Skript zur Vorlesung „Rechnerstrukturen, Teil 2” WS 2013/2014

P. Marwedel, Informatik 12
Technische Universität Dortmund

8. September 2013

Inhaltsverzeichnis

Kapitel 2

Teil 2 der Vorlesung

2.1 Einleitung

"*Die Wissenschaft Informatik befasst sich mit der Darstellung, Speicherung, Übertragung und Verarbeitung von Information*" [Ges06]. In diesem Kapitel geht es um die Verarbeitung.

Die Beschäftigung mit Rechnerarchitekturen kann mit der Tatsache motiviert werden, dass alle Formen der Informationsverarbeitung letztlich auf irgendeiner Form von technischen Systemen realisiert werden müssen. Für die heute üblichen Programme sind das konventionelle Rechner, wie wir sie in dieser Vorlesung kennen lernen werden. Diese Rechner bilden **Ausführungsplattformen** (engl. *execution platforms*), auf denen Programme ausgeführt werden können. Für verschiedene Anwendungsbereiche, wie z.B. die Motorsteuerung oder Handys, versucht man derzeit, standardisierte Ausführungsplattformen bereitzustellen. *Platform-based design* ist ein Thema auf vielen großen Konferenzen. Auch stärker in Richtung Software orientierte Studierende sollten sich mit dem Thema beschäftigen, da Plattformen in der Regel kein ideales Verhalten zeigen. Vielmehr muss man einen Teil der Plattformeigenschaften kennen, um sie wirklich gut auszunutzen. Wir werden versuchen, Bezüge zwischen Programmen und Plattformeigenschaften möglichst deutlich werden zu lassen[1]. Weiterhin verfolgt die Beschäftigung mit den Plattformen auch das Ziel, Studierenden eine ganzheitliche Sicht von Informatiksystemen zu vermitteln, auf deren Basis sie Systeme wirklich verstehen, beurteilen und Konsequenzen entwickeln können.

Die ganzheitliche Sicht und die Bezüge zwischen Programmen und Plattformeigenschaften wird grob in der Abbildung 2.1 dargestellt. Diese Abbildung zeigt die verschiedenen Abstraktionsebenen, auf denen man Rechnerhardware betrachten kann.

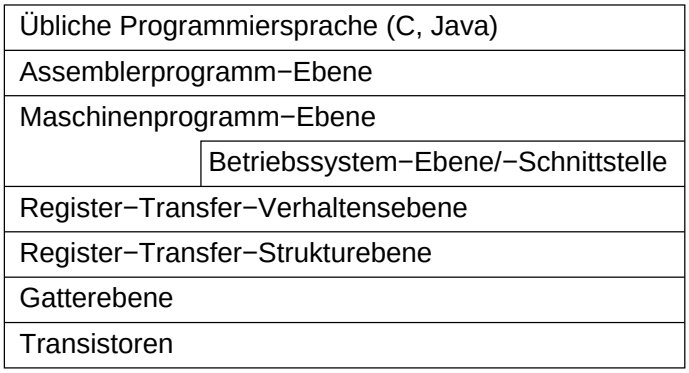


Abbildung 2.1: Abstraktionsebenen

Die Ebene der Transistoren, welche die Basis für die elektronische Datenverarbeitung bilden, betrachten wir im Rahmen dieser Vorlesung nur noch sehr eingeschränkt. Aus Transistoren können wir die Gatter und Speicherelemente konstruieren, mit denen wir uns intensiver beschäftigt haben. Aus Gattern und Speicherelementen können wir komplexere Bausteine wie Addierer, Register, Multiplexer, RAM- und ROM-Bausteine aufbauen. Diese Bausteine bezeichen wir

[1] Eine Ergänzung bietet hier das Buch von Bryant und O'Hallarin [BO03]

als Bausteine der **Register-Transfer (RT)- Struktur**. Dieser Begriff spiegelt wider, dass auf dieser Abstraktionsebene Register eine wesentliche Rolle spielen. Zu Modellen dieser Ebene gehören weiterhin Informationen über die Art und Weise, in der die Inhalte gelesener Register durch Addierer, Multiplizierer usw. transformiert und dann schließlich einem Zielregister zugewiesen werden. Das Wort „Struktur" schließlich drück aus, dass Modelle auf dieser Ebene die Hardware-Komponenten enthalten, welche Transformationen durchführen.

Auf der nächsten Ebene abstrahieren wir von den Komponenten, welche Transformationen durchführen und beschreiben nur noch die Transformation. Wir schreiben beispielsweise „R1 := R2 + R3" ohne uns darum zu kümmern, durch welche Hardwarekomponente die Addition realisiert wird. Wir nennen Beschreibungsmodelle auf dieser Ebene **Register-Transfer (RT)- Verhaltens**-Modelle. Zwischen RT-Verhalten und RT-Struktur wird übrigens in vielen Büchern nicht unterschieden.

Man könnte Rechnerhardware vollständig auf einer der RT-Ebenen modellieren. Dieses wäre aber sehr unübersichtlich und es ist besser, die von einem Rechner ausgeführten Befehle auf einer eigenen Ebene zu modellieren. Dieses ist die Befehlsebene. Die Befehlsebene stellt eine Programmierschnittstelle (engl. *application programming interface (API)*) bereit, die zum Programmieren benutzt werden kann. Maschinenprogramme bestehen aus Folgen von Maschinenbefehlen. In vielen Umgebungen können dabei neben den durch die Maschine direkt realisierten Befehle auch Aufrufe an ein Betriebssystem benutzt werden, die durch nicht hardwaremäßig realisierte Befehle erzeugt werden. Derartige Befehle führen zu einer Verzweigung in das Betriebssystem, das gewisse Dienste wie beispielweise die Ein- und Ausgabe bereitstellen kann. Maschinenprogramme werden in Form von Bitvektoren kodiert, die im Speicher eines speicherprogrammierbaren Rechners abgelegt werden.

Als eigene Ebene wollen wir in diesem Skript Assemblerprogramme betrachten, die in Form von lesbarem Text notiert werden und aus denen mittels eines Assemblers binäre Maschinenprogrogramme erzeugt werden. Die Unterscheidung zwischen Assembler- und (binären) Maschinenprogrammen wird vielfach nicht getroffen. Für uns ist sie wesentlich, da der in den Übungen benutzte MIPS-Assembler mehr Befehle realisert als die MIPS-Hardware.

Die oberste der hier betrachteten Ebenen ist die der „normalen" Programmiersprachen wie C, C++ oder Java[2].

2.2 Die Befehlsschnittstelle

Wir wollen uns einmal vorstellen, wir müssten einen Rechner in seiner Maschinensprache programmieren. Dies kann beispielsweise erforderlich sein, wenn für einen Rechner keine Software vorhanden ist[3] oder wenn wir überlegen müssen, welche Maschinenprogramme ein Compiler erzeugen muss. Wie sehen solche Maschinenprogramme aus? Wir wollen sie zunächst anhand der Maschinensprache der MIPS-Prozessoren kennenlernen. Die Maschinensprache der MIPS-Prozessoren wurde zu Beginn der 90er Jahre entwickelt. MIPS-Prozessoren kann man heute in vielen Geräten wie Kameras oder Druckern finden. Wir benutzen diese Maschinensprache als Beispiel, weil sie verhältnismäßig klar aufgebaut ist, weil wir Programme in dieser Sprache mit einem Simulator simulieren können und weil MIPS-Prozessoren in dem Buch von Hennessy und Patterson [HP08], das wir als Leitfaden für diesen Teil der Vorlesung verwenden, ausführlich erklärt werden.

2.2.1 Der MIPS-Befehlssatz

MIPS-Programme bestehen aus MIPS-Befehlen, welche die vom Prozessor auszuführenden Befehle beschreiben.

2.2.1.1 Arithmetische und Transport-Befehle

Jeder Rechner muss arithmetische Operationen ausführen. Der MIPS-Befehl

```
add a,b,c
```

weist einen Rechner an, `b` und `c` zu addieren und das Ergebnis in `a` zu speichern. Komplexere Ausdrücke sind durch Folgen von Befehlen zu realisieren. So kann beispielsweise die Summe von `b, c, d` und `e` dem Ergebnis `a` durch die Folge

[2] Auf die Unterscheidung zwischen compilierter und interpretierter Ausführung von Java gehen wir hier nicht ein.

[3] Bei manchen kleinen Steuerrechnern, für kleine Roboter –so genannten Mikrocontrollern– ist man nicht weit davon entfernt.

```
add a,b,c
add a,a,d
add a,a,e
```

zugewiesen werden.

Zu beachten ist dabei, dass `a, b, c, d` und `e` nicht Variablen im Sinne von normalen Programmiersprachen sein können. Vielmehr handelt es sich bei allen Argumenten arithmetischer Befehle in der MIPS-Architektur um Register. Die MIPS-Architektur besitzt 32 dem Maschinenprogrammierer zugängliche Register, üblicherweise mit `$0` bis `$31` bezeichnet. Dabei hat das Register `$0` stets den Wert 0 (Bitvektor "000..00").

Befehle wie der `add`-Befehl können mit dem **MARS-Simulator**, der über die Web-Seite http://www.cs.missouristate.edu/MARS/ zu erhalten ist, simuliert werden. Gegenüber einer Ausführung von Assemblerprogrammen auf einer echten Maschine besitzt der Simulator die folgenden Vorteile:

- die Programmierumgebung ist deutlich benutzungsfreundlicher. Programmfehler können die Integrität des Systems, auf dem entwickelt wird, nicht gefährden.
- Es können Zugriffe auf Ein- und Ausgabegeräte simuliert werden, die man in der echten Hardware nicht gestatten wird.
- Alle Studierenden können denselben Befehlssatz simulieren, unabhängig davon, welche Maschine sie besitzen.

Der MARS-Simulator kann unter Window XP, Windows Vista-, Windows 7, MAC OS X- und Linux-Umgebungen genutzt werden[4].

Die Bedeutung (Semantik) des `add`-Befehls können wir mit Hilfe der so genannten **Registertransfernotation** beschreiben. Die Registertransfernotation stellt dar, wie Inhalte zwischen verschiedenen Speicherelementen (wie z.B. Registern) transferiert werden. Die Registertransfernotation beschreibt Zuweisungen von Werten an Register. In dieser Notation können wir Registernamen und Speichernamen als Ziele und Quellen von Informationen verwenden. Adressen notieren wir wie bei Arrays üblich. Beispielsweise können wir die Bedeutung des Assemblerbefehls `add $3,$2,$1` erklären durch:

```
Reg[3] := Reg[2] + Reg[1]
```

`Reg` steht dabei für die als Array modellierten Register. Inhalte von Speichern und Registern stellen für uns Bitvektoren dar. `+` steht für die Addition der einzelnen Bits der Register gemäß der Gleichungen für den Volladdierer. Zur Abgrenzung zwischen Zahlen und Bitvektoren schließen wir in der Registertransfernotation konstante Bitvektoren in Hochkommata ein. Beispiel:

```
"101001010"
```

Als Operatoren verwenden wir die üblichen arithmetischen und logischen Operatoren. Eine Besonderheit ist die Konkatenation (Aneinanderreihung). Für diese benutzen wir das &-Zeichen (dieses Zeichen bedeutet damit keine logische UND-Verknüpfung!). Bei allen außer den Schiebeoperationen setzen wir dieselbe Länge der Bitvektoren voraus. Notfalls muss eine explizite Angleichung der Längen erfolgen. Beispiel:

```
PC := Reg[31] + ("0000000000000000" & adr-teil)
```

Mit Punkten trennen wir die Selektion von Teilbereichen von Bitvektoren ab. Beispiel:

```
Reg[4] := (Hi & Lo).(31:0)
```

Es gehört zu den charakteristischen Eigenschaften der heute üblichen Rechner, dass die Kodierung der Daten, z.B. in Registern, keinerlei Auskunft darüber gibt, wie diese interpretiert werden sollen, also welche der in Teil 1 der Vorlesung vorgestellten Kodierungen genutzt wird. Die Kodierung muss aus dem Kontext heraus bekannt sein. Bei Benutzung des Additionsbefehls `add` wird angenommen, dass die Zahlen entweder im Zweierkomplement oder als Betragszahlen kodiert sind. Welche der beiden Kodierungen benutzt wird, ist weniger wichtig, da die Bildung des Ergebnisses gemäß der Gleichungen für den Volladdierer für die Addition in beiden Fällen das richtige Ergebnis liefert (siehe Teil 1 der Vorlesung).

[4] Windows 8 ist ungetestet.

Dennoch gibt es einen Befehl `addu`, wobei das `u` die Annahme der Kodierung als Betragszahlen (***u**nsigned integers*) nahe zu legen scheint. Tatsächlich unterscheiden sich die Befehle add und addu aber in der Behandlung von Überläufen: beim Überschreiten des darstellbaren Zweierkomplement-Zahlenbereichs erfolgt beim `add`-Befehl eine Ausnahmebehandlung bei Überläufen bei Interpretation der Bitvektoren als ganze Zahlen. Beim `addu`-Befehl werden Überläufe einfach ignoriert. Der `addu`-Befehl ist damit insbesondere zur Realisierung von Additionen in der Programmiersprache C geeignet, da diese ebenfalls für das Ignorieren von Überläufen ausgelegt ist. Eine Ausnahmebehandlung bei Überläufen bei Interpretation der Bitvektoren als natürliche Zahlen wird bei der MIPS-Maschine nicht direkt unterstützt.

Zusätzlich zu den Registern gehört zum Programmiermodell der MIPS-Architektur der Speicher. Der Speicher besteht aus Worten von je 32 Bit, zu denen jeweils ihre Adresse gehört. Das Vorhandensein von Adressen ist das erste Merkmal der so genannten **Von-Neumann-Rechner**, die wir in dieser Vorlesung betrachten. Mit Hilfe dieser Adresse können Worte aus dem Speicher in die Register kopiert werden. Die entsprechenden Befehle nennt man **Datentransportbefehle** (engl. *data transfer instructions*). Der wichtigste ist der Lade-Befehl (*load instruction*) `lw`, der ein Kopieren eines Speicherwortes in ein Register erlaubt. Die Adresse des Speicherwortes kann in der MIPS-Architektur durch die Addition eines Registerinhaltes und einer spätestens beim Laden des Programms bekannten Konstanten gebildet werden.

Beispiel: Der Befehl

```
lw $8, Astart($19)
```

addiert die Konstante Astart (z.B. die Anfangsadresse eines Arrays) zum Inhalt des Registers 19 (z.B. ein Index des Arrays), addressiert mit der Summe den Speicher und lädt das Ergebnis in das Register 8. Etwas präziser können wir die Wirkung des Befehls in der Registertransfernotation ausdrücken:

```
Reg[8] := Speicher [Astart + Reg[19]]
```

wobei wir Register und Speicher als Arrays modellieren.

In der Praxis werden relativ häufig Folgen von Zeichen im Speicher abgelegt, die in jeweils einem Byte kodiert sind. Es ist daher sinnvoll, jedes Byte im Speicher einzeln adressieren zu können. Daher verwenden Maschinen wie die MIPS-Maschine eine **Byteadressierung** und zu jedem Wort im Speicher gehören tatsächlich 4 Adressen. Wir werden später besprechen, wie die Bytes innerhalb eines Wortes nummeriert werden.

Das Pendant zum Lade-Befehl ist der Speicherbefehl `sw`. Die Bildung von Adressen erfolgt wie beim Lade-Befehl. Beispiel (# kennzeichnet einen Kommentar in der Assemblernotation):

```
sw $8, 8($19) # Speicher[8+Reg[19]] := Reg[8]
```

Compiler, die Sprachen wie C oder Java in Maschinensprachen übersetzen, versuchen möglichst viele der Variablen den Registern zuzuordnen (sofern die Variablen nicht mehr als 32 Bit belegen). Nur wenn die Anzahl der Register hierzu nicht ausreicht, dann erfolgt eine Zuordnung zum Speicher. Wenn ein Wert in einem Register aufgrund der begrenzten Anzahl von Registern in den Speicher ausgelagert werden muss, dann spricht man von *spilling*.

Die bislang verwendete Form der Bezeichnung von Maschinenbefehlen ist nicht diejenige, die tatsächlich im Speicher des Rechners benutzt wird. Tatsächlich wird ein (verhältnismäßig einfaches) Programm, ein **Assembler** benötigt, um aus Schreibweisen wie `lw $8, Astart($19)` die Bitvektoren zu erzeugen, mit denen die Befehle im Speicher kodiert werden. Wir müssen also streng genommen zwischen den Assemblerprogrammen und den binären Maschinenprogrammen im Speicher unterscheiden[5].

Die Kodierung von Befehlen basiert auf der Einteilung in **Befehlsfelder**. Die Zergliederung von Befehlsworten in Felder nennt man auch **Befehlsformat** (engl. *instruction format*). Es kann mehrere Formate geben, solange eine eindeutige Dekodierung sichergestellt ist.

Die bislang vorgestellten MIPS-Befehle besitzen die in Tabelle 2.1 dargestellten Maschinenformate [HP08].

Jeweils 6 Bits sind für die **Operationscodes** vorgesehen, welche den jeweils auszuführenden Befehl kodieren. Der Assembler ist dafür verantwortlich, die Bezeichnungen der Befehle wie `add` (*mnemnonics* genannt) in die Operationscodes zu übersetzen. Arithmetische Befehle wie `add` besitzen weiterhin drei Felder zur Angabe der drei Register. Die Felder „shamt" (*shift amount*) und „funct" dienen der Kodierung von Befehlsvarianten. Lade- und Speicherbefehle benötigen nur zwei Registerfelder. Für den Adressteil sind bei diesen Befehlen nur jeweils 16 Bit vorgesehen. Sollen

[5] Tatsächlich gibt es noch weitere Unterscheidungen, wenn man beispielsweise die Form der Speicherung in Dateien mit einbezieht.

	Größe [bit]	6	5	5	5	5	6
Arithmetische Befehle	Format R	op	rs	rt	rd	shamt	funct
LOAD/STORE	Format I	op	rs	rt	adr-teil		

Tabelle 2.1: Befehlsformate der Rechner MIPS R2000-R8000

Adressen benutzt werden, die sich nicht in 16 Bit darstellen lassen, so müssen diese zunächst in ein Register geladen werden und dieses muss dann zur Adressierung benutzt werden. Der MARS-Simulator erzeugt eine solche Sequenz automatisch. Wird also eine Konstante angegeben, die nicht in 16 Bit darstellt werden kann, so wird zunächst ein Register mit Adressinformationen geladen und dieses Register zur Adressierung benutzt. Eingesetzt wird hierzu das Register $1.

Bei den heute üblichen Rechnern werden die kodierten Befehle in einem Speicher untergebracht. Dieses ist das zweite Kennzeichen der **Von-Neumann-Rechner**. Die Adresse des jeweils gerade ausgeführten Befehls ist dabei in einem speziellen Register, dem so genannten **Programmzähler**, enthalten. Eine andere mögliche Bezeichnung für dieses Register wäre *instruction address register*.

