Schreiben durch den ersten Prozessor und eine vor dem Zugriff durch den anderen Prozessor erfolgt.

Eine der Synchronisationsoperationen räumt das Recht ein, auf eine Variable zuzugreifen. Wir nennen sie *aquire*-Operation. Die zweite Synchronisationsoperation gibt das Recht, auf eine Variable zuzugreifen, wieder frei. Wir nennen sie *release*-Operation. *lock*- und *unlock*-Operationen sind spezielle Fälle von *aquire*- und *release*-Operationen.

Unter Benutzung von *aquire*- und *release*-Operationen können wir festhalten, dass ein Programm synchronisiert ist, wenn Zugriffe auf gemeinsame Variable immer die Form

```
write(x)
...
release(s)
...
acquire(s)
...
access(x)
```

haben.

Eine Einschränkung der Reihenfolgen der Zugriffe auf Speicher gelingt mit den sog. **Lesezäunen** (engl. *read fences*) bzw. **Schreibzäunen** (engl. *write fences*). Ein von einem Prozessor P ausgeführter Schreibzaun S stellt sicher, dass alle gemäß *program order* vor S liegenden Schreibvorgänge abgeschlossen werden und vor dem Ende der Ausführung von S kein gemäß *program order* nachfolgendes Schreiben gestartet wird. Für Lesezäune gilt das Entsprechende.

Ein **Speicherzaun** ist eine Operation, die sowohl als Lese- wie auch als Schreibzaun wirkt.

Bei sequenzieller Konsistenz sind alle Lesevorgänge *read fences* und alle Schreibvorgänge *write fences*.

Die Beziehungen zwischen verschiedenen abgeschwächten Konsistenzmodellen kann man am Besten verstehen, wenn man Reihenfolgen von Lese- und Schreibvorgängen betrachtet. Es gibt vier derartige Ordnungen:

1. Lesen gefolgt von Lesen (R→R),
2. Lesen gefolgt von einem Schreiben (R→W)

   Diese Reihenfolge wird in allen Prozessoren eingehalten, wenn es sich um dieselbe Zelle (Adresse) handelt. Es ist dies der Fall der Antidatenabhängigkeit (siehe Kapitel 3.1).
3. Schreiben gefolgt von einem Schreiben (W→W)

   Diese Reihenfolge wird in allen Prozessoren eingehalten, wenn es sich um dieselbe Zelle handelt. Es ist dies der Fall der Ausgabeabhängigkeit (siehe Kapitel 3.1).
4. Schreiben gefolgt von einem Lesen (W→R)

   Diese Reihenfolge wird eingehalten, wenn es sich um dieselbe Zelle handelt. Es ist dies der Fall der Datenabhängigkeit (siehe Kapitel 3.1).

Auf dieser Basis definieren wir verschiedene Konsistenzprotokolle:

1. **Sequenzielle Konsistenz:**

   Sequenzielle Konsistenz verlangt, dass alle vier Reihenfolgen eingehalten werden, unabhängig von der Adresse.

   Sequentielle Konsistenz ist einfach zu realisieren. Wenn der Prozessor anhält, bis alle Operationen abgeschlossen sind und wenn alle Kopien in Caches invalidiert werden, bevor ein Schreiben erfolgt, dann laufen alle Speicherzugriffe mit der vom Programm her vorgegebenen Ordnung (*program order*) ab.

2. ***Processor consistency*** oder ***total store ordering*** (TSO):

   Bei der ersten Abschwächung der sequenziellen Konsistenz wird die Reihenfolge 'Schreiben gefolgt von einem Lesen' der *program order* für unterschiedliche Adressen nicht beibehalten. Dies erlaubt, Schreibvorgänge durch Lesevorgänge überholen zu lassen und macht damit Schreibpuffer und *prefetching* für Lesevorgänge möglich. Explizite Synchronisationsoperationen sind notwendig, um sicherzustellen, dass alle Schreibvorgänge vor einer bestimmten Leseoperation ausgeführt werden. Dieses Modell heißt *processor consistency* oder *total store ordering* (TSO).

   Dieses Modell kann mit Schreibpuffern realisiert werden, die lediglich überprüfen müssen, ob eine ausstehende Schreiboperation eine Zelle beschreiben soll, für die auch ein Leseauftrag ansteht. Auf diese Weise kann die Latenzzeit des Schreibens teilweise versteckt werden.