Die Benutzung desselben Speichers für Daten und Befehle ist das dritte Kennzeichen der Von-Neumann-Rechner. Ein weiteres Kennzeichen ist es, dass die Befehle in der Regel in der Reihenfolge verarbeitet werden, in der sie im Speicher stehen. Von dieser Reihenfolge kann mittels **Sprungbefehlen** abgewichen werden (siehe unten). Diese Bildung von Programmfolgeadressen gehört zum fünften Kennzeichen der Von-Neumann-Rechner.

Dem Format in Tabelle 2.1 kann man entnehmen, dass der konstante Anteil der Adresse in 16 Bit kodiert werden muss, also nicht zur direkten Adressierung des gesamten Adressraums von 2^{32} Speicherworten ausreicht. Es stellt sich damit auch die Frage, ob die Konstante als vorzeichenlose Betragszahl oder als vorzeichenbehaftete Zweierkomplementzahl zu interpretieren ist. Da die Länge der Bitvektoren im Register und in der Konstanten unterschiedlich ist, reicht es nicht mehr aus, zur Erklärung der Semantik einfach auf die Gleichungen des Volladdierers zu verweisen. Die Darstellung in Registertransfernotation zeigt, dass die Konstante als vorzeichenbehaftete Zweierkomplementzahl interpretiert wird:

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

Der Adressteil wird durch Replizierung des Vorzeichens auf 32 Bit erweitert (siehe auch Abb. 2.2).

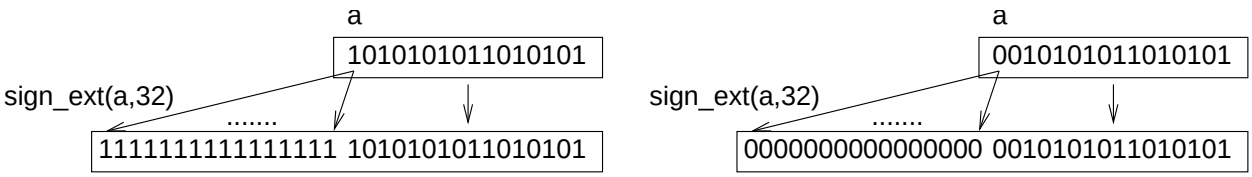


Abbildung 2.2: Vorzeichenerweiterung

Eine solche Replizierung liefert einen Bitvektor, dessen Interpretation als Zweierkomplementzahl wieder denselben Wert liefert wie der ursprüngliche Bitvektor. Diese Aussage belegt der folgende Beweis:

Ein Bitvektor $a = (a_{n-1}, ..., a_0)$ repräsentiert bei einer Kodierung im Zweierkomplement die Zahl

$$int(a) \quad = \quad -a_{n-1}2^{n-1} + \sum_{i=0}^{n-2} a_i 2^i \tag{2.1}$$

In Folgenden sei immer $m > n$ (d.h., es erfolgt tatsächlich eine Erweiterung des Bitvektors). Sei zunächst weiterhin $a_{n-1} =' 0'$, die dargestellte Zahl also positiv. Dann ist

$$int(sign\_ext(a,m)) \quad = \quad int(\text{"00..0"}\&(a_{n-1}, ..., a_0)) = \sum_{i=0}^{n-2} a_i 2^i = int(a) \tag{2.2}$$

Dabei enthält der konstante Bitvektor insgesamt $m - n + 1$ Nullen. Der vorzeichenerweiterte Bitvektor stellt also dieselbe Zahl wie der ursprüngliche Bitvektor dar.

Sei jetzt $a_{n-1} =' 1'$, die dargestellte Zahl also negativ.

$$int(sign\_ext(a,m)) \quad = \quad int("11..1"\&(a_{n-1},...,a_0)) = -2^{m-1} + \sum_{i=n-1}^{m-2} 2^i + \sum_{i=0}^{n-2} a_i 2^i \tag{2.3}$$

Dabei enthält der konstante Bitvektor insgesamt $m-n+1$ Einsen. Es gilt:

$$\sum_{i=n-1}^{m-2} 2^i + 2^{n-1} \quad = \quad 2^{m-1} \tag{2.4}$$

also auch

$$-2^{m-1} + \sum_{i=n-1}^{m-2} \quad = \quad -2^{n-1} \tag{2.5}$$

Damit folgt aus Gleichung 2.3

$$int(sign\_ext(a,m)) \quad = \quad -2^{n-1} + \sum_{i=0}^{n-2} a_i 2^i = int(a) \tag{2.6}$$

Der vorzeichenerweiterte Bitvektor stellt also auch bei negativen Zahlen dieselbe Zahl wie der ursprüngliche Bitvektor dar. q.e.d.

Vielfach zeigt das Register `Reg[rs]` in `sw`- und `lw`-Befehlen wie oben auf einen gewissen Speicherbereich, wie z.B. dem einer Prozedur zugeordneten Bereich. Die Interpretation des konstanten Teils der Adresse als Zweierkomplementzahl bewirkt, dass man über `sw`- und `lw`-Befehle sowohl unterhalb wie auch oberhalb der im Register enthaltenen Adresse auf den Speicherbereich zugreifen kann. Würde man den konstanten Anteil der Adresse mit Nullen auffüllen, so könnte man lediglich oberhalb der im Register enthaltenen Adresse auf den Speicher zugreifen[6].

Neben dem Additionsbefehl `add` gibt es natürlich noch viele weitere arithmetische Befehle im MIPS-Befehlssatz. Zunächst einmal können kleine Konstanten auch direkt addiert werden. Hierzu gibt es den *add immediate*-Befehl `addi`, der das Format I der Tabelle 2.1 benutzt und die Bedeutung

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

besitzt. Es ist eine Besonderheit von MIPS-Assemblern, die Kodierung für einen solchen Immediate-Befehl auch dann zu erzeugen, wenn man bei einem normalen `add`-Befehl direkt eine Konstante angibt.

```
add $4,$5,3
```

hat also dieselbe Bedeutung wie

```
addi $4,$5,3.
```

Dasselbe gilt für die anderen Arithmetik-Befehle.

Zusätzlich zum `addi`-Befehl gibt es noch den `addiu`-Befehl. Dieser Befehl addiert die Konstante **vorzeichenerweitert** zum Registerinhalt, was man von einer *unsigned*-Operation nicht erwarten würde. Der Sinn liegt in der Konsistenz mit dem `addu`-Befehl.

Neben der Addition gibt es natürlich noch die Subtraktionsbefehle mit den Varianten `sub` (mit Ausnahmeerzeugung)und `subu` (ohne Ausnahmeerzeugung). Beispiele:

[6] Skizze siehe Folien!

```
sub $4,$3,$2 # Reg[4]:= Reg[3]-Reg[2]
```

```
subu $4,$3,$2 # Reg[4]:= Reg[3]-Reg[2]
```

Es gibt keinen Befehl für die Subtraktion von immediate-Operanden, da man dafür den `addi`-Befehl mit einer negativen Konstanten benutzen kann.

Bei der Multiplikation und der Division gibt es die Besonderheit, dass zur Darstellung des Ergebnisses 64 Bit benötigt werden. Aus diesem Grund werden die Ergebnisse in der MIPS-Architektur in zwei Spezialregistern `Hi` und `Lo` abgelegt. Beispiel:

```
mult $2,$3 # Hi & Lo := Reg[2] * Reg[3]
```

Das `Hi`-Register enthält den höherwertigen Teil des Ergebnisses, das `Lo`-Register den niederwertigen Teil.

Aus den `Hi`- und `Lo`-Registern müssen die Ergebnisse mit Transportbefehlen `mfhi` und `mflo` kopiert werden. Beispiel:

```
mfhi, $3 # Reg[3]:=Hi
mflo, $3 # Reg[3]:=Lo
```

Der Assembler stellt über **Pseudobefehle** `mul, mulo` und `mulou` Befehlssequenzen zusammen, welche dieses Kopieren bereits enthalten. `mul` enthält dabei wie Additionsbefehle drei Register und kann damit analog zu den Befehlen benutzt werden[7]. `mulo` hat dieselbe Wirkung wie `mul`, generiert aber eine Ausnahme, wenn das Ergebnis nicht im `Lo`-Register dargestellt werden kann. `mulou` hat dieselbe Wirkung für vorzeichenlose Zahlen. Bei der Multiplikation werden die Zusätze `u` und `o` wirklich eingesetzt, um *unsigned*-Operationen und *overflow*-Tests zu kennzeichnen. Einen im Prinzip sinnvollen Befehl `mulu` gibt es nicht.

Da übliche Divisionsalgorithmen sowohl den Quotienten wie auch den Rest bestimmen, nutzt man die `Hi`- und `Lo`-Register für die Speicherung des Ergebnisses dieser Algorithmen. Es gibt Befehle für vorzeichenlose und für vorzeichenbehaftete Zahlen. Beispiele:

Für ganze Zahlen in Zweierkomplement-Darstellung:

```
div $2,$3 # Lo:=Reg[2]/Reg[3]; Hi:=Reg[2] mod Reg[3]
```

Für vorzeichenlose Zahlen:

divu $2,$3 # Lo:=Reg[2]/$_u$ Reg[3]; Hi:=Reg[2] mod$_u$ Reg[3]

Dabei kennzeichnet ein Index `u` die Division für eine vorzeichenlose Interpretation der Bitvektoren.

Eine Übersicht über die bislang erklärten Befehle liefern die Tabellen 2.2 und 2.3.

Beispiel	Bedeutung	Kommentar
lw $3,100($2)	Reg[3] := Speicher[Reg[2]+100]	load word
sw $3,100($2)	Speicher[Reg[2]+100] := Reg[3]	store word
lui $3,100	Reg[3] := 100 << 16	Lade oberes Halbwort (<< = schiebe nach links)
mfhi $3	Reg[3] := Hi	Lade aus dem Multiplizierregister Hi
mflo $3	Reg[3] := Lo	Lade aus dem Multiplizierregister Lo

Tabelle 2.2: Datentransferbefehle (zu `lui` siehe Seite 12)

Logische Befehle arbeiten bitweise auf den beteiligten Registern. Es gibt logische Befehle für die UND- und für die ODER-Verknüpfung. Dabei sind auch Direktoperanden möglich. Diese werden sinnvollerweise mit Nullen an die Wortbreite der Register angepasst. Zusätzlich gibt es Schiebebefehle, welche die Bitvektoren um eine angebbare Zahl von Positionen nach links oder rechts schieben. Unterstützt werden v.a. logische Schiebebefehle, bei denen in die Register Nullen hineingeschoben werden. Zusätzlich gibt es einen *shift right arithmetical* Befehl, bei dem das Vorzeichen immer erhalten bleibt. Die Befehle sind in Tabelle 2.4 aufgeführt.

Zur Addition größerer Konstanten kann zunächst mit dem Befehl `lui` (*load upper immediate*) eine Konstante in einem oberen Halbwort eines Hilfsregisters erzeugt werden. Durch eine Kombination des lui und des ori-Befehls kann man

[7] Merkregel: Weniger ist mehr: der Assemblerbefehl mit drei Buchstaben besorgt gleich das Kopieren des Ergebnisses in die üblichen Register; der Befehl mit vier Buchstaben lässt das Ergebnis in den Spezialregistern.

Beispiel	Bedeutung	Kommentar
add $4,$2,$3	Reg[4] := Reg[2] + Reg[3]	Addition, Ausnahmen (2k-Überlauf) möglich
sub $4,$2,$3	Reg[4] := Reg[2] - Reg[3]	Subtraktion, Ausnahmen (2k-Überlauf) möglich
addi $4,$2,100	Reg[4] := Reg[2] + 100	Ausnahmen (2k-Überlauf) möglich
addu $4,$2,$3	Reg[4] := Reg[2] + Reg[3]	Addition natürl. Zahlen, keine Ausnahmen
subu $4,$2,$3	Reg[4] := Reg[2] - Reg[3]	Subtraktion natürl. Zahlen, keine Ausnahmen
addiu $4,$2,100	Reg[4] := 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] $*_u$ Reg[3]	Multiplikation m. 64-Bit Ergebnis, natürl. Z.
mul $4,$3,$2	Reg[4] := (Reg[3] * Reg[2])(31:0)	wie `mult` & Kopieren nach Reg[4]
mulo $4,$3,$2	Reg[4] := Reg[3] * Reg[2](31:0)	wie `mult` & Kopieren nach Reg[4]], mit Überlauf
mulou $4,$3,$2	Regs[4]:= Reg[3] $*_u$ Reg[2](31:0)	wie `multu` & Kopieren nach Reg[4], mit Überlauf
div $2,$3	Lo:= Reg[2] / Reg[3], Hi := Reg[2] mod Reg[3]; ganze Zahlen	
divu $2,$3	Lo:= Reg[2] $/_u$ Reg[3], Hi := Reg[2] mod$/_u$ Reg[3]; natürl. Zahlen	

Tabelle 2.3: Arithmetische Befehle

Beispiel	Bedeutung	Kommentar
and $4,$3,$2	Reg[4] := Reg[3] $\wedge$ Reg[2]	und
or $4,$3,$2	Reg[4] := Reg[3] $\vee$ Reg[2]	oder
xor $4,$3,$2	Reg[4] := Reg[3] xor Reg[2]	Exklusiv-oder
andi $4,$3,100	Reg[4] := Reg[3] $\wedge$ 100	und mit Konstanten
ori $4,$3,100	Reg[4] := Reg[3] $\vee$ 100	oder mit Konstanten
xori $4,$3,100	Reg[4] := Reg[3] xor 100	Exklusiv-oder mit Konstanten
sll $4,$3,10	Reg[4] := Reg[3] $<<$ 10	schiebe nach links logisch
srl $4,$3,10	Reg[4] := Reg[3] $>>$ 10	schiebe nach rechts logisch
sra $4,$3,10	Reg[4] := Reg[3] $>>_a$ 10	schiebe nach rechts arithmetisch

Tabelle 2.4: Logische Befehle

beliebige Konstanten in ein 32-Bit-Register laden. Diese Funktion steht über den `li` (*load immediate*) - Pseudobefehl in anwendungsfreundlicher Form zur Verfügung. Der Befehl

```
li $2, konstante
```

kann beispielsweise realisiert werden durch[8]

```
lui $1, konstante div 2^16
ori $2, $1, konstante mod 2^16.
```

MIPS-Assembler erzeugen derartige `li`-Befehle selbst, wenn sonst kein Code erzeugt werden könnte, also etwa wenn die Adresskonstante eines `lw`-Befehls nicht in das dafür vorgesehene Befehlsfeld passt. Er verwendet dafür immer das Register $1 als Hilfsregister. Damit dabei keine durch den Benutzer erzeugten Informationen überschrieben werden, ist das Register $1 immer für den Assembler freizuhalten.

Adressen von Variablen kann man mit dem Pseudobefehl `la` (*load address*) laden. Er entspricht in der Wirkung dem `lw`-Befehl, wobei allerdings der Speicherzugriff unterbleibt.

Beispiel:

```
la $2 0x20($3) # Reg[2] := 0x20 + Reg[3]
```

Der Befehl ist zur Verbesserung der Lesbarkeit sinnvoll einzusetzen, wenn beispielsweise an eine Prozedur die Adresse eines Parameters zu übergeben ist. Der Befehl kann über eine Folge aus einem `li`- und einem `add`-Befehl realisiert werden.

[8] Die Realisierung im MARS-Simulator unterscheidet sich von der im SPIM-Simulator, wobei beide korrekt sind.

2.2.1.2 Der MARS-Simulator

2.2.1.3 Einteilung in Speicherbereiche

Der MARS-Simulator unterstützt eine Einteilung des Speichers in Bereiche, wie sie auch von vielen Programmiersprachen, insbesondere bei C/Unix so oder ähnlich vorgenommen wird. Es ist dies eine Einteilung in:

- Einen reservierten Bereich, der für uns nicht weiter von Bedeutung ist.
- Das **Text-Segment** (engl. ***text segment***): Dies ist ein traditionell so genannter Bereich, in dem nicht wirklich Texte, sondern die Befehle abgelegt werden. Dieser Bereich des Speichers wird vielfach vor dem dynamischen Überschreiben geschützt, sofern für einen solchen Speicherschutz Hardware vorhanden ist. Dadurch kann eine versehentliche Modifikation des Codes verhindert werden.
- Das **Datensegment** (engl. ***data segment***): In diesem Bereich hinterlegen Programme dauerhaft Daten. Dieser Bereich beginnt zunächst mit den Daten bekannter Größe. Es wird üblicherweise erlaubt, dass sich dieser Bereich während der Laufzeit ausdehnt, beispielsweise aufgrund von expliziten Speicheranforderungen.
- Das ***stack segment***: In diesem werden ausschließlich zur Laufzeit des Programms Informationen abgelegt. Auf dieses Segment verweist der *stack pointer*. Dieser Bereich wird üblicherweise oberhalb des Datensegments angeordnet. Er wächst dem Datensegment entgegen. Leider ist in vielen Systemen kein Schutz vor einer Überlappung zwischen *stack segment* und Datensegment vorhanden. Das Fehlen eines solchen Schutzes kann dann für vielerlei Angriffe gegen ein System ausgenutzt werden.
- Ein Bereich von **Adressen für Ein/Ausgabegeräte**.
- Adressen, die für ein **Betriebssystem** (auch *kernel* genannt) reserviert sind. Eine übliche Konvention ist, diesen Bereich durch das höchstwertige Adressbit zu kennzeichnen. Dieser Konvention folgt auch der MARS-Simulator. Sofern die Hardware entsprechende Schutzmechanismen erlaubt, wird ein direkter Zugriff auf Adressen in diesem Bereich durch ein Benutzerprogramm verboten. Damit wird verhindert, dass beliebiger Code des Betriebssystems von einem Benutzerprogramm ausgeführt werden kann. Der Zugang zum Betriebssystem muss dann über spezielle **Systemaufrufe** erfolgen. Leider unterstützt der MARS-Simulator selbst keinerlei Speicherschutz.

Abb. 2.3 zeigt diese Aufteilung des Speichers (engl. *memory map*).

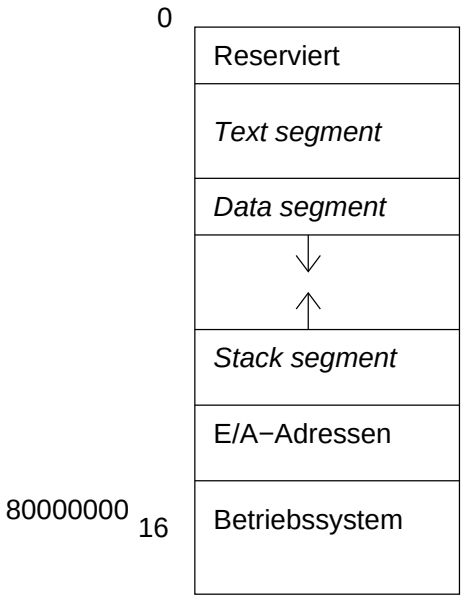


Abbildung 2.3: Aufteilung der Speicheradressen in verschiedene Bereiche

Wenn der physikalische Speicher tatsächlich direkt mit den bislang benutzten Adressen angesprochen wird, dann ergibt sich aus dieser Benutzung der Speicherbereiche ein Nachteil: Für die Bereiche gemäß Abb. 2.3 (mit Ausnahme der

Adressen für Ein/Ausgabegeräte) muss dann auch physikalischer Speicher vorhanden sein. Es ist relativ aufwändig, in verschiedenen Teilen des Adressbereichs physikalischen Speicher vorzusehen. Für dieses Problem gibt es im Wesentlichen zwei Lösungen:

1. Mittels spezieller Hardware werden die Adressen so umgerechnet, dass die neuen Adressen weitgehend lückenfreie Wertebereich belegen. Techniken dazu werden wir in den Abschnitten 2.4.2.2 und 2.4.2.3 kennen lernen.
2. Man verwendet von vornherein weitgehend zusammenhängende Adressbereiche. Dazu nutzt man aus, dass der Speicherbedarf des Betriebssystems bei einfachen Systemen beim Start überwiegend bekannt ist. Man kann dann Lücken zwischen den Speicherbereichen des Betriebssystems und des Benutzerprogramms vermeiden (siehe Abb. 2.4).

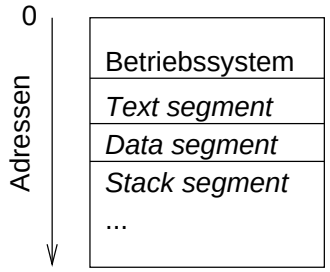


Abbildung 2.4: Aufteilung der Speicheradressen in Systemen ohne spezielle Adressumrechnunghardware

Man kommt so ohne die unter Punkt 1. angesprochene Hardware aus. Sie ist allerdings weniger flexibel als die Methode unter 1., da die Größe der meisten Speicherbereiche bekannt sein muss. Die Speicheraufteilung des MARS-Simulators kann über die Konfigurationseinstellungen entsprechend einer solchen Aufteilung verändert werden. Diese Methode ist Ausgangspunkt einiger Überlegungen im dem Betriebssystemsbuch von Tanenbaum, das im 2. Semester benutzt wird.

Es ist relativ praktisch, die Anfangsadresse des Datenbereichs jeweils in einem Register zu halten:

```
main: li  $4, 0x10010000    #Anfang Datenbereich
      lw  $2, 0($0)
      lw  $3, 4($4)
      add $3, $2, $3
      sw  $3, 8($4)
```

Auf diese Weise vermeidet man, dass man für jeden Zugriff auf den Datenspeicher zunächst mittels explizitem oder durch den Assembler implizit erzeugtem `lui`-Befehl einen Teil der Adresse laden muss.

2.2.1.4 Funktion des Assemblers

Der mit dem MARS-Simulator gelieferte Assembler besitzt die folgenden Aufgaben:

- Er übersetzt mnemotechnische Befehlsbezeichner (engl. *mnemnonics*) in Operationscodes.
- Der MARS-Assembler realisiert Pseudobefehle, die auf der echten Maschine nicht als eigenständige Codes existieren. Dazu gehören beispielsweise die Übersetzung von *load immediate*-Befehlen in *or immediate*-Befehle unter Verwendung der Register $0 und $1:

  `li $1,15` → `ori $1,$0,15`

  sowie die Übersetzung von Pseudobefehlen wie `blt` (*branch if less than*) in Folgen von Befehlen.
- Die Übersetzung von symbolischen Marken (*label*) in Bitvektoren.

- Die Übersetzung von symbolischen Bezeichnern für Register in Bitvektoren, die in den Registerfeldern der Befehle gespeichert werden. Dabei gibt es u.a. die folgenden Entsprechungen:

Symbolisch	Register	Verwendung
`$zero`	`$0`	=0
`$at`	`$1`	Assembler
`$v0`	`$2`	Ergebnisregister für Funktionen$(s.u.)$
`$v1`	`$3`	Ergebnisregister für Funktionen
`$a0`	`$4`	Argumentregister für Funktionen
`$a1`	`$5`	Argumentregister für Funktionen
`$a2`	`$6`	Argumentregister für Funktionen
`$a3`	`$7`	Argumentregister für Funktionen
`$t0-t7`	`$8-$15`	Hilfsregister, vor Funktionsaufruf nicht gesichert.
`$s0-s7`	`$16-$23`	Hilfsregister, vor Funktionsaufruf gesichert.
`$t8-t9`	`$24-$25`	Hilfsregister, vor Funktionsaufruf nicht gesichert.
`$k0-k1`	`$26-$27`	Für das Betriebssystem reserviert. (s.u.)
`$gp`	`$28`	Zeiger auf globalen Datenbereich.
`$sp`	`$29`	Stackpointer (s.u.)
`$fp`	`$30`	So genannter Framepointer
`$ra`	`$31`	Rücksprungadresse für Funktionen.

- Die Unterteilung der zu ladenden Speicherworte in Code- und Datenbereiche.
- Die Verarbeitung von Anweisungen an den Assembler. Dazu gehören Anweisungen, den Speicher mit Daten zu initialisieren oder Speicher für Daten freizuhalten.

  Beispiele:

  `.ascii text` # Text wird im Datensegment abgelegt

  `.asciiz text` # Text wird im Datensegment abgelegt, 0 am Ende

  `.data` # die nächsten Worte sollen im Daten-Bereich abgelegt werden,

  `.extern` # Bezug auf externes globales Symbol

  `.globl id` # Bezeichner id soll global sichtbar sein (von uns nicht benötigt),

  `.kdata id` # die nächsten Worte kommen in den Daten-Bereich des Betriebssystems (des *kernels*),

  `.ktext id` # die nächsten Worte kommen in den Text-Bereich des Betriebssystems (des *kernels*),

  `.set` # SPIM-Befehl, von MARS ignoriert

  `.space n` # reserviere n Bytes im Daten-Segment,

  `.text` # die nächsten Worte kommen in den Text-Bereich,

  `.word wert, ... wert` # Werte, die im aktuellen Bereich abgelegt werden,

  Eine vollständige Liste der Assembleranweisungen des mit dem MARS-Simulator ausgelieferten Assemblers enthält die Online-Dokumentation des MARS-Simulators.

2.2.1.5 Sprungbefehle

Für die Funktionalität eines Rechners ist es zentral, dass er Entscheidungen treffen kann. Im MIPS-Befehlssatz gibt es zwei Befehle, mit denen Entscheidungen getroffen werden können:

- `beq register, register, Sprungziel` (*branch if equal*): Wenn die beiden Register dieselben Inhalte haben, dann erfolgt ein Sprung an das `Sprungziel`.

- `bne register, register, Sprungziel` (*branch if not equal*): Wenn die beiden Register unterschiedliche Inhalte haben, dann erfolgt ein Sprung an das `Sprungziel`.

Diese beiden Befehle heißen **bedingte Sprünge** (engl. *conditional branches*).

Wir können diese Sprünge nutzen, um das nachfolgende kleine C-Programm

```
   if (i==j) goto L1;
   f = g + h;
L1:f = f - i;
```

in ein Maschinenprogramm zu übersetzen:

```
   beq $19,$20,L1      # springe an L1 falls i=j
   add $16,$17,$18     # f = g + h (ggf. übersprungen)
L1:sub $16,$16,$19     # immer ausgeführt
```

Dabei haben wir angenommen, dass die Variablen den Registern `$16` bis `$20` zugeordnet werden können.

Bei speicherprogrammierbaren Rechnern werden Befehle im Speicher abgelegt und besitzen daher ebenso wie Daten Adressen. Die Marke `L1` korrespondiert zu einer Befehlsadresse. Derartige Marken sind nur in Assemblerprogrammen verfügbar. Der Assembler nimmt dem Programmierer die Arbeit ab, diese Marken in binär codierte Werte umzurechnen.

Sprünge an Marken müssen auch erzeugt werden, wenn im C-Quellprogramm keine Marke auftaucht, wie das nachfolgende Beispiel zeigt. Aus

```
   if (i==j) f = g + h; else f = g - h;
```

wird

```
      bne $19,$20,Else      # goto ELse, wenn nicht i=j
      add $16,$17,$18       # übersprungen, wenn nicht i=j
      j   Exit              #
Else: sub $16,$17,$18       # übersprungen, wenn i=j
Exit: ...
```

In diesem Fall ist ein **unbedingter Sprung** (*unconditional jump*) `j Exit` erforderlich.

Sprungbefehle werden auch benötigt, um Schleifen zu realisieren. Als Beispiel betrachten wir die folgende Schleife:

```
   while (save[i]==k) i = i + j;
```

Unter der Annahme, dass die Variablen `i, j` und `k` den Registern `$19, $20` und `$21` zugeordnet werden, dass das Array `save` an der Adresse `Sstart` beginnt und dass das Register `$10` eine 4 enthält ergibt sich folgendes Maschinenprogramm:

```
Loop: mult $9,$19,$10      # Hilfsregister $9 = i*4
      lw   $8,Sstart($9)   # Hilfsregister $8 = save[i]
      bne  $8,$21, Exit    # Nach Exit falls ungleich
      add  $19,$19,$20     # i = i + j
      j    Loop
Exit: ...
```

	Größe [bit]	6	5	5	5	5	6
Arithmetische Befehle	Format R	op	rs	rt	rd	shamt	funct
`lw, sw`	Format I	op	rs	rt	adr-teil		
`j, jal`	Format J	op	adr-teil				

Tabelle 2.5: Befehlsformate der Rechner MIPS R2000-R8000 einschl. Format J

Unbedingte Sprünge erfordern ein eigenes Befehlsformat gemäß der ergänzten Übersicht in Tabelle 2.5.

Damit stehen für den `j`- und den später erklärten `jal`-Befehl nur 26 Bit zur Angabe eines Sprungziels zur Verfügung. Zwei weitere Bits können wir gewinnen, indem wir ausnutzen, dass Befehle immer auf Wortgrenzen ausgerichtet sein müssen. Damit müssen die letzten beiden Adressbits immer =’0’ sein und müssen nicht im Sprungbefehl gespeichert werden. Zur Bildung einer 32 Bit langen Zieladresse werden jetzt die signifikantesten 4 Bits aus dem bisherigen Wert des Programmzählers übernommen. Damit wird die Folgeadresse bei Sprüngen durch Konkatenation wie folgt gebildet:

```
PC := PC.(31:28) & adr-teil & "00"
```

Mit 28 Bit kann man 256 MB erreichen. Mit dem unbedingten Sprungbefehl kann man damit immer nur innerhalb des aktuellen Blocks von 256 MB springen, da ja die signifikantesten Bits aus dem alten Wert des Programmzählers übernommen werden. Diese Einschränkung entsteht durch den Wunsch, die Befehlswortlänge konstant zu halten und zur Darstellung von Sprungbefehlen nicht mehrere Worte zu verwenden. Dieser Wunsch ist typisch für so genannte **reduced instructions set computers** (RISCs). Die MIPS-Maschine gehört zu dieser Klasse von Maschinen. Will man den gegenwärtigen 256 MB-Block verlassen, so müssen andere Sprungbefehle eingesetzt werden. Wir werden später sehen, dass man dazu zunächst das Sprungziel in ein Register laden und dann springen muss.

Neben dem Test auf Gleichheit oder Ungleichheit wird vielfach auch ein Test „kleiner”, „kleiner gleich”, „größer” oder „größer gleich” benötigt. Es ist eine Besonderheit des MIPS-Befehlssatzes, dass diese Tests aus jeweils zwei Maschinenbefehlen zusammengesetzt werden müssen. Der erste Befehl ist jeweils ein *set on less than*-Befehl `slt`. Der Befehl `slt $8, $19, $20` beispielsweise setzt das Register 8 auf 1 wenn der Wert in Register 19 kleiner ist als der Wert in Register 20, sonst wird das Register 8 auf 0 gesetzt. Eine Kombination von `slt` mit den Befehlen `beq` und `bne` zusammen mit dem festen Wert 0 im Register 0 liefert alle notwendigen bedingten Verzweigungen. Solche Kombinationen werden bereits durch den MIPS-Assembler realisiert. Er übersetzt beispielsweise den **Pseudobefehl** `blt` (*branch on less than*) in eine Abfolge von `slt` und `bne`-Befehlen. Aus

```
blt $2,$3,L
```

wird

```
slt $1,$2,$3
bne $1,$0,L
```

Analog dazu erfolgt die Übersetzung von Befehlen wie `bge` (*branch if greater or equal*), `bgt` (*branch if greater than*) und `ble` (*branch if if less than or equal*). Zur Unterstützung derartiger Pseudobefehle wird wieder ausgenutzt, dass MIPS-Programme nie das Register 1 benutzen dürfen und dass dieses daher uneingeschränkt dem Assembler zur Verfügung steht.

Als Variation des `slt`-Befehls gibt es den `slti`-Befehl, bei dem der zweite Operand eine Konstante ist. Die `slt`- und `slti`-Befehle interpretieren die Bitvektoren dabei als Zweierkomplementzahlen. Zur Interpretation als natürliche Zahlen (*unsigned integers*) gibt es noch die beiden Befehle `sltui` und `sltu`.

Als nächstes wollen wir uns die Übersetzung einer `for`-Schleife ansehen. Die Schleife sei wie folgt gegeben:

```
j = 0;
for (i=0; i<n; i++)  j = j+i;
```

Dieses Programm kann in das folgende Assemblerprogramm übersetzt werden (die Variablen `j,i,n` seien dabei den Registern 2,3 und 4 zugeordnet):

```
main:  li   $2,0          # j:=0
```

```
       li   $3,0          # i:=0
loop:  bge  $3,$4, ende   # i>=n?
       add  $2,$2,$3      # j=j+i
       addi $3,$3,1       # i++
       j  loop
ende   ...
```

Die meisten Programmiersprachen erlauben auch berechnete Sprünge. Solche berechneten Sprünge erlauben es, statt mit Ketten von geschachtelten *if-statements* mit Tabellen von Sprungadressen zu arbeiten, was wesentlich effizienter ist. Zu diesem Zweck besitzt die MIPS-Architektur einen *jump register*-Befehl `jr`. Er bewirkt einen Sprung an die im Register vorhandene Adresse.

In C werden berechnete Sprünge über ein so genanntes *switch statement* ausgedrückt. Ein Beispiel dafür ist der folgende C-Code:

```
switch(k) {
      case 0: f = i + j; break;    /* k = 0 */
      case 1: f = g + h; break;    /* k = 1 */
      case 2: f = g - h; break;    /* k = 2 */
      case 3: f = i - j; break;    /* k = 2 */
           }
```

Wir nehmen an, dass Register $10 eine 4 enthält, dass `k` dem Register $21 zugeordnet ist und dass an Adresse `Jumptable` eine Tabelle beginnt, welche die Adressen zu den Marken `L0, L1, L2` und `L3` enthält:

```
jumptable: .word L0, L1, L2, L3
```

Dann kann das Programm in folgendes Assemblerprogramm übersetzt werden:

```
Loop: mul  $9, $10, $21       # Hilfsregister $9 = k*4
      lw   $8, Jumptable($9)  # Hilfsregister $8 = Jumptable[k]
      jr   $8                 # Springe an die Adresse in Register 8
L0:   add  $16, $19, $20      # k==0; f = i + j
      j    Exit               # entspricht break
L1:   add  $16, $17, $18      # k==1; f = g + h
      j    Exit
L2:   sub  $16, $17, $18      # k==2; f = g - h
      j    Exit
L3:   sub  $16, $19, $20
Exit: ...
```

Die Tabelle 2.6 enthält eine vollständige Übersicht über alle Sprungbefehle der MIPS-Architektur.

Die Befehle `bne` und `beq` benutzen das Format I der Tabelle 2.5. Die Befehle `slt` und `jr` verwenden das Format R derselben Tabelle. Der Befehl `j` benötigt das Format J.

Der auf einen Sprung statisch folgende Befehl wird stets noch ausgeführt (*delayed jump*), dies wird aber durch den Assembler ggf. durch Einführen eines *no operation*-Befehls NOP (eines Befehls ohne Wirkung) versteckt.

2.2.1.6 Realisierung von Prozeduren

Mit Hilfe von Prozeduren können Programme übersichtlich gestaltet werden und es ist möglich, Programmteile wiederzuverwenden. Welche Befehle können zur Realisierung von Prozeduren eingesetzt werden? Zunächst einmal muss es Befehle geben, um zu Prozeduren verzweigen zu können sowie Befehle, mit denen man an den auf den Sprung folgenden Befehl zurückkehren kann. Weiter muss es Möglichkeiten geben, Parameter zu übergeben und Ergebnisse zurückzureichen.

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* (siehe 2.2.1.6)

Tabelle 2.6: Sprünge und Tests

Der Sprung an eine Prozedur kann über den *jump-and-link*-Befehl des MIPS-Befehlssatzes realisiert werden. Der Befehl besitzt die Form

```
jal ProzedurAdresse
```

Er rettet die Adresse des auf den Sprung folgenden Befehls in das Register 31 und bewirkt einen Sprung an die angegebene Adresse. `link` steht hier für die Eigenschaft, die Rückkehradresse zu speichern. Der Rücksprung kann dann sehr einfach mit dem `jr`-Befehl erfolgen:

```
jr $31
```

führt zum Rücksprung.

Der `jal`-Befehl muss auf ein Register zugreifen, welches die Adresse des aktuellen Befehls oder des Folgebefehls enthält. Diese Funktion übernimmt der **Programmzähler** PC.

Nehmen wir jetzt einmal an, dass eine Prozedur A eine weitere Prozedur B aufruft. In diesem Fall muss die Prozedur A den Inhalt des Register 31 retten, denn sonst ginge beim Aufruf von B mit `jal` die Rückkehradresse für A verloren. Prozeduren können immer weitere Prozeduren aufrufen und die Rückkehradressen werden in der zu den Aufrufen umgekehrten Reihenfolge benötigt. Die ideale Datenstruktur zum Abspeichern der Rückkehradressen ist der **Keller** (engl. *stack*), auch **Stapel**[9] oder LIFO-Schlange (*last-in, first-out*) genannt. Ein solcher Stapel kann im Speicher realisiert werden. Man muss aber stets wissen, an welcher Adresse der nächste freie Platz im Speicher beginnt oder (äquivalent) was die jeweils letzte durch den Keller belegte Speicheradresse ist. Diese Adresse, den so genannten ***stack pointer*** kann man wieder in einem Register halten (siehe Abb. 2.5). Über den Inhalt dieses Registers „wissen" Prozeduren, wohin Rückkehradressen zu retten sind.

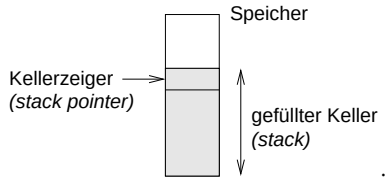


Abbildung 2.5: Realisierung eines Stapels im Speicher

Die Operation, mit der man Informationen auf dem Stapel ablegt und dabei den Zeiger auf den nächsten freien Speicherplatz anpasst, nennt man *push*-Operation. Die Operation, mit der man Informationen vom Stapel holt und dabei den Zeiger auf den nächsten freien Speicherplatz ebenfalls anpasst, nennt man *pop*-Operation.