3. ***Partial store ordering***:

   Bei der nächsten Abschwächung wird zusätzlich die Reihenfolge 'Schreiben gefolgt von einem Schreiben' nicht beibehalten. Dieses Modell heißt *partial store ordering*. Dieses Modell erlaubt Implementierungen mit mehreren Fließbändern, in denen sich Schreibvorgänge überholen können. Ein Anhalten einer Schreiboperation muss weiterhin bei Ausführung von Schreibzäunen erfolgen.

4. ***Weak ordering***:

   Bei der dritten Abschwächung werden auch die beiden anderen Reihenfolgen nicht respektiert. Dieses Modell ignoriert alle Reihenfolgen der *program order*, abgesehen von solchen, welche Synchronisationsoperationen betreffen. Dieses Modell heißt *weak ordering*. *Weak ordering* erlaubt die verzögerte Ausführung von Leseoperationen. Ein Leistungsgewinn kann dadurch höchstens im Falle eines *caches misses* beim Lesen erzielt werden und dies nur dann, wenn *cache read misses* nicht den Cache blockieren, also die weitere Bearbeitung von Zugriffen während des Ladevorgangs von Speicher möglich ist. Generell beruht der Leistungszuwachs der schwächeren Konsistenzmodelle v.a. auf dem Verstecken der Bearbeitungszeit von Schreiboperationen (*hiding of write latency*), weniger auf dem Verstecken der Bearbeitungszeit von Leseoperationen (*hiding of read latency*).

5. ***Release consistency***:

   Bei der vierten Abschwächung sprechen wir von *release consistency*. *Release consistency* geht von der Annahme aus, dass in allen synchronisierten Programmen der Zugriff auf gemeinsame Daten durch *aquire*- und *release*-Operationen eingerahmt ist. Dann ist es möglich, Schreib- und Lese-Operationen vor einem *aquire* nicht abzuschliessen und nicht notwendigerweise erst nach einem *release* zu starten. Nur zwischen den *aquire*- und *release*-Operationen und zu den von ihnen eingeschlossenen Operationen gibt es einzuhaltende Reihenfolgebedingungen.

   Release consistency erlaubt es, weitere Schreib-Latenzzeit zu verstecken. Falls nicht-blockierende Leseoperationen unterstützt werden, so kann auch die Latenzzeit dieser Operationen versteckt werden. Bei *release consistency* können Schreibvorgänge bereits vor dem Invalidieren aller Kopien erfolgen. Abhängige Leseoperationen können dadurch früher starten.

Tabelle 6.3 zeigt eine Übersicht der bei verschiedenen Konsistenzmodellen eingehaltenen Reihenfolgen.

Modell	Benutzung	Eingehaltene Reihenfolgen	Synchronisationsreihenfolgen
Sequential consistency	Optional bei den meisten Maschinen	R → R, R → W, W → R, W → W	S→W, S→R, R→S, W→S, S→S
Total store order oder *processor consistency*	IBM S/370, VAX, SPARC	R → R, R → W, W → W	S→W, S→R, R→S, W→S, S→S
Partial store order	SPARC	R → R, R → W	S→W, S→R, R→S, W→S, S→S
Weak ordering	PowerPC		S→W, S→R, R→S, W→S, S→S
Release consistency	Alpha, MIPS		S_A→W, S_A→R, R→S_R*, W→S_R, S_A→S_A, S_A →S_R, S_R →S_A, S_R →S_R

Tabelle 6.3: Bei den verschiedenen Konsistenzmodellen eingehaltene Reihenfolgen

Abb. 6.9 zeigt die eingehaltenen Reihenfolgen an einem Beispiel[6].