Als Beispiel betrachten wir eine Prozedur A, welche eine Prozedur B ruft und diese wiederum ruft C. Die Abbildungen 2.6 bis 2.9 zeigen die Operationen, die ausgeführt werden müssen.

Die gezeigten geschachtelten Prozeduraufrufe lassen sich mit dem folgenden Maschinenprogramm realisieren. Dabei wird angenommen, dass sich die Anzahl der auf dem Stapel benötigten Zellen im Register 24 befindet:

[9] Der Begriff Stapel trifft die Eigenschaften viel besser als der des Kellers, obwohl im deutschen Sprachraum zunächst der Begriff des Kellers eingeführt wurde. Im englischen Sprachraum würde man mit der Übersetzung *cellar* erhebliche Verwirrung stiften.

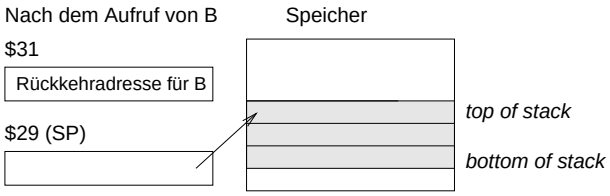


Abbildung 2.6: Speicherinhalte nach dem Aufruf von B

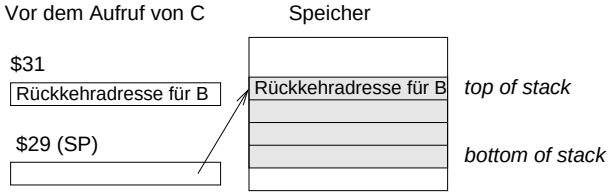


Abbildung 2.7: Speicherinhalte vor dem Aufruf von C

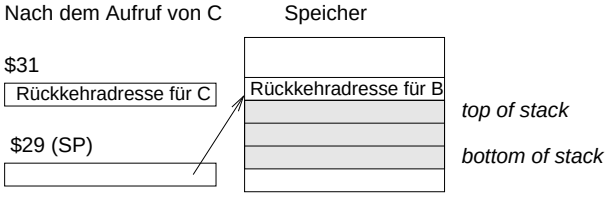


Abbildung 2.8: Speicherinhalte nach dem Aufruf von C

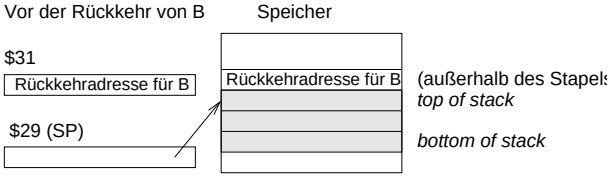


Abbildung 2.9: Speicherinhalte vor der Rückkehr von B

```
A: ...
   jal B              # Rette Rücksprungadresse in $31 und rufe B
   ...

B: ...
   sub $29,$29,$24    # korrigiere Stapelzeiger
   sw  $31, 0($29)    # speichere Rückkehradresse
   jal C              # Rette Rücksprungadresse in $31 und rufe C
   lw  $31,0($29)     # restauriere Rückkehradresse von B
   add $29,$29,$24    # setze Stapelzeiger zurück
   ...
   jr  $31            # kehre zur Prozedur zurück, die B rief

C: ...
   ...
   jr  $31
```

Dabei haben wir bereits eine Optimierung vorgenommen: wir sparen uns das Retten der Rückkehradresse auf den Stapel in C, da C keine weiteren Prozeduren ruft.

Der Stapel hilft auch, zwei weitere Probleme zu lösen:

- Prozedurparameter und -ergebnisse müssen jeweils zwischen den Prozeduren ausgetauscht werden. Parameter

können entweder in Registern übergeben werden (dies ist schnell, aber auf eine gewisse maximale Zahl von Parametern begrenzt) oder aber auf dem Stapel hinterlegt werden (dies ist langsamer, funktioniert aber für eine praktisch unbegrenzte Anzahl von Parametern –lediglich die maximale Stapelgröße ist hier eine Einschränkung).

- Prozeduren benötigen zumindest einen Teil der Register, um Zwischenergebnisse abspeichern zu können. Diese dürfen nicht Werte überschreiben, die eine rufende Prozedur in den Registern erwartet. Sinnvoll ist es daher, bei Prozeduraufrufen einen Teil der Register auf dem Stapel zu retten.

Für MIPS-Software gibt es die Konvention, die Register $4 bis $7 für Parameter zu verwenden und evtl. vorhandene weitere Parameter über den Stapel zu übergeben. Die Register $4 bis $7 müssen bei weiteren Prozeduraufrufen ebenfalls auf dem Stapel gesichert werden. Für diese Sicherung gibt es zwei Standards:

- **Speicherung durch die aufrufende Prozedur** (engl. *caller save*): die aufrufende Prozedur muss alle Register retten, deren Inhalt sie nach Rückkehr von der Prozedur unverändert vorfinden möchte.
- **Speicherung durch gerufene Prozedur** (engl. *callee save*): die gerufene Prozedur muss alle Register retten, die sie für Zwischenergebnisse benutzt.

2.2.1.7 Prozeduren mit Parametern

Wir wollen jetzt etwas umfangreichere Assemblerprogramme kennenlernen. Wir beginnen mit der Entwicklung der Assemblerversion einer Prozedur zum Vertauschen von zwei Variablen innerhalb eines Arrays. In C hat diese Prozedur die folgende Form:

```
swap (int v[], int k)
{
 int temp;
 temp   = v[k];
 v[k]   = v [k+1];
 v[k+1] = temp
}
```

Zur Übersetzung dieser Prozedur gehen wir in drei Schritten vor:

- Wir ordnen zunächst den Variablen Register zu.
- Anschließend erzeugen wir die Maschinenbefehle für den Rumpf der Prozedur.
- Schließlich sichern wir die Register.

Grundsätzlich muss man dabei zwischen den Parameterübergabetechniken *call by reference* und *call by value* unterscheiden. Bei der *call by value* Technik wird der Wert des Parameters selbst übergeben, bei der *call by reference* Technik die Adresse des Parameters. *Call by value* kann bei Ausgabeparametern nicht eingesetzt werden; *call by reference* ist dann eine Möglichkeit der Parameterübergabe. *Call by value* ist ineffizient, wenn große Datenstrukturen zu übergeben sind. Sonst müssten nämlich diese Strukturen bei Prozeduraufruf kopiert werden. *Call by reference* birgt die Gefahr, dass in der Prozedur der Wert der Variablen verändert wird, selbst wenn man dies nicht wollte.

Bei C kann der Benutzer durch Verwendung von * und &-Operatoren selbst entscheiden, wie die Parameter übergeben werden. Dabei müssen der Aufruf und die Realisierung der Prozedur zueinander passen. In C wird sehr viel Wert auf die Effizienz der Implementierung legt. Man würde deshalb bei größeren Variablen wie Arrays *Call by reference* nutzen. Bei skalaren Datentypen kann man *call by value* benutzt, wenn es sich im einen reinen Eingabeparameter handelt, sonst muss auch *Call by reference* benutzt werden. Für das Programmieren bedeutet dieses Vorgehen Nachteile bedeutet, weil bei der Übergabe von Arrays das Hantieren mit Adressen nicht mehr vermieden werden kann und leicht zu Programmierfehlern führen kann.

Gemäß MIPS-Konvention werden die Register $4 und $5 benutzt, um die Parameter zu in obigem Beispiel zu übergeben. Register $4 hat danach bei Aufruf der Prozedur die Adresse des Arrays v zu enthalten. Der Datentyp `int` kann durch 4 Bytes dargestellt werden. Für die Variable `temp` können wir ein Register verwenden, welches sonst nicht benutzt wird, also z.B. Register $15. Ein erster Realisierungsversuch des Rumpfes besteht aus dem folgenden Assemblerprogramm:

```
add $2, $4, $5  # addiere die Anfangsadresse von v und den Index k
lw  $15, 0($2)  # $15 (temp) enthält jetzt den Wert von v[k]
lw  $16, 1($2)  # $16 enthält jetzt den Wert von v[k+1]
sw  $16, 0($2)  # speichere v[k+1] unter v[k]
sw  $15, 1($2)  # speichere v[k] unter v[k+1]
```

Dieses Programm berücksichtigt allerdings nicht, dass ein Speicherwort jeweils vier Adressen entspricht. Daher muss der Index k mit 4 multipliziert werden, bevor er auf die Anfangsadresse von v addiert wird. Ebenso ist zu berücksichtigen, dass die Elemente v[k] und v[k+1] um 4 verschiedene Adressen haben. Das neue Programm hat die folgende Form:

```
ori $2, $0, 4   # impliziter load immediate-Befehl; lade Konstante
mult $2, $5     # k*4 jetzt in Register Lo
mflo $2         # k*4 jetzt in $2
add $2, $4, $2  # addiere die Anfangsadresse von v und 4*k
lw  $15, 0($2)  # $15 (temp) enthält jetzt den Wert von v[k]
lw  $16, 4($2)  # $16 enthält jetzt den Wert von v[k+1]
sw  $16, 0($2)  # speichere v[k+1] unter v[k]
sw  $15, 4($2)  # speichere v[k] unter v[k+1]
```

Jetzt müssen wir sicherstellen, dass unsere Prozedur keine Registerinhalte zerstört, die von einer aufrufenden Prozedur gesetzt werden. Wir nehmen an, dass wir die *callee save*-Konvention verwenden. Dann müssen die von uns benutzten Register $2, $15 und $16 gesichert werden. Hierfür benötigen wir insgesamt 3 Worte oder 12 Bytes auf dem Stapel. Das Register $29 zeigt standardmäßig auf den Stapel. Man beachte, dass der Stapel gemäß MIPS-Konvention **zu den niedrigen Adressen wächst (!)**. Der Code für die Korrektur des Stapelzeigers und das Speichern der Register hat das folgende Aussehen:

```
addi $29, $29, -12  # korrigiere Stapelzeiger um 12
sw   $2,  0($29)    # sichere $2
sw   $15, 4($29)    # sichere $15
sw   $16, 8($29)    # sichere $16
```

Da der Stapel auch für Unterbrechungen genutzt wird, muss immer zunächst der Stapelzeiger so verändert werden, dass freier Platz auf dem Stapel vorhanden ist (siehe Abschnitt über Unterbrechungen). Der `addi-Befehl` muss also der erste sein, wenn Unterbrechungen korrekt verarbeitet werden sollen. Die Befehle insgesamt müssen dem o.a. Rumpf vorangehen.

Vor dem Verlassen der Prozedur müssen wir noch die Register zurückschreiben, den Stapelzeiger wieder anpassen und an die rufende Prozedur zurückspringen. Dies leisten die folgenden Befehle, die nach dem Rumpf der Prozedur anzuordnen sind:

```
lw   $2,  0($29)    # restauriere $2
lw   $15, 4($29)    # restauriere $15
lw   $16, 8($29)    # restauriere $16
addi $29, $29, 12   # korrigiere Stapelzeiger um 12
jr   $31            # Rücksprung
```

Nunmehr betrachten wir ein längeres Beispiel: ein Programm zum Sortieren innerhalb eines Arrays. Wir benutzen dazu den nachstehend angegebenen *bubble sort*-Algorithmus, dessen genaue Funktion aber hier nicht interessiert:

```
int v[10000];
sort (int v[], int n)
{
int i, j;
  for (i=0; i<n; i=i+1) {
```

```
          for (j=i-1; j>=0 && v[j]>v[j+1]; j=j-1) {   % && : UND-Operation
            swap(v,j);
          }
        }
      }
```

Die beiden Parameter von `sort`, die Adresse von `v` und den Wert von `n`, erwarten wir wieder in den Registern $4 und $5. Wir ordnen `i` dem Register $19 und `j` dem Register $17 zu. Nachfolgend zeigen wir sofort das vollständige Programm. Zur Programmentwicklung ist aber ebenso wie im vorherigen Beispiel zunächst einmal der Rumpf der Prozedur zu entwickeln, bevor die notwendigen Register auf dem Stapel gesichert werden.

```
# Retten aller Register; Code nach dem Schreiben des Rumpfes erstellen!
       addi $29, $29, -36 # Platz für 9 Register auf dem Stapel
       sw   $15, 0($29)   # rette Register
       sw   $16, 4($29)   # rette Register
       sw   $17, 8($29)   # rette Register
       sw   $18,12($29)   # rette Register
       sw   $19,16($29)   # rette Register
       sw   $20,20($29)   # rette Register
       sw   $24,24($29)   # rette Register
       sw   $25,28($29)   # rette Register
       sw   $31,32($29)   # rette Rückkehradresse
       add $18, $4, $0    # sichere Parameter $4 in $18
       add $20, $5, $0    # sichere Parameter $5 in $20
# Initialisierung Schleife 1
       add $19,$0,$0   # i := 0
for1:  slt $8,$19,$20  # $8 = 0, wenn $19 >= $20(i >=n)
       beq $8,$0, ex1  # springe nach ex1, wenn i>=n.
# Initialisierung Schleife 2
       addi $17,$19,-1    # j:=i-1
for2:  slti $8, $17,0     # $8 = 0, wenn $17<0 (j<0)
       bne  $8, $0,ex2    # wenn $17 < 0, dann kein weiterer Test mehr nötig.
       muli $15, $17,4    # wir nehmen an, dieser Pseudobefehl multipliziert mit 4
       add  $16, $18,$15  # $16 := Adresse von v[j]
       lw   $24, 0($16)   # $24 := v[j]
       lw   $25, 4($16)   # $25 := v[k+1]
       slt  $8, $25, $24  # $8 = 0, wenn $25 >= $24
       beq  $8, $0, ex2   # springe nach ex2, wenn $25 >= $24
# Vorbereitung des Prozeduraufrufs
       add $4, $18, $0    # $4 (erster Parameter) ist die Adresse des Arrays v
       add $5, $17, $0    # $5 (zweiter Parameter) ist der Wert von j
       jal swap
# Ende Schleife 2
       addi $17, $17, -1  # j := j-1
       j    for2          # an den Anfang der Schleife
ex2:
# Ende Schleife 1
       addi $19,$19,1  # i = i+1
       j   for1        # an den Anfang der Schleife
ex1:
       lw   $15, 0($29)   # restauriere Register
       lw   $16, 4($29)   # restauriere Register
       lw   $17, 8($29)   # restauriere Register
       lw   $18,12($29)   # restauriere Register
       lw   $19,16($29)   # restauriere Register
       lw   $20,20($29)   # restauriere Register
       lw   $24,24($29)   # restauriere Register
       lw   $25,28($29)   # restauriere Register
```

```
lw   $31,32($29)   # restauriere Rückkehradresse
addi $29, $29, 36  # korrigiere Stapelzeiger
jr   $31
```

In diesem Beispiel haben wir von möglichen Optimierungen nicht Gebrauch gemacht.

2.2.2 Allgemeine Sicht der Befehlsschnittstelle

Bislang haben wir mit der MIPS-Befehlsschnittstelle einen speziellen Fall einer solchen Schnittstelle kennengelernt. In den folgenden Abschnitten wollen die Befehlsschnittstelle oder **externe Rechnerarchitektur** (engl. *instruction set architecture (ISA)*) allgemein besprechen. Zur Befehlsschnittstelle gehören

- die nutzbaren Maschinenbefehle
- ein Modell des Speichers
- die logische Sicht auf das Interrupt-System (siehe Abschnitt 2.2.2.3)

Wir betrachten zunächst eine allgemeine Sicht auf das Speichermodell. Insgesamt realisieren Speicher den **Zustand** eines Rechensystems bzw. eines Programms.

2.2.2.1 Das Speichermodell

2.2.2.1.1 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. Ein Übergang auf 64-Bit Adressen hat bei Großrechnern schon länger stattgefunden und findet bei PCs derzeit (2013) statt.

2.2.2.1.2 *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]).

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

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

little endian		big endian	
Adresse	Wert	Adresse	Wert
"xx00"	"02"	"xx00"	"00"
"xx01"	"01"	"xx01"	"00"
"xx10"	"00"	"xx10"	"01"
"xx11"	"00"	"xx11"	"02"

Tabelle 2.7: Speicherbelegung durch eine 32-Bit Zahl

Die Zahl 258 = 256 + 2 würde in den beiden Systemen wie in Tabelle 2.7 abgelegt werden. Dabei entspricht "xx00" der Anfangsadresse der Zahl und "02" dem abgelegten Bitvektor aus zwei Hexadezimalstellen.

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

- **Bei der Zeichenketten-Verarbeitung**

  Tabelle 2.8 zeigt, wie das Wort ”Esel” bei den beiden Speicherbelegungsverfahren abgelegt wird.

little endian		big endian	
Adresse	Wert	Adresse	Wert
"*xx*00"	l	"*xx*00"	E
"*xx*01"	e	"*xx*01"	s
"*xx*10"	s	"*xx*10"	e
"*xx*11"	E	"*xx*11"	l

Tabelle 2.8: 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 ?

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.

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 2.4 vorstellen.

2.2.2.1.3 Registerspeicher: Neben den großen Hauptspeichern besitzen praktisch 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 wäre für einen solchen zeitkritischen Befehl möglicherweise auch die Taktperiode zu kurz. 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 Stapelzeiger (engl. *stack pointer*).

Beispiele:

1. Motorola 68000er-Prozessoren.

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

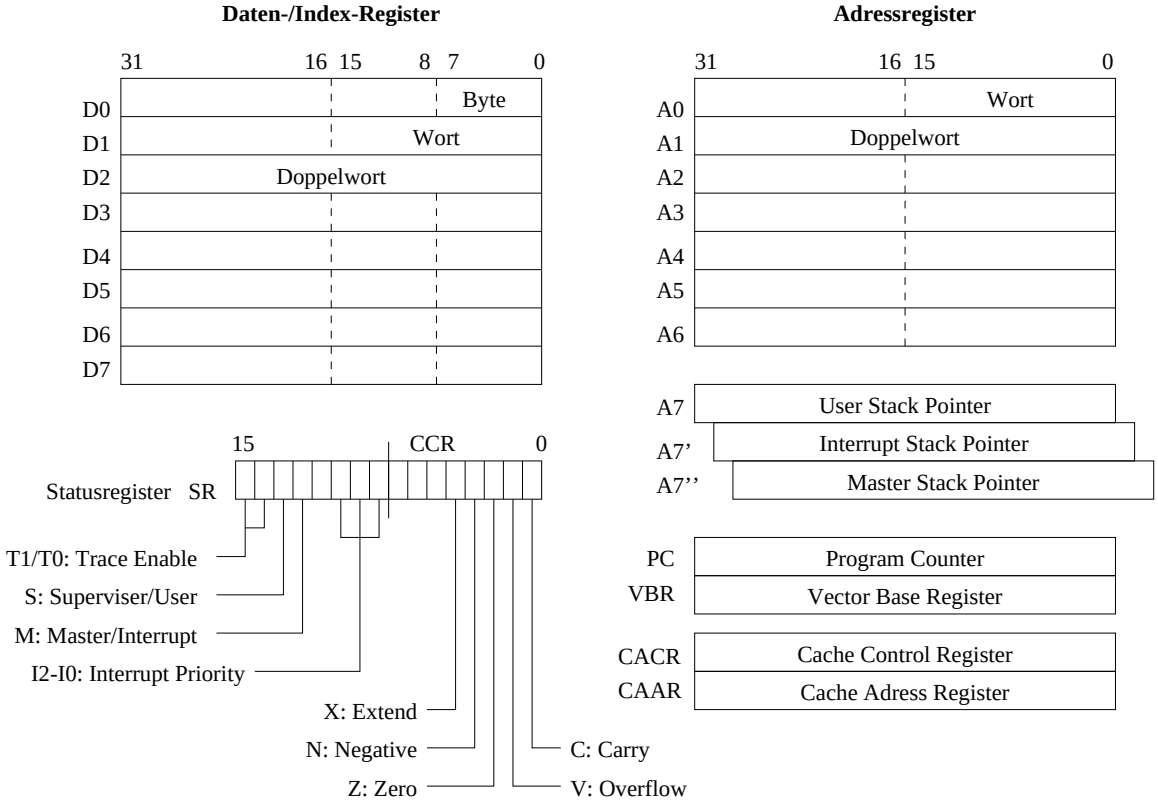


Abbildung 2.10: Registersatz der Motorola 680x0-Prozessoren

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 Adressregisters 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).