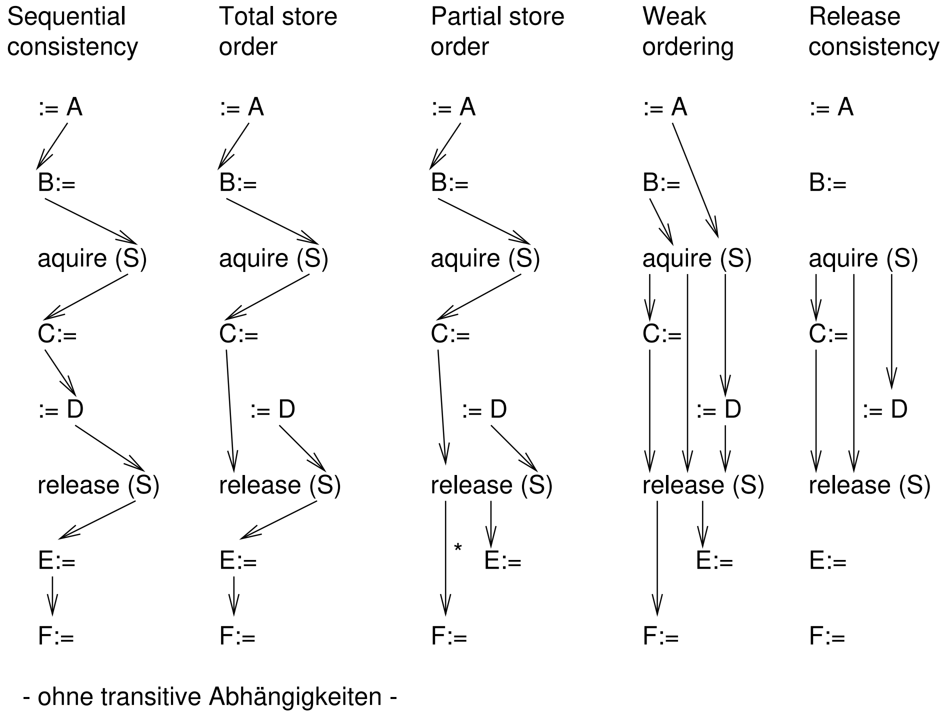


Abbildung 6.9: Beispiel der bei verschiedenen Konsistenzmodellen eingehaltenen Reihenfolgen

Es können Konsistenzmodelle definiert werden, die noch schächer sind, als *release consistency.* Allerdings ist ihr Realisierungsaufwand hoch und die gewonnenene Leistung gering [HP96].

Die in den Konsistenzmodellen vorgeschriebenen Eigenschaften können mit verschiedenen Verfahren realisiert werden. Jedes dieser Konsistenzverfahren heißt **Konsistenzprotokoll**,

Im Folgenden wird die Leistung von Rechensystemen mit verschiedenen Konsistenzprotokollen verglichen. Dabei werden vier verschiedene Hardwaremodelle benutzt:

1. SSBR: Statisches Scheduling mit blockierendem Lesen,
2. SS: Statisches Scheduling ohne blockierendes Lesen,
3. DS16: Dynamisches Scheduling mit *reorder buffer* mit 16 Einträgen,
4. DS64: Dynamisches Scheduling mit *reorder buffer* mit 64 Einträgen. Dieser *reorder buffer* ist groß genug, um potenziell die Latenzzeit des Caches vollständig zu verbergen.

Für alle vier Modelle gelten die folgen Werte:

- mit jedem Takt wird ein Befehl gestartet,
- *cache misses* benötigen 50 Takte,
- der Prozessor hat einen Schreibpuffer der Tiefe 16,
- die Caches haben eine Größe von 64 kB und eine Block-Größe von 16 Bytes.

Abb. 6.10 zeigt die Leistung von zwei Benchmark-Programmen für die vier Prozessorvarianten bei verschiedenen Konsistenzmodellen[7].

[6] In Abb. 6.9 und in Tabelle 6.3 bezeichnet * Stellen, an denen [HP96] jeweils einen Fehler enthält.
[7] Die Werte beziehen sich offenbar auf 16 Prozessoren.

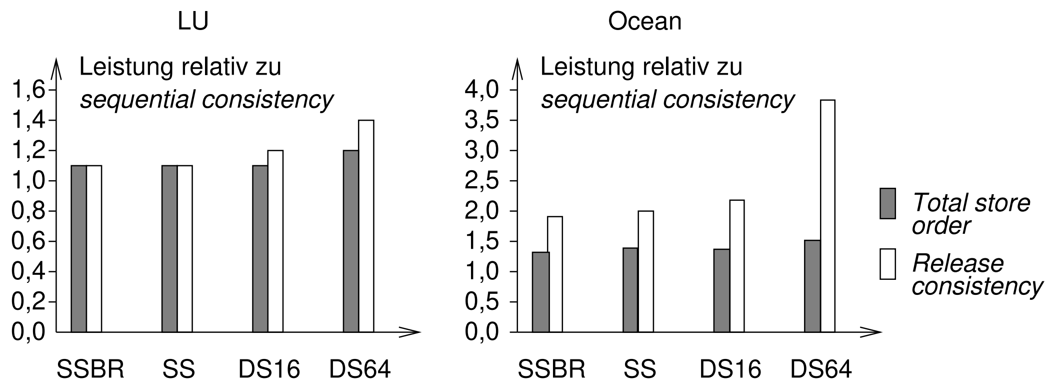


Abbildung 6.10: Vergleich der Leistung verschiedener Konsistenzprotokolle

Die Beispiele sind allerdings nicht realistisch: Designer würden die Cache-Größe erhöhen, bevor sie dynamisches Scheduling und nicht-blockierende Caches realisieren. Bei einer größeren Cache-Größe würden die Vorteile der schwachen Konsistenzmodelle geringer ausfallen.

6.5 Bewertung

Die beschriebenen Techniken zeigen, dass die Realisierung von Hochleistungs-Rechensystemen recht komplex ist und man muss sich gelegentlich fragen, ob die Richtung eigentlich noch stimmt. Mit Datenflussrechnern oder automatisch parallelisierbaren funktionalen Programmen ließen sich manche komplexen Implementierungsdetails vermeiden. Den Widersinn der gegenwärtigen mehrfachen Extraktion möglicher Ausführungsreihenfolgen haben Šilc, Robič und Ungerer bereits im Kontext des Prozessors superskalarer Rechner sehr gut auf den Punkt gebracht [SRU99]:

But why should

- *a* programmer
  - *design a* partially *ordered algorithm,*
  - *and then code the algorithm in* total *ordering because of the use of a sequential von Neumann language,*
- the compiler
  - *regenerate the* partial *order in a dependence graph,*
  - *and then generate a reordered 'optimized' sequential machine code,*
- the microprocessor
  - *dynamically regenerate the partial order in its out-of order execution, execute due to a* micro data flow *principle,*
  - *and then reestablish the unnatural serial program order for in-order commitment in the retire stage?*

Es bleibt abzuwarten, ob sich grundsätzliche Änderungen in der Arbeitsweise von Rechensystemen ergeben oder ob einfach nur die bisherigen Systeme immer komplexer werden.

Literaturverzeichnis

[Abe91] B.W. Abeysundara. High-speed local area networks and their performance: A survey. *acm computing surveys*, 23:221, 91.

[ABM+93] C. Albrecht, S. Bashford, P. Marwedel, A. Neumann, and W. Schenk. The design of the PRIPS microprocessor. *4th EUROCHIP-Workshop on VLSI Training*, 1993.

[ANH+93] A. Alomary, T. Nakata, Y. Honma, M. Imai, and N. Hikichi. An ASIP instruction set optimization algorithm with functional module sharing constraint. *Int. Conf. on Computer-Aided Design (ICCAD)*, pages 526–532, 1993.

[Ass98] Semiconductor Industry Association. The national technology roadmap for semiconductors. *www.semichips.org*, 1998.

[Bae80] J.L. Baer. *Computer Systems Architecture.* Pitmen, 1980.

[Bae94] H. Baehring. *Mikrorechnersysteme.* Springer, 1994.

[Beh99] B. Behr. Am laufenden band. *c't, Nov. 1999*, 1999.

[BH80a] A. Bode and W. Händler. *Rechnerarchitektur.* Springer, 1980.

[BH80b] A. Bode and W. Händler. *Rechnerarchitektur II.* Springer, 1980.

[BK95] V. Bhaskaran and K. Konstantinides. Multimedia enhancements for RISC processors. *in: Image and Video Compression Standards, Kluwer Acad. Publishers*, 1995.

[BN71] Bell and Newell. *Computer Structures.* McGraw-Hill, 1971.

[Bod] A. Bode. *Risc-Architekturen.* Bibliographisches Institut, Mannheim.

[BS88] L. Bic and A. C. Shaw. *The logical design of operating systems.* Prentice-Hall, 1988.