2. MIPS R2000-8000

   Die Register dieser Architektur haben wir bereits ausführlich besprochen. Es gibt keine Condition-Code-Register.

3. Intel 80x86

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

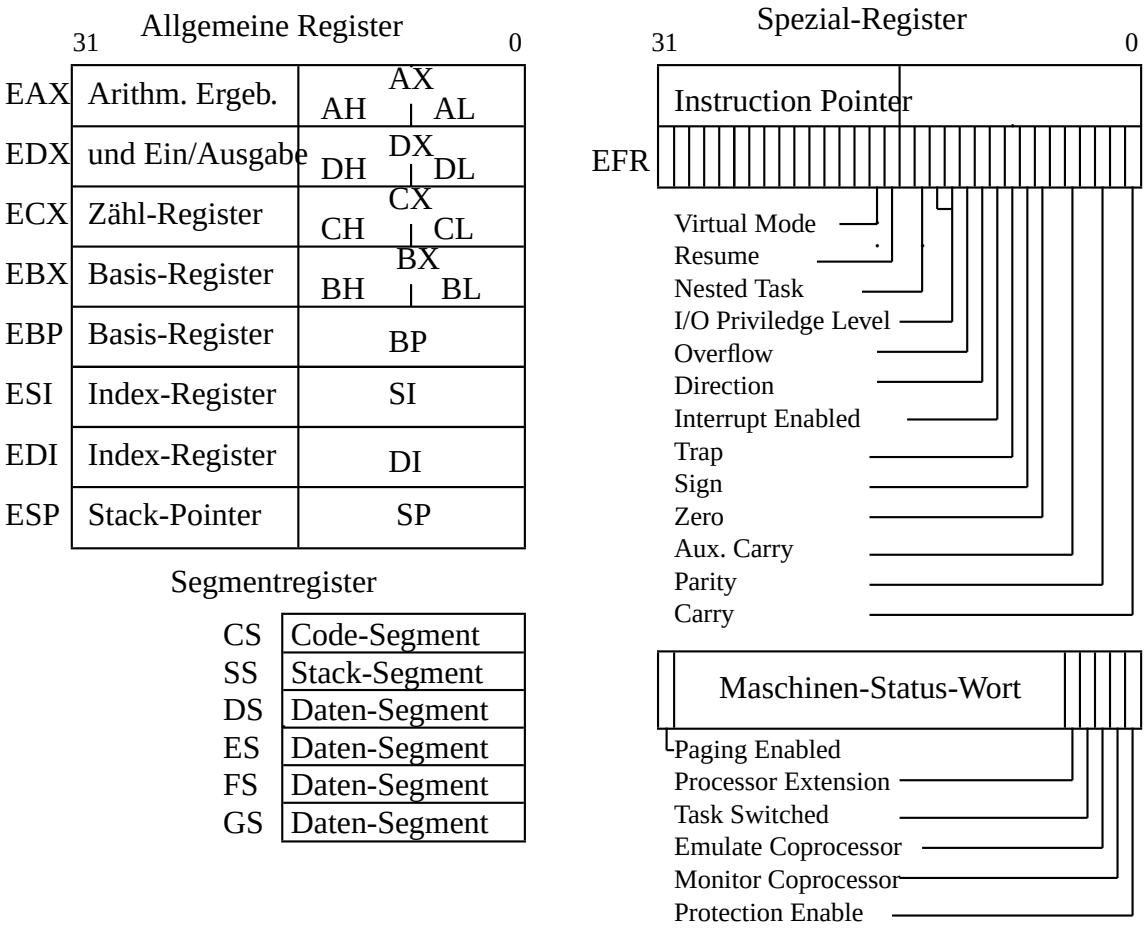


Abbildung 2.11: Registersatz der Intel-Prozessoren 80386 und 80486

2.2.2.2 Befehlssätze

Zur Befehlsschnittstelle gehört neben den Speichern v.a. der Befehlssatz. Verfügbare Befehle können anhand der so genannten **Befehlsgruppen** klassifiziert werden. Beispielsweise kann eine Einteilung in folgende Befehlsgruppen vorgenommen werden:

- Transferbefehle (lw, sw, 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

- Schiebebefehle (Angabe von Richtung und Quelle für das Auffüllen)
- Sprungbefehle
- Kontrollbefehle (Disable Interrupt)
- Ununterbrechbare Befehle (z.B. Test-and-set (TAS))

2.2.2.2.1 Adressierungsarten: Speicherzellen werden wie bereits erwähnt über Speicher-**Adressen** angesprochen. Die Berechnung dieser Adressen kann durch die Maschinenbefehle unterschiedlich vorgeschrieben werden. 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.

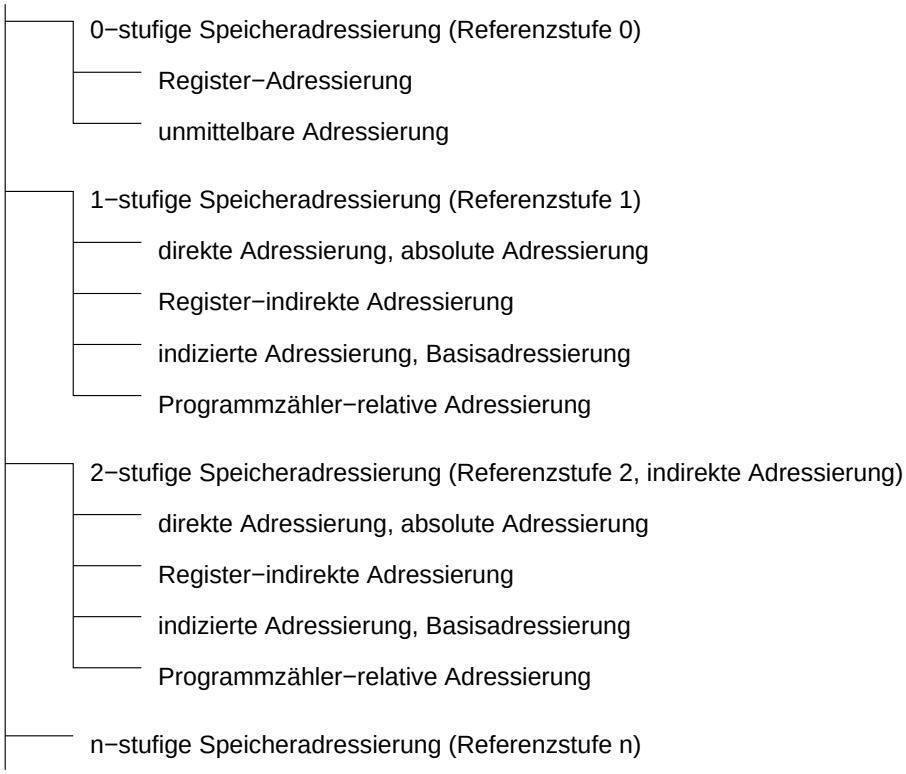


Abbildung 2.12: Adressierungsarten

Beispiele für die einzelnen Arten sind die folgenden[10]:

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

Assembler-Notation	Semantik	
mflo $15	Reg[15]:= Lo	(MIPS)
CLR,3	D[3]:=0	(68.000)
LDPSW,3	D[3]:=PSW	(68.000): lade Programm-Status-Wort aus Spezialregister

  - unmittelbare Adressierung, immediate Adressierung
    Beispiele: (#: bezeichnet unmittelbare Adressierung beim Assembler des 68.000)

Assembler-Notation	Semantik	Kommentar
lui $15,3	Reg[15]:= 3<<16	(MIPS)
LD D3,#100	D[3]:= 100	(68.000), 100 gespeichert im Adressteil
LD D3,#100	D[3]:= 100	(68.000), 100 gespeichert im Adressteil

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

Assembler-Notation	Semantik	Kommentar
lw $15,AdrTeil	Reg[15]:= Speicher[AdrTeil]	(MIPS)
LD D3,AdrTeil	D[3]:= Speicher[AdrTeil]	(68.000)

  - Register-Indirekte Adressierung

Assembler-Notation	Semantik	
lw $15,($2)	Reg[15]:= Speicher[Reg[2]]	(MIPS)
LD D3,(A4)	D[3]:= Speicher[A[4]]	(68.000)

    Varianten: Prä/Post- De/Increment zur Realisierung von Stapeloperationen.
  - Relative Adressierung, Indizierte Adressierung, Basis-Adressierung

Assembler-Notation	Semantik	Kommentar
lw $15,AdrTeil($2)	Reg[15]:= Speicher[AdrTeil+Reg[2]]	(MIPS)
LD D3,AdrTeil(IReg)	D[3]:= Speicher[AdrTeil+D[IReg]]	(68.000)

    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	Kommentar
LD D3,AdrTeil(IReg)(BReg)	D[3]:= Speicher[AdrTeil+D[IReg]+D[BReg]]	(68.000)

      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

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

- Programmzähler-Relative Adressierung

Assembler-Notation	Semantik	Kommentar
BRA $7FE	PC := PC + $7FE +i, mit i=0,1,2 oder 4	(68.000)

$ bedeutet hier einen hexadezimalen Wert. 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

  D[Reg] := Speicher[Speicher[AdrTeil+PC]]
  PC := Speicher[AdrTeil+PC+i]

  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.

2.2.2.2.2 ***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[11]. Es gibt die folgenden Fälle:

[11] Quelle: [Jes75].

- 3-Adressmaschinen, 3-Adressbefehle

  3 vollständige Adressen, IBM-Bezeichnung: Befehle vom „Typ **SSS**"
  (d.h. Befehle der Form Speicher[a] := Speicher[b] ∘ 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.

  In Abwandlung des Begriffs „SSS"-Befehl bezeichnet man die arithmetischen Befehle der MIPS-Architektur auch als „RRR"-Befehl, da drei Registernummern angegeben werden können.

  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!

  ← 32 Bit →

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. Man sagt, die Befehle seien vom „**Typ SS**". 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.

  ← 32 Bit →

Op-Code	weitere Informationen
a	
b	

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 so genannten 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 (so genanntes *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. Wir gehen dazu von der Baumdarstellung des Ausdrucks aus (siehe Abb. 2.14).

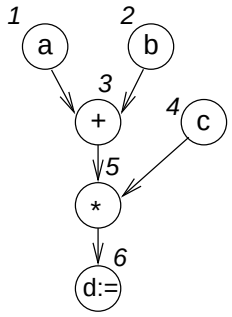


Abbildung 2.14: Baumdarstellung; Zahlen kennzeichnen die Betrachtungsreihenfolge

Wir durchlaufen den Baum immer so, dass wir uns zunächst die beiden Zweige eines Knotens ansehen, bevor wir uns mit dem Inhalt des Knoten beschäftigen. Wenn wir uns nun mit einem Knoten beschäftigen, so wird aus jeder lesenden Referenz einer Variablen ein `PUSH`-Befehl für diese Variable, aus jeder arithmetischen Operation der entsprechende arithmetische Befehl und aus der Zuweisung der `POP`-Befehl für die betroffene Variable.

Auf diese Weise übersetzen wir die o.a. Anweisung wie folgt:

```
PUSH a        - - a: Adresse im Hauptspeicher
PUSH b
ADD           - - 0-Adressbefehl
PUSH c
MULT          - - 0-Adressbefehl
POP d         - - speichere oberstes Stapelelement 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: bei verschiedenen abstrakten Maschinen (einschließlich der virtuellen Java-Maschine, s.u.), bei manchen Taschenrechnern und beim Gleitkommaprozessor der Intel 80x86-Serie.

Die Befehlssätze moderner Rechner enthalten häufig Maschinenbefehle mit einer unterschiedlichen Anzahl von Operanden-Adressen.

2.2.2.2.3 Klassifikation von Befehlssätzen

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 und 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 so genannte *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 in Abb. 2.9 gezeigte Format:

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 2.9: Befehlsformate der 68000er-Prozessorfamilie

Tabelle 2.10 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 2.10: MOVE-Befehle des 68000

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

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[12]. 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 so genannte CPI-Wert, der wie folgt definiert wird:

[12]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.

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 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).

2.2.2.3 Unterbrechungen

Zur Befehlsschnittstelle 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). Bei der MIPS-Maschine wird hierzu der Speicherbereich genutzt, auf den das Register 29 zeigt. Das Register 29 muss daher immer so gesetzt sein, dass eine (in der Regel unerwartete) Unterbrechung keine anderweitigen Informationen im Speicher überschreibt. Besser wäre hier ein separater Unterbrechungs-Stapel, wie ihn z.B. die Motorola 680x0-Prozessoren haben.
- **Verzweigen aufgrund der Unterbrechungsursache**
- ggf. **Senden einer Bestätigung** (interrupt acknowledge)
- **Ausführen der Interrupt-Routine**
- **Restaurieren des Prozessorzustandes**. Hier wird bei der MIPS-Maschine wieder auf das Register 29 zugegriffen.
- **Beenden der Interrupt-Routine** durch einen *return from interrupt*- Befehl. Die MIPS-Maschine enthält dazu einen solchen `rti`-Befehl. Die Rückkehr-Adresse wird ebenfalls dem Stapel entnommen.

Tabelle 2.11 enthält Klassen von Unterbrechungen[13].

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 2.11: Unterbrechungsklassen (Auszug)

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

2.2.2.4 Systemaufrufe und Ausnahmen in der MIPS-Maschine

Systemaufrufe

Wenn auf einem Prozessor ein Betriebssystem zur Verfügung steht, dann kann man neben den Maschinenbefehlen, die direkt durch die Hardware unterstützt werden, auch spezielle Befehle ausführen, welche Servicefunktionen des Betriebssystems (auch SVC=*supervisor call* genannt) zugänglich machen. MARS stellt über den `syscall`-Befehl eine vereinfachte Menge von solchen Systemaufrufen bereit (siehe Tabelle 2.12 und A.1 im Anhang). Danach muss das Register 2 (symbolisch auch als `$v0` bezeichnet) die Nummer enthalten, welche die Servicefunktion identifiziert. Die Register 4 bis 7 (symbolisch als `$a0` bis `$a3` bezeichnet), können die Argumente des Systemaufrufs enthalten.

Servicename	Servicenummer	Argumente	Ergebnis
print_int	1	`$a0` = integer	-
print_float	2	`$f12` = Gleitkommazahl	-
print_double	3	`$f12` = Gleitkommazahl, doppelt genau	-
print_string	4	`$a0` = Adresse der Zeichenkette	-
read_int	5	-	in `$v0`
read_float	6	-	in `$f0`
read_double	7	-	in `$f0`, doppelt genau
read_string	8	`a0`: Pufferadresse, `a1`: Pufferlänge	-
exit	10		-
read_character	12	-	in `$v0`: gelesenes Zeichen
read_file	...	..	...

Tabelle 2.12: Systemaufrufe in MARS (Auszug)

Mit diesen Systemaufrufen kann beispielsweise die folgende kleine Ausgaberoutine erstellt werden (siehe Anhang):

```
.glob main
        .data
  str:  .asciiz "die Antwort ist: " # Nullbyte am Ende
        .text
  main: li $v0, 4        # Nummer des Systemaufrufs
        la $a0, str      # Adresse der Zeichenkette
        syscall          # Systemaufruf
        li $v0, 1        # Nummer des Systemaufrufs
        li $a0, 5        # integer
        syscall          # Systemaufruf
```

Diese Ein- und Ausgaben des MARS-Simulators beziehen sich auf das Terminal. Es gibt in MARS auch Systemaufrufe zur Ein- und Ausgabe von Dateien, aber keine Systemaufrufe zur Verwaltung mehrerer Prozesse. Ein MARS-Programm wird üblicherweise mit dem exit-SVC abgeschlossen.

Systemaufrufe werden in der Regel durch nicht hardwaremäßig realisierte Maschinenbefehle codiert (beim MIPS als 0x0000000c). Der entsprechende Befehl erzeugt zur Laufzeit eine Ausnahme. Die Ausnahmebehandlung sorgt dann für eine Verzweigung an eine passende Stelle im *kernel*-Bereich. Der MARS-Simulator simuliert diese Ausnahmebehandlung allerdings nicht mit MIPS-Befehlen, sondern direkt im Simulator. Die Verzweigung in den *kernel*-Bereich ist daher in den Simulationen nicht zu sehen.

Der Traphandler des SPIM-Simulators

Zur Behandlung von Unterbrechungen dient in der MIPS-Maschine der so genannte Coprozessor 0. Dieser enthält Spezialregister, über die die notwendigen Statusabfragen vorgenommen werden können, bzw. über welche die notwendigen Beeinflussungen vorgenommen werden können. Einige Aussagen dazu macht der Anhang B. Die Tabelle auf Seite 130 enthält die Nummern der relevanten Coprozessor-Register. Leider sind die Informationen relativ spärlich. Der MARS-Simulator behandelt Unterbrechungen üblicherweise im Simulatorcode. Man kann zwar einen *traphandler* angeben, aber standardmäßig scheint ein solcher nicht mitgeliefert zu werden, im Unterschied zum älteren SPIM-Simulator. Eine Analyse des Standard-*traphandlers* der SPIM-Distribution gibt Hinweise zur praktischen Programmierung. Das Prinzip des *traphandlers* zeigt die folgende vereinfachte Darstellung:

```
# SPIM S20 MIPS simulator. The default trap handler for spim.
# Copyright (C) 1990-2000 James Larus, larus@cs.wisc.edu. ALL RIGHTS RESERVED.
# SPIM is distributed under the following conditions:
# You may make copies of SPIM for your own use and modify those copies.
# All copies of SPIM must retain my name and copyright notice.
# ...
s1: .word 0         # Speicher zum Sichern von Registern; Stapel nicht nutzbar ($29 korrekt?)
s2: .word 0         # dto.
 .ktext 0x80000180  # Code gehört zum Kernel Textsegment ab der angegebenen Adresse
 .set noat          # Nutzung von $1 durch Assembler wird verboten
 move $k1 $at       # Retten von $1 in $27 (für Kernel reserviert)
 .set at            # Nutzung durch von $1 durch Assembler wieder möglich
 sw $v0 s1          # Rette $v0 in feste Speicherzelle
 sw $a0 s2          # Rette $a0 in feste Speicherzelle
                    # Nutzung von $1=$at, $v0 und $a0 von nun an erlaubt
 mfc0 $k0 $13       # Lade Unterbrechungsursache
 ...                # Verzweige abhängig von $k0
 ...                # Lösche Ursachenregister
 lw $v0 s1          # Rückschreiben von $v0
 lw $a0 s2          # Rückschreiben von $v0
 .set noat
 move $at $k1       # Restore $at
 .set at
 rfe                # Restaurieren von Statusregistern (Kernel-Mode etc.)
 mfc0 $k0 $14       # Hole alten Wert des PC
 addiu $k0 $k0 4    # Erhöhe um 4
 jr $k0             # Führe unterbrochenes Programm fort
```

Danach führen alle Unterbrechungen zu einem Aufruf von Befehlen ab Adresse `0x80000180` im *kernel*-Bereich. Im *traphandler* werden zunächst einige Register gerettet. Generell muss dabei geprüft werden, ob während einer Unterbrechungsbehandlung eine weitere Unterbrechungsbehandlung erforderlich werden kann. Dies könnte beispielsweise passieren, wenn während einer Unterbrechungsbehandlung noch eine Schutzverletzung auftritt, die sofort zu behandeln ist. In diesem Fall entsprechen die Unterbrechungsbehandlungen geschachtelten Prozedurbehandlungen und man müsste, wie bei geschachtelten Prozeduren, Registerinhalte auf einem Stapel sichern. Eine Unterbrechungsbehandlung, die selbst wieder unterbrochen werden kann, nennt man **wiedereintrittsfähig** (engl. *reentrant*).

Der SPIM-Simulator rechnet nicht damit, dass während einer Unterbrechung eine weitere Unterbrechungsbehandlung erforderlich wird und sichert daher Register in festen Speicherzellen. Anschließend muss mit dem Befehl `mfc0 $k0 $13` zunächst einmal die Ursache der Unterbrechung herausgefunden und anschließend entsprechend verzweigt werden. Es gibt also keinen *vectored interrupt*. Der vollständige *traphandler* ist in Anhang C wiedergegeben. Der SPIM-Traphandler kann auch zusammen mit einem MARS-System genutzt werden. Unklar bleibt, ob dieser *traphandler* in einem echten MIPS-System ausreichen würde, oder ob z.B. externe Interrupts durch das Löschen des Ursachenregisters verloren gehen könnten.

2.2.3 Das Von Neumann-Modell

Fast alle der heute gebräuchlichen Rechner folgen dem Modell des so genannten **Von-Neumann-Rechners**. Im Anschluss an die Darstellung der Befehlsschnittstelle aus heutiger Sicht wollen wir hier noch einmal angeben, was eigentlich das Von-Neumann-Modell ausmacht. Dies sind die Kernpunkte der Ideen von Neumann's[14]:

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. 2.15 zeigt diese Einheiten und deren Verschaltung innerhalb einer busorientierten Architektur.

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.

[14] Quelle: [Jes75]

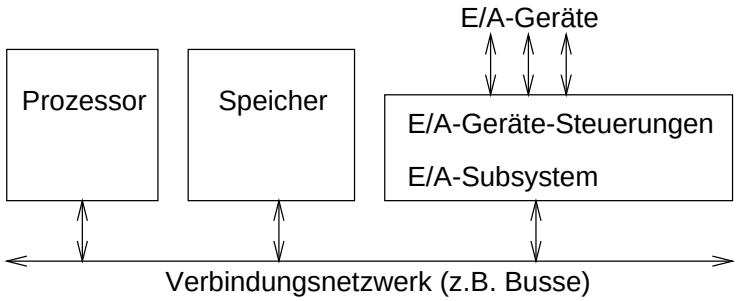


Abbildung 2.15: Funktionseinheiten des von-Neumann-Rechners

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.

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. 2.16), wie sie bei den so genannten Datentyp-Architekturen [Gil92] vorgesehen ist, hat sich im Allgemeinen nicht durchgesetzt.

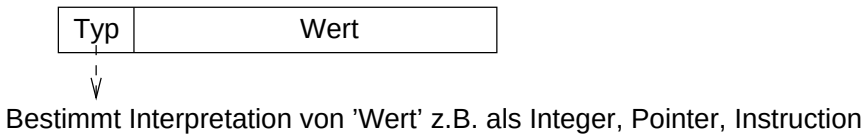


Abbildung 2.16: Explizite Datentyp-Speicherung

2.3 Register-Transfer-Strukturen

Computers in the future may weigh no more than 1.5 tons

[Popular mechanics, 1949]

Die Realisierung eines Befehlssatzes mittels RT-Strukturen nennen wir auch **Mikroarchitektur** bzw. **interne Rechnerarchitektur**, in Abgrenzung von den Bezeichnungen **Architektur** bzw. **externe Rechnerarchitektur** für die Befehlsschnittstelle.

2.3.1 Komponenten der internen Mikroarchitektur

Zur Darstellung der Struktur werden wir verschiedene Symbole von Funktionseinheiten benutzen.

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. 2.17, links) und die jeweils benötigten Leitungen dargestellt. Abb. 2.17, rechts, zeigt die Darstellung der häufig benötigten Multiplexer, die der Auswahl von Daten dienen.

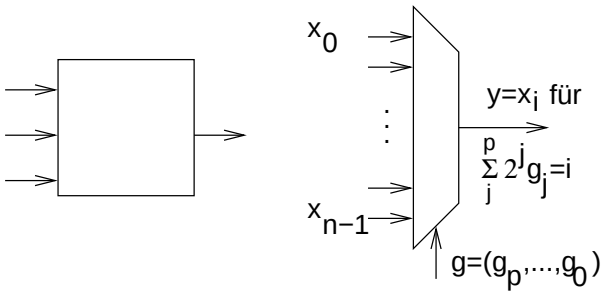


Abbildung 2.17: Funktionseinheiten und Multiplexer

Dekoder, welche Bitvektoren in andere Bitvektoren umkodieren, werden in Abb. 2.18 (links) gezeigt.

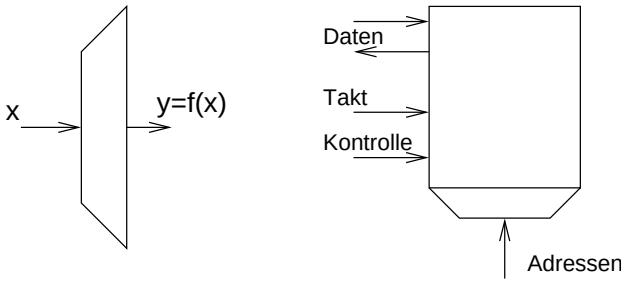


Abbildung 2.18: Dekoder und adressierbarer Speicher

Häufig werden wir Speicher verwenden. Abb. 2.18 (rechts) zeigt unsere Darstellung von Speichern. Im Gegensatz zu den meisten Büchern benutzen wir eine explizite Darstellung des Adressdekoders. Den Adressdekoder von Speichern werden wir benutzen, um klarzustellen, dass gewisse interne Tabellen **adressiert** werden. Wir werden den Dekoder auch explizit darstellen, damit der Adresseingang von Speichern leichter erkannt werden kann. In den mündlichen Prüfungen hat sich herausgestellt, dass so manche Missverständnisse vermieden werden können. In vielen Fällen werden so genannte Multiport-Speicher verwendet. Dies sind Speicher mit mehreren Adresseingängen. Über jeden der Adresseingänge können unabhängig voneinander Daten gelesen oder geschrieben werden. Sofern dieselben Adressen benutzt werden, werden natürlich auch dieselben Speicherzellen angesprochen. Abb. 2.19 zeigt Symbole verschiedener Multiport-Speicher.

Die Zuordnung zwischen den Adresseneingängen und den Datenein- bzw. ausgängen erfolgt über die Beschriftung.

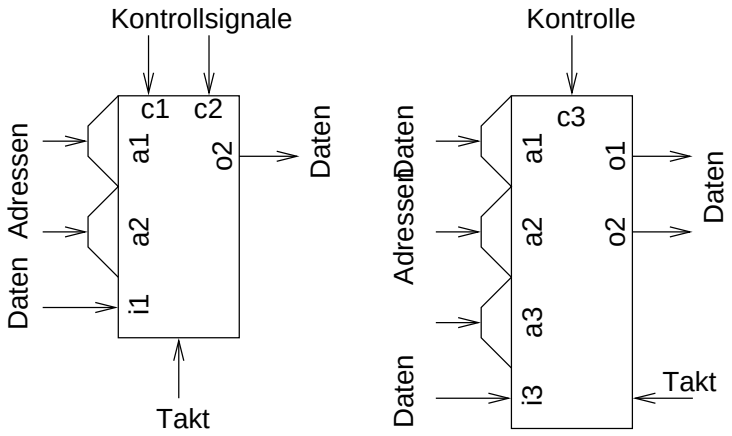


Abbildung 2.19: Multiport-Speicher

Der linke Teil der Abbildung zeigt einen Zweiportspeicher, wobei Port 1 einen reinen Dateneingang bildet und Port 2 einen reinen Ausgang (1 Schreib- und ein Leseport). In diesem Fall erfordern beide Ports Kontrollsignale an den Eingängen c1 bzw. c2, um die jeweilige Operation auszuwählen (schreiben bzw. nicht schreiben und lesen bzw. nicht lesen). Der Takt ist in diesem Fall explizit dargestellt. Der rechte Teil der Abbildung zeigt einen Dreiportspeicher mit einem Schreib- und zwei Leseports. Die beiden Leseports werden beispielsweise benötigt, um zwei Argumentregister in der MIPS-Architektur gleichzeitig lesen zu können. In diesem Fall ist angenommen, dass lediglich der Schreibport ein Kontrollsignal erfordert. Schreibports benötigen immer Kontrollsignale, sofern man das Beschreiben mit jedem Takt verhindern will. Leseports werden normalerweise immer die zur Adresse gehörige Zelle mit einer gewissen Verzögerung auslesen, bräuchten somit in der Regel keine Kontrolleingänge. Große Speicher möchte man u.a. des Energieverbrauchs wegen allerdings nicht immer lesen. Für diese ist es daher sinnvoll, mittels einer Kontrollleitung angeben zu können, ob überhaupt gelesen werden soll. Takt- und Kontrollleitungen werden wir nur dann einzeichnen, wenn sie für das Verständnis wichtig sind.