[CEF+95] H. P. Cinka, J. Ebert, B. Fischer, J. Freytag, and R. Gunzenhäuser. Empfehlung des Fachbereichs 7 (Ausbildung und Beruf) der Gesellschaft für informatik zur weiterbildung für Informatiker durch die Hochschulen. *Informatik-Spektrum*, pages 106–109, 1995.

[CLG+94] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson. RAID: High-performance, reliable secondary storage. *acm computing surveys*, 26:145–186, 1994.

[Coh81] Cohen. On holy wars and a plea for peace. *IEEE Computer*, 1981.

[Coy92] W. Coy. *Aufbau und Arbeitsweise von Rechenanlagen.* Vieweg, 1992.

[CS99] D. Culler and J. P. Singh. *Parallel Computer Architecture - A hardware / software approach.* Morgan Kaufman Publishers, 1999.

[Dal97] M.K. Dalheimer. *Java Virtual Machine - Sprache, Konzept, Architektur.* O'Reilly, 1997.

[Den80] J. Dennis. Data flow supercomputers. *IEEE Computer*, page 48, 1980.

[DSTG93] R. Dittmann, T. Stock, and P. Tran-Gia. Das DQDB-zugriffsprotokoll in hochgeschwindigkeitsnetzen und der IEEE-standard 802.6. *Informatik-Spektrum*, pages 143–158, 1993.

[Ebe93] J. Eberspächer. Schwerpunktthema: Hochschwindigkeitsnetze. **it + ti** *(Informationstechnik und Technische Informatik*, 4/93, 1993.

[Fis81] J.A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Trans. on Computers, Vol.C–30*, pages 478–790, 1981.

[Fis93] W. Fischer. Datenkommunikation mittels ATM – Architekturen, Protokolle, betriebsmittelverwaltung. **it + ti** *(Informationstechnik und Technische Informatik)*, pages 3–11, 1993.

[Gil81] W. Giloi. *Rechnerarchitektur.* Springer, 1981.

[Gol91] D. Goldberg. What every computer scientist should know about floating point arithmetic. *acm computing surveys, Vol. 23*, page 5, 1991.

[Hay79] J.P. Hayes. *Computer Architecture and Organization.* McGraw-Hill, 1979.

[Hof77] R. Hoffmann. *Rechenwerke und Mikroprogrammierung.* Oldenbourg, 1977.

[HP95] J. L. Hennessy and D. A. Patterson. *Computer Organization – The Hardware/Software Interface.* Morgan Kaufmann Publishers Inc., 1995.

[HP96] J. L. Hennessy and D. A. Patterson. *Computer Architecture – A Quantative Approach.* Morgan Kaufmann Publishers Inc.,2. Aufl., 1996.

[HQ89] W. Heise and P. Quattrocchi. *Informations- und Codierungstheorie.* Springer, 2. Auflage, 1989.

[HS93] B. Huber and G. Schnurrer. SCSI 1-2-3. *c't, Nov.*, 1993.

[HT93] W.D. Hillis and L. W. Tucker. The CM-5 connection machine: A scalable supercomputer. *Communications of the ACM*, pages 31–49, 1993.

[IEE88] Design Automation Standards Subcommittee of the IEEE. IEEE standard VHDL language reference manual (IEEE Std. 1076-87). *IEEE Inc., New York*, 1988.

[IEE92] Design Automation Standards Subcommittee of the IEEE. Draft standard VHDL language reference manual. *IEEE Standards Department, 1992*, 1992.

[Jes75] E. Jessen. *Architektur digitaler Rechenanlagen.* Springer, 1975.

[Joh00] R. Johnson. RNA computer clears 10-bit hurdle. *EETimes (www.eet.com), 1.2.2000*, 2000.

[JS92] D. Jungmann and H. Stange. *Einführung in die Rechnerarchitektur.* Hanser, 1992.

[Kai89a] R. Y. Kain. *Computer Architecture Vol. I.* Prentice-Hall, 1989.

[Kai89b] R. Y. Kain. *Computer Architecture Vol. II.* Prentice-Hall, 1989.

[Kog91] Peter M. Kogge. *The architecture of symbolic computing.* McGraw-Hill, 1991.

[Kuc78] D.J. Kuck. *The Structure of Computers and Computations, Vol. I.* Wiley, 1978.

[Lev00] John Levine. Linkers and loaders. *zur Publikation bei Morgan-Kaufman vorgesehen, Vorab-Version unter //www.iecc.com/linker*, 2000.

[Lie80] H. Liebig. *Rechnerorganisation.* Springer, 1980.

[Lin98] C. Lindemann. *Performance Modelling with Deterministic and Stochastic Petri Nets.* Wiley, 1998.

[Mal78] P.W. Mallett. Methods of compacting microprograms (dissertation). *University of Southwestern Louisiana, Lafayette*, 1978.

[MC80] C. Mead and L. Conway. Introduction to VLSI systems. *Addison–Wesley*, 1980.

[MSS91] Müller-Schloer and Schmitter. *RISC-Workstation-Architekturen.* Springer, 1991.

[PB61] W.W. Peterson and D.T. Brown. Cyclic codes for error correction. *Proc. of the IRE*, pages 228–235, 1961.

[Pet00] C. Peterson. Taking technology to the molecular level. *IEEE Computer, Jan. 2000*, pages 46–53, 2000.

[PWW97] A. Peleg, S. Wilkie, and U. Weiser. Intel MMX for multimedia PCs. *Communications of the ACM*, pages 25–38, 1997.

[Sch] Schütt. *Feldrechner*. Springer.

[Sch73] H. Schecher. *Funktioneller Aufbau digitaler Rechenanlagen*. Springer, 1973.

[Sch83] U. Schmidt. Ein neuartiger, auf VDM basierender Codegenerator-Generator. Technical Report 4/83, Computer Science Dpt., University of Kiel, 1983.

[Sch93] G. Schnurrer. Moderne PC-Bussysteme. *c't, Okt.*, 1993.

[SK83] C. Schmittgen and W. Kluge. A system architecture for the concurrent evaluation of applicative program expressions. *10th Int. Symp. on Computer Arch*, 1983.

[Spa76] O. Spaniol. *Arithmetik in Rechenanlagen*. Teubner, 1976.

[SR00] A.M. Steane and E.G. Rieffel. Beyond bits: The future of information processing. *IEEE Computer, Jan. 2000*, pages 38–45, 2000.

[SRU99] J. Silc, B. Robic, and T. Ungerer. *Processor Architecture - From Dataflow to Superscalar and Beyond*. Springer Verlag, 1999.

[Sta94] W. Stallings. *Data and Computer Communications*. MacMillan, 1994.

[Sta95] W. Stallings. *Operating Systems*. Prentice Hall, 1995.

[Sti94] A. Stiller. EPP, ECP ... etc. – Druckerport-Metamorphosen. *c't*, 1994.

[Sto80] H.S. Stone. *Introduction to Computer Architecture*. 1980.

[Sun97] Sun Microsystems. picojava^{TM} i microprocessor core architectur. *http://www.sun.com/sparc/-whitepapers/wpr-0014-01/index.html*, 1997.

[Tan76] A.S. Tanenbaum. *Structured Computer Organization*. Prentice Hall, 1976.

[Tan92] A. Tanenbaum. *Modern Operating Systems*. Prentice Hall, 1992.

[Tea82] P. Treleaven and et al. Data-driven and demand-driven computer architectures. *acm computing surveys*, pages 93–144, 1982.

[TM93] M. Tomašević and V. Milutović. *The Cache Coherence Problem in Shared-Memory Multiprocessors: Hardware Solutions*. IEEE Computer Society Press, 1993.

[Tra96] Fa. Transtec. Info-Teil. *Gesamtkatalog Frühjahr*, 1996.

[Tre84] P. Treleaven. Future computers: Logic,...,data flow, control flow. *IEEE Computer*, pages 47–55, 1984.

[Veg84] S. Vegdahl. A survey of proposed architectures for the execution of functional languages. *EEEE Trans. Comp.*, page 1050, 1984.

[Weg89] I. Wegener. Effiziente Algorithmen für grundlegende Funktionen. *Teubner-Verlag*, 1989.

[WZ93] G. Weikum and P. Zabback. I/O-Parallelität und Fehlertoleranz in Disc-Arrays. *Informatik-Spektrum*, pages 133–142, 1993.

[ZH74] G. Zimmermann and J. Höffner. *Elektrotechnische Grundlagen der Informatik II*. Bibliographishes Institut Mannheim, 1974.

Index