Die Abb. 2.20 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.

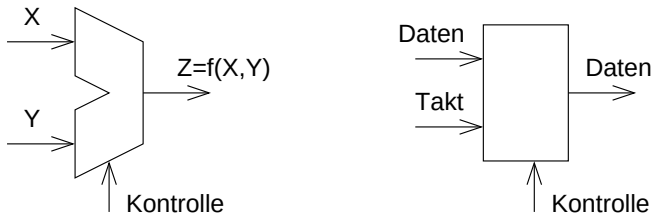


Abbildung 2.20: Dyadischer Operator und Register

2.3.2 Mikroprogrammierung

Wir wollen im Folgenden im Detail kennen lernen, wie man einen Befehlssatz auf einer Hardware-Struktur realisieren kann. Wir versuchen dabei zunächst, mit möglichst wenigen Hardware-Blöcken auszukommen. Abb. 2.21 zeigt eine Hardware-Struktur, die so entworfen ist, dass sie den MIPS-Befehlssatz ausführen kann[15].

Der Registerspeicher `Reg` ist 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

[15] Die Struktur basiert auf den Angaben bei Hennessy und Patterson ([HP96], Kapitel 5). Im Unterschied zu diesen Autoren nehmen wir hier aber nicht bereits Annahmen über die später betrachtete Realisierung mit einem Fließband vorweg.

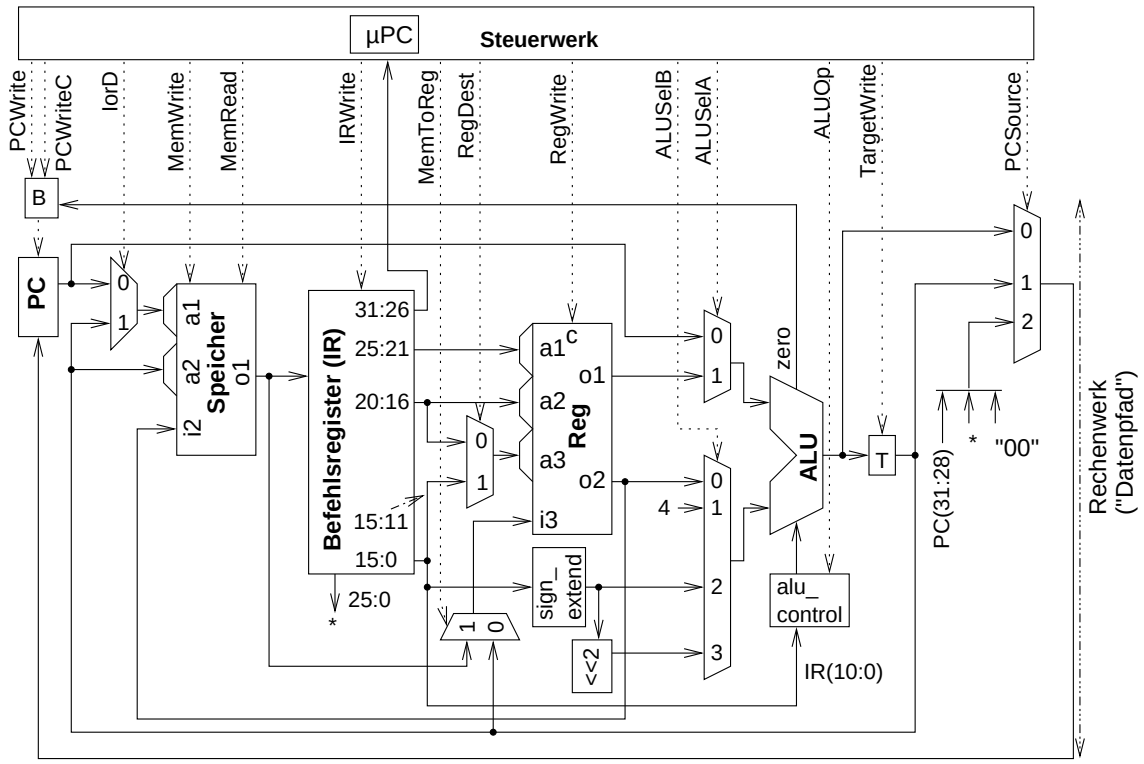


Abbildung 2.21: Hardware-Struktur zur Ausführung des MIPS-Befehlssatzes

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. 2.22 beschreibt.

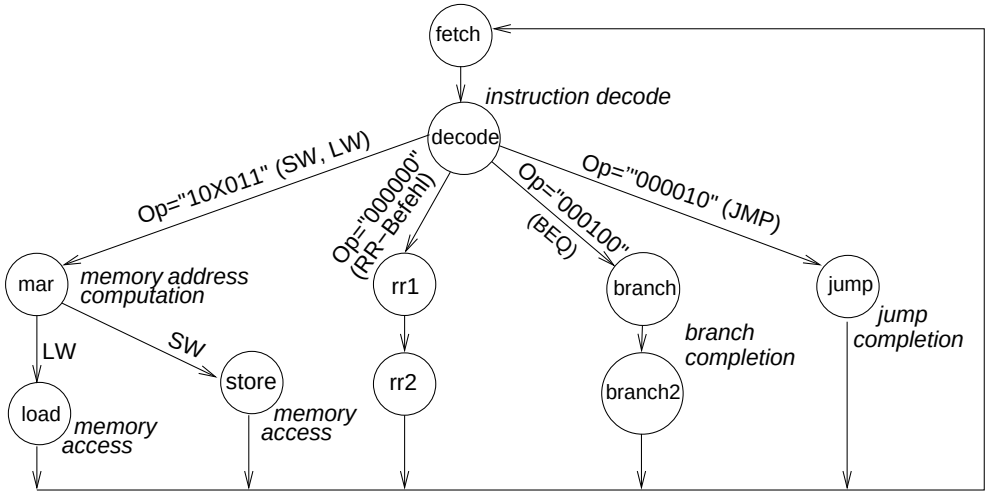


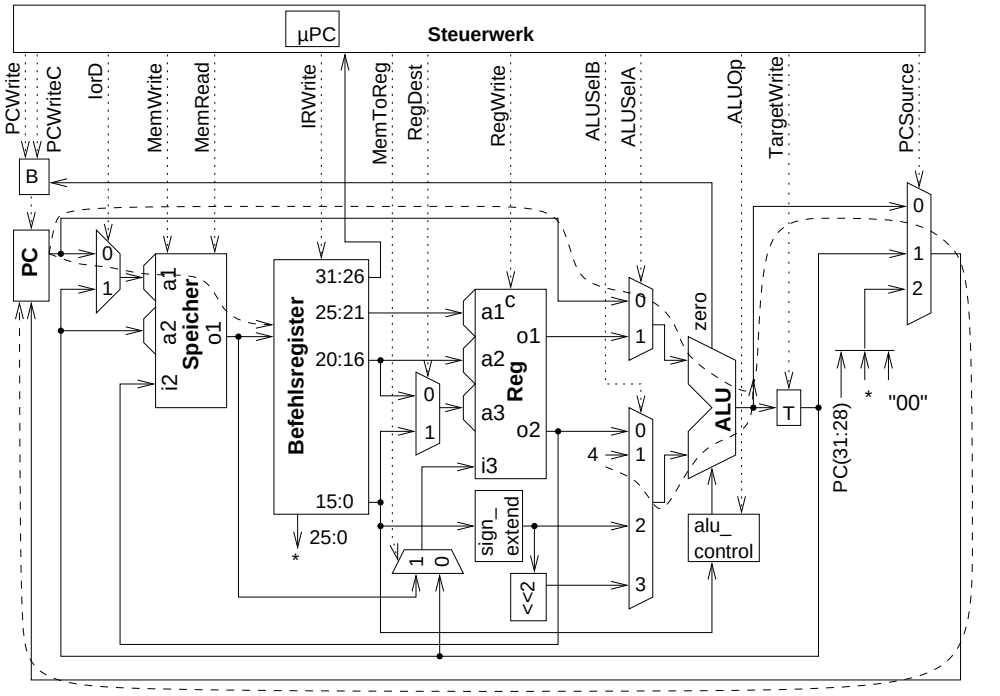
Abbildung 2.22: Zustandsgraph der Ausführung von MIPS-Befehlen

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.

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. 2.23 gestrichelt eingezeichnet.



Steuerwerk: Ausgaben: IorD=0, MemRead='1', MemWrite='0', IRWrite='1', ALUSelA=0, ALUSelB=1, ALUOp=+, PCSource=0, PCWrite='1', RegWrite='0', TargetWrite='0'; Folgezustand = decode

Abbildung 2.23: 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.

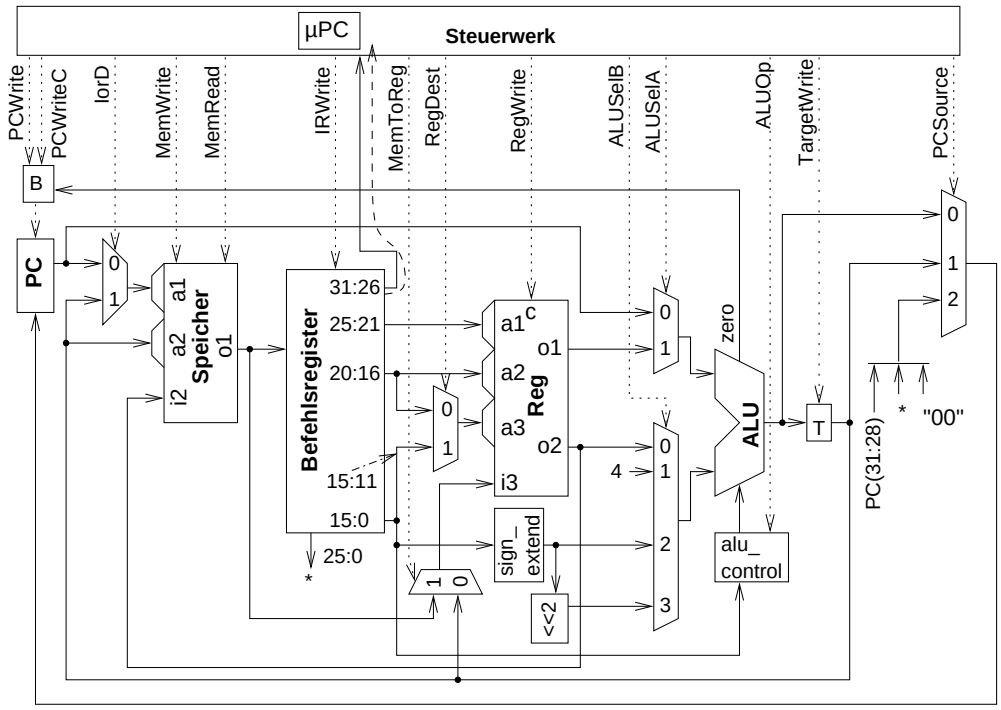
Im nächsten Schritt wird der aktuelle Befehl dekodiert (siehe Abb. 2.24).

Zu diesem Zweck wird das Zustandsregister des Steuerwerks mit einem Wert geladen, der den Typ des auszuführenden Befehls eindeutig charakterisiert (ein beliebter Trick ist hier, den Operationscode des Befehls um wenige konstante Bits zu erweitern und in das Zustandsregister zu laden. Die Kodierung der übrigen Zustände verwendet dann andere Werte der Erweiterungsbits).

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.

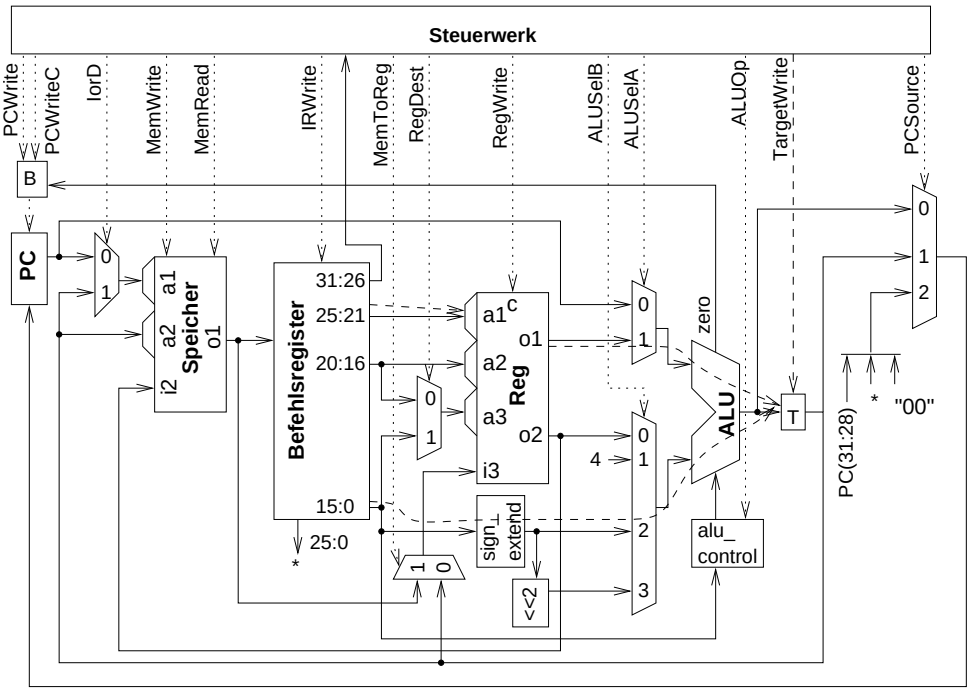
Im Falle von Speicherbefehlen wird zunächst die effektive Adresse ausgerechnet. Diesem Zweck dient der Zustand mar. Die in diesem Zustand benutzten Datenwege zeigt die Abb. 2.25.

Die effektive Adresse wird im Register T abgelegt. Man beachte, dass alle Additionen von Konstanten vorzeichener-



Steuerwerk: Folgezustand = CASE Befehlsregister(31:26) OF when 'LW' then.. when 'SW' then when ..... xxWrite='0'

Abbildung 2.24: Benutzung der Struktur in der Dekodier-Phase

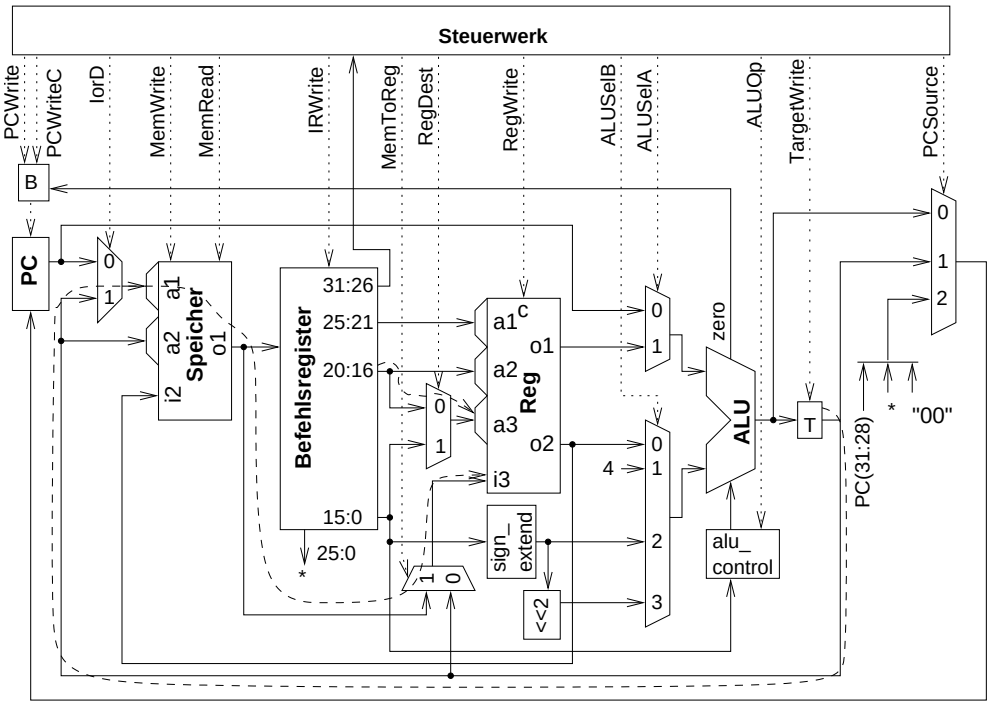


Steuerwerk: Ausgaben: ALUSelA=1, ALUSelB=2, ALUOp=+, TargetWrite='1', andere Register: xxWrite='0'
Folgezustand = if Befehlsregister(31:26)='LW' then load else store,

Abbildung 2.25: Benutzung der Struktur in der Adressrechnungs-Phase

weitert erfolgen.

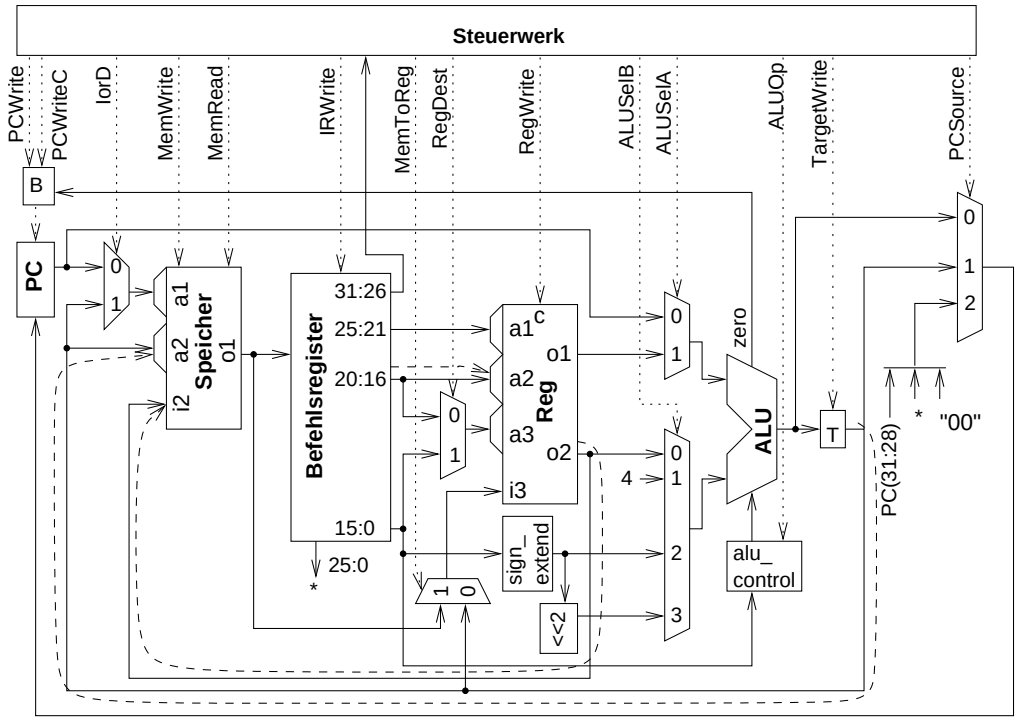
Im Falle eines *load*-Befehls folgt auf die Adressrechnung das Laden vom Speicher. Abb. 2.26 zeigt die benutzten Datenwege.



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

Abbildung 2.26: Benutzung der Struktur beim load-Befehl

Im Falle eines *store*-Befehls folgt auf die Adressrechnung das Schreiben des Speichers. Abb. 2.27 zeigt die benutzten Datenwege.

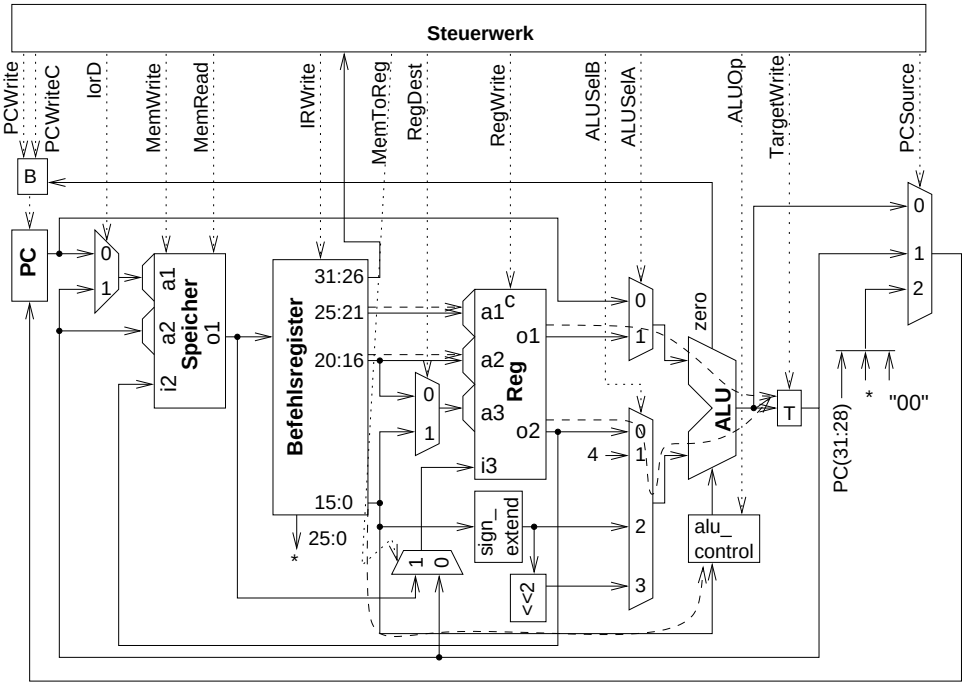


Steuerwerk: Ausgaben: MemWrite='1', für alle anderen Register: xxWrite='0'; Folgezustand = fetch

Abbildung 2.27: Benutzung der Struktur beim store-Befehl

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. 2.28.



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

Abbildung 2.28: Benutzung der Struktur im Zustand rr1

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. 2.29.

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. 2.30.

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

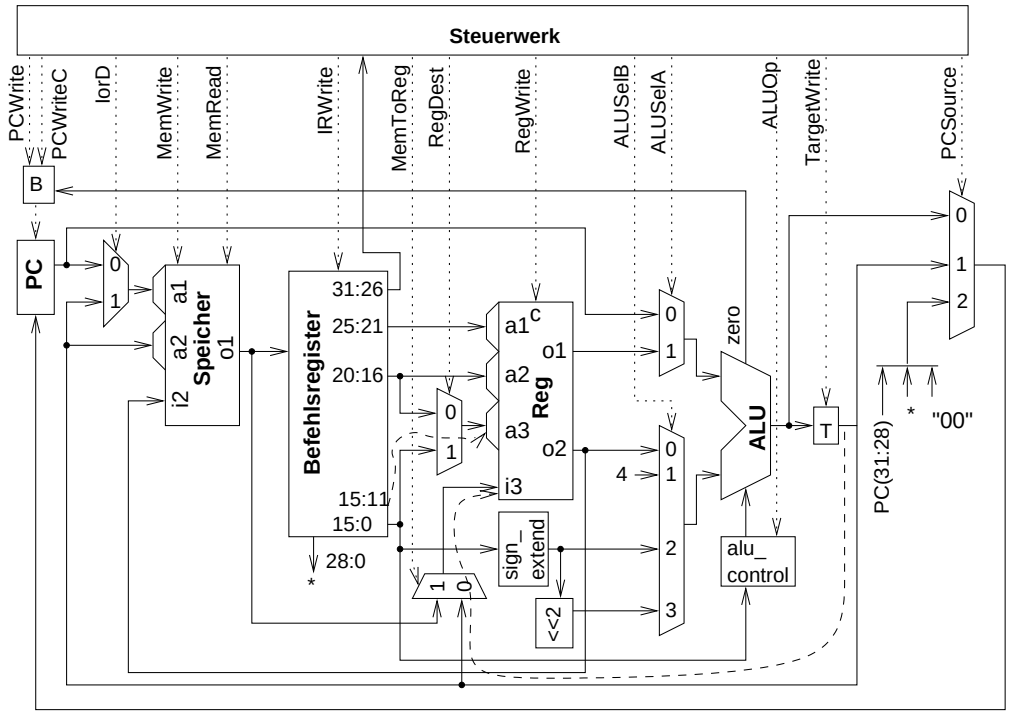
Als letzte Befehlsgruppe betrachten wir den BEQ-Befehl. Für diesen wird in einem ersten Zustand zunächst einmal das Verzweigungsziel ausgerechnet (Abb. 2.31).

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 2.32).

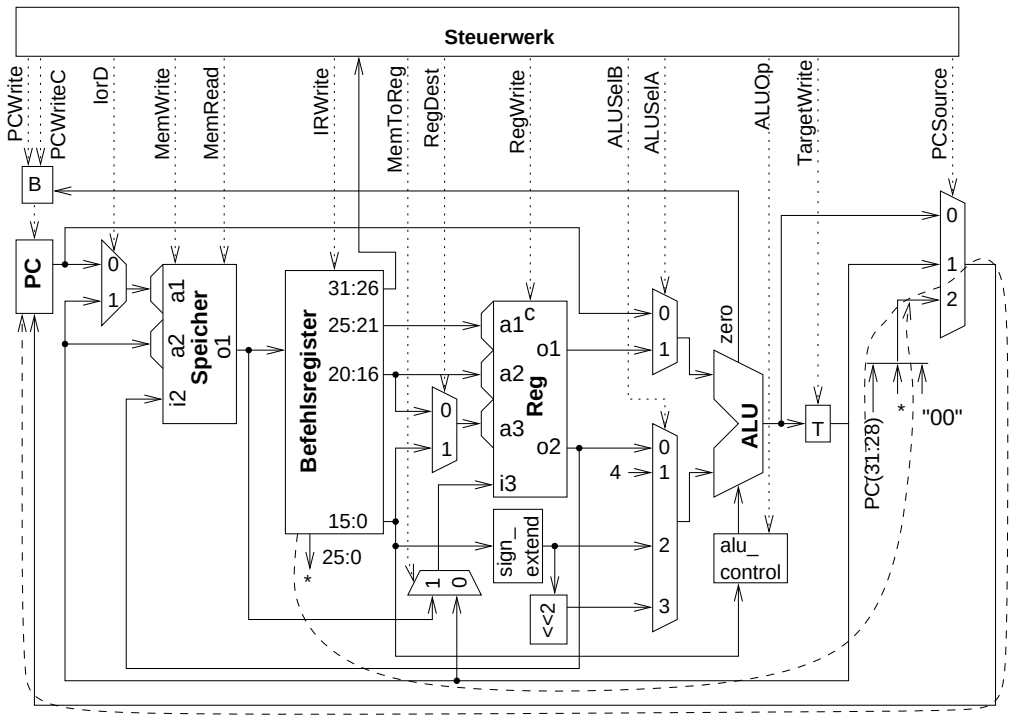
Der Befehl BNE wurde hier nicht betrachtet. Für ihn wäre die Bedeutung der Leitung zero zu komplementieren.

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.



Steuerwerk: Ausgaben: MemToReg=0, RegWrite='1', sonst: xxWrite='0'; Folgezustand = fetch

Abbildung 2.29: Benutzung der Struktur im Zustand rr2

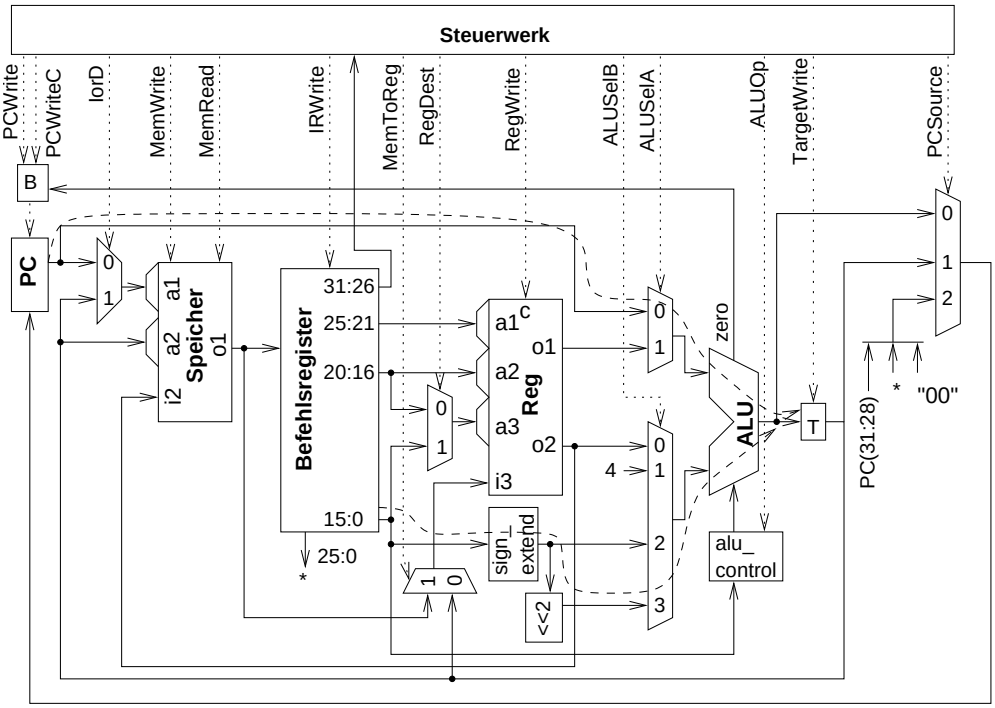


Steuerwerk: Ausgaben: PCSource=2, PCWrite='1', andere: xxWrite='0'; Folgezustand = fetch

Abbildung 2.30: Benutzung der Struktur beim Sprüngen

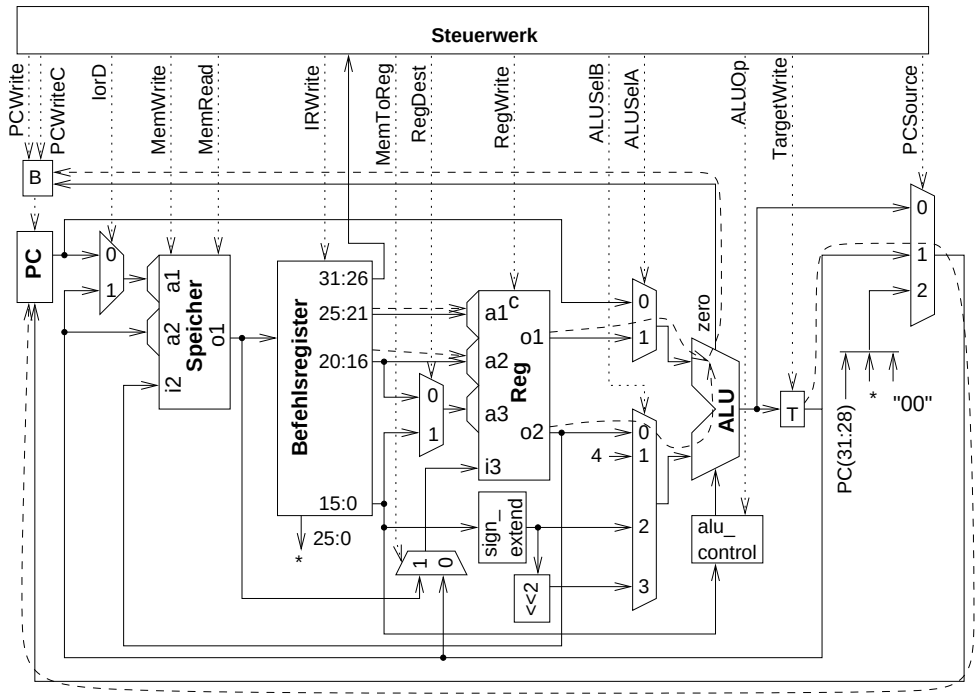
Aus den für die einzelnen Zustände angegebenen Ausgaben und Folgezuständen des Controllers kann jetzt die vollständige Funktion des Controllers bestimmt werden (siehe Tabelle 2.13).

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**. Mikropro-



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

Abbildung 2.31: 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 2.32: Benutzung der Struktur bei Verzweigungen

grammspeicher können als reine Lesespeicher (ROM) oder wiederbeschreibbar (als RAM) ausgeführt werden. Die Mikroprogramm-Speicher ist Teil der Struktur des Steuerwerks, das in Abb. 2.14 gezeigt wird.

Der aktuelle Zustand des Steuerwerks wird jeweils im so genannten Mikroprogrammzähler μPC gespeichert. Jedem aktuellen Zustand werden über den Mikroprogrammspeicher die Kontrollsignale des Rechenwerks zugeordnet. Der

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 2.13: Automatentabelle des Steuerwerks

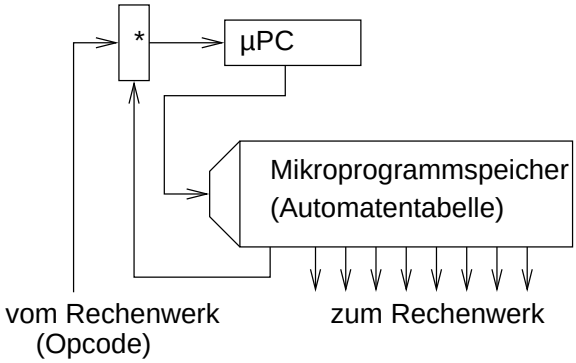


Tabelle 2.14: Struktur des Steuerwerks

mit * bezeichnete Baustein bestimmt aus den Informationen des aktuellen Zustands und der Informationen vom Rechenwerk den jeweils nächsten Zustand. Im Wesentlichen wird dabei der im Mikroprogrammspeicher gespeicherte Folgezustand übernommen. Nur beim Dekode-Zustand ist der Opcode maßgebend. Eine zusätzliche Spalte im Mikroprogramm-Speicher enthält dazu Information über die Art der Bildung der Folgeadresse. Alternativ kann man über diese Spalte auswählen, ob der Inhalt des Mikroprogrammspeichers den Folgezustand bestimmt oder ob dieses über die Signale aus dem Rechenwerk (wie dem Opcode) passiert. Eine Verzweigung im Zustand mar kann man relativ leicht vermeiden, in dem man den Zustand mar für load- und store-Befehle dupliziert.

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, recht 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.

2.3.3 Fließbandverarbeitung

2.3.3.1 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. 2.33), dass überwiegend mit jedem Takt die Bearbeitung eines Befehls beendet wird.

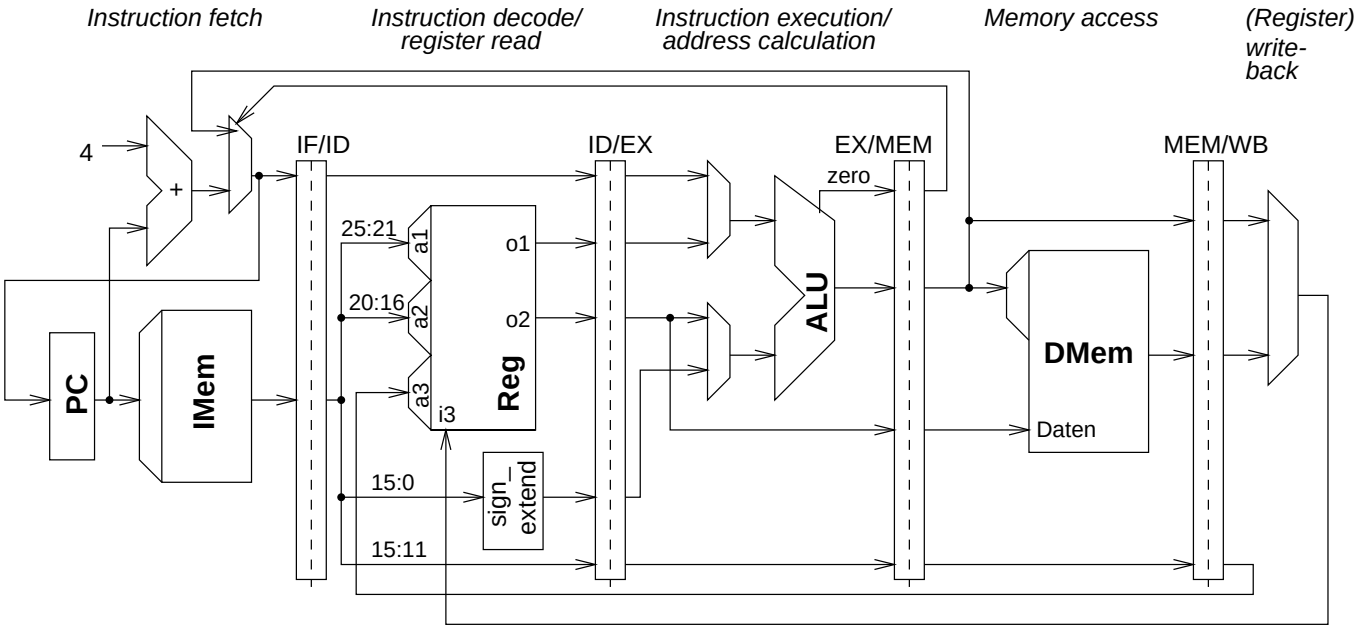


Abbildung 2.33: Fließband-Stufen

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 kennen lernen, 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 *write-back*-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 2.34.

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 2.34: 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.

2.3.3.2 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*.

2.3.3.3 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 $12,$2,$3
SUB $4,$5,$12
AND $6,$12,$7
OR  $8,$12,$9
XOR $10,$12,$11
```

Die Befehle `SUB, AND, OR` und `XOR` sind wegen der in `$12` 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 `R12` 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. 2.35.

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

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 2.35: Datenabhängigkeiten

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.

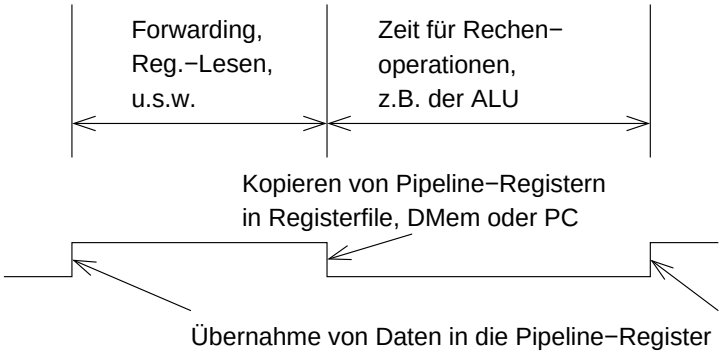


Abbildung 2.36: Taktung des Fließbands

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. 2.37, 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
AND $6,$1,$7
OR  $8,$1,$9
```

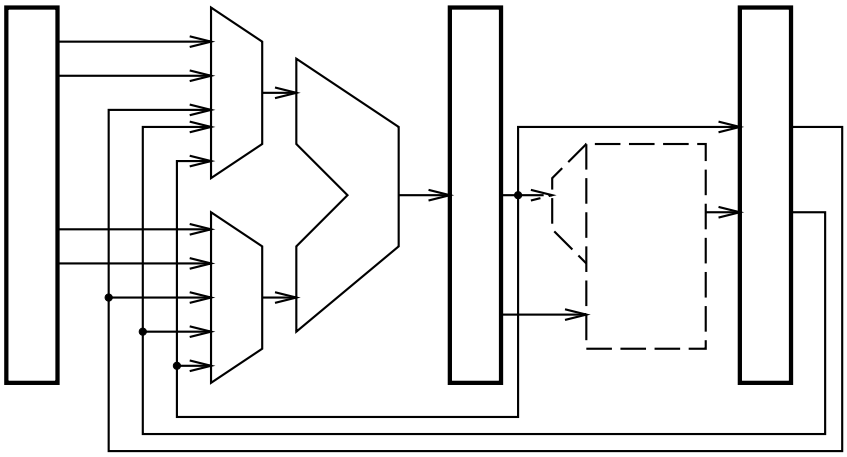


Abbildung 2.37: Bypass-Logik an den Eingängen der ALU

Abb. 2.38 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 2.38: 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* [HP07] im Fließband. Die Einträge NOOP in der Abb. 2.39 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 2.39: Anhalten des Fließbands

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

2.3.3.4 Kontrollfluss-Abhängigkeiten

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

```
BEQ  $12,$2 t --branch to t if equal
```

```
      SUB  ...
      ....
t:    ADD
```

Abb. 2.40 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 zwei 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 2.40: 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. 2.41 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.

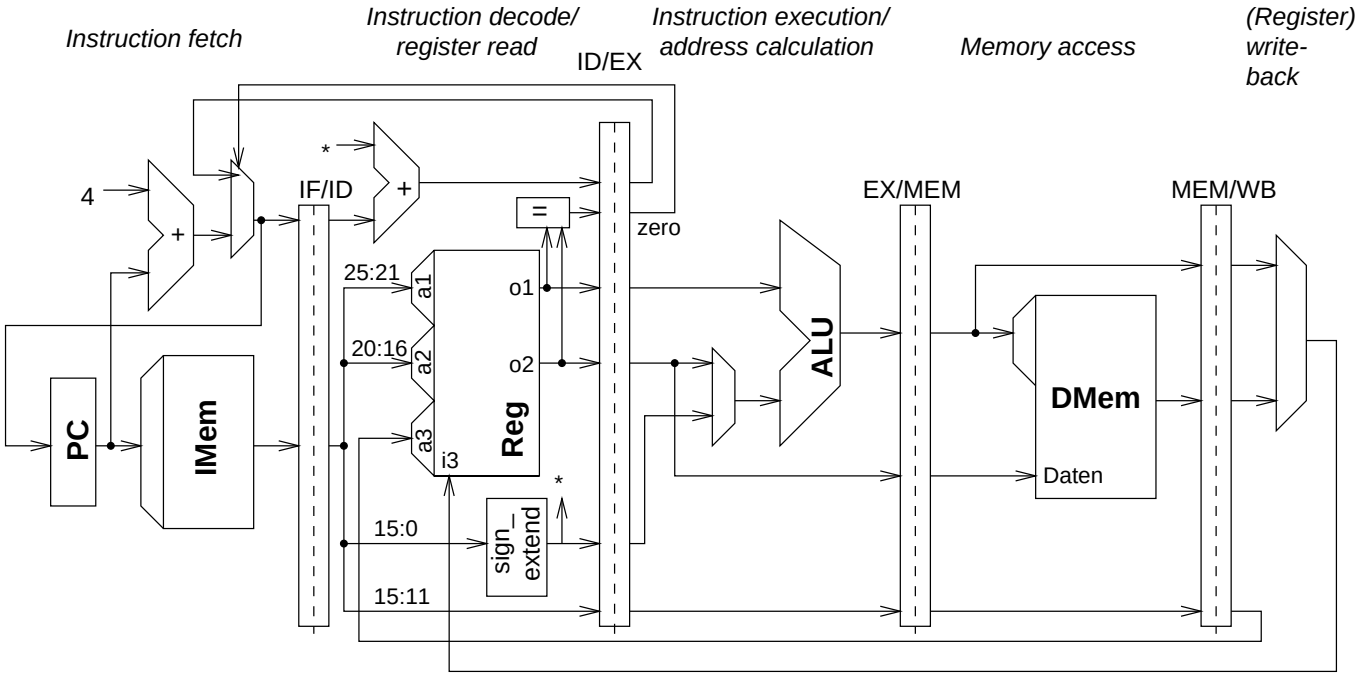


Abbildung 2.41: Fließband-Stufen

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. 2.42).

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,

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 2.42: Delayed Branch

wenn der Compiler die *slots* mit Befehlen füllt, die sinnvolle Berechnungen ausführen. Dafür kommen v.a. Befehle in Frage, die sonst **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.

2.3.4 Trends in der Prozessorentwicklung

2.3.4.1 Interne Struktur von Pentium-Prozessoren

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. 2.43). Dies ist zwar keine sehr effiziente Verwendung des Siliziums, aber darauf kommt es bei PC-Applikationen nicht an.

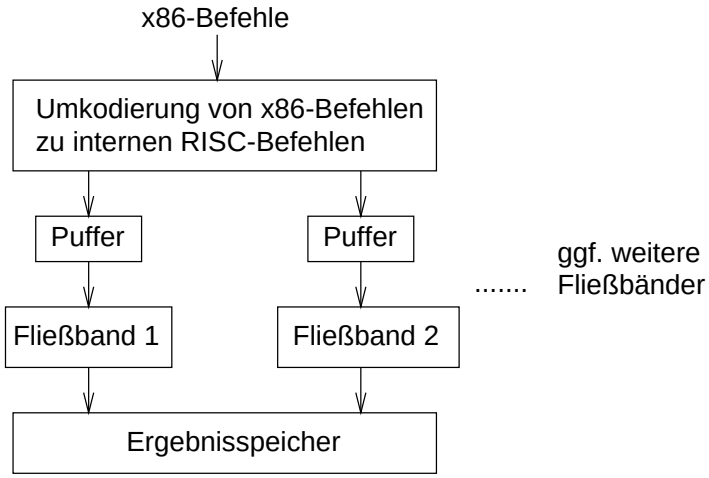


Abbildung 2.43: Interne Struktur von Pentium-Prozessoren

2.3.4.2 Trends der Entwicklung energieeffizienter Prozessoren

Neben den von uns bislang betrachteten Befehlsschnittstellen sind weitere spezielle Schnittstellen und Prozessorklassen im Gebrauch. Es sind dies u.a.

- Digitale Signalprozessoren (DSP): Prozessoren, die auf die Verarbeitung digitaler Signale (Sprache, Video) optimiert sind
- *Very long instruction word* (VLIW) Prozessoren: Prozessoren, die mit breiten Paketen von Befehlen mehrere Befehle gleichzeitig starten können
- *application specific instruction set processors* (ASIPs): Prozessoren, die für bestimmte Anwendungen (z.B. MPEG) optimiert sind
- *Field programmable gate arrays* (FPGAs): Schaltungen, deren Verhalten man durch Programmierung verändern kann.
- *Application specific integrated circuits* (ASICs): speziell für eine Anwendung entwickelte integrierte Schaltkreise

Aufgrund von Schwierigkeiten, Prozessoren zu kühlen und aufgrund von nur sehr begrenzt verfügbarer elektrischer Energie (insbesondere bei mobilen Geräten) ist die Energieeffizienz zu einem sehr wichtigen Kriterium bei der Entwicklung neuer Prozessoren geworden. Die Enegieeffizienz von betreffenden Prozessor- und Hardwareklassen kann man auf der Basis der pro Energieeinheit ausführbaren Anzahl von Operationen (Milliarden Operationen pro Joule) vergleichen (siehe Abb. 2.44).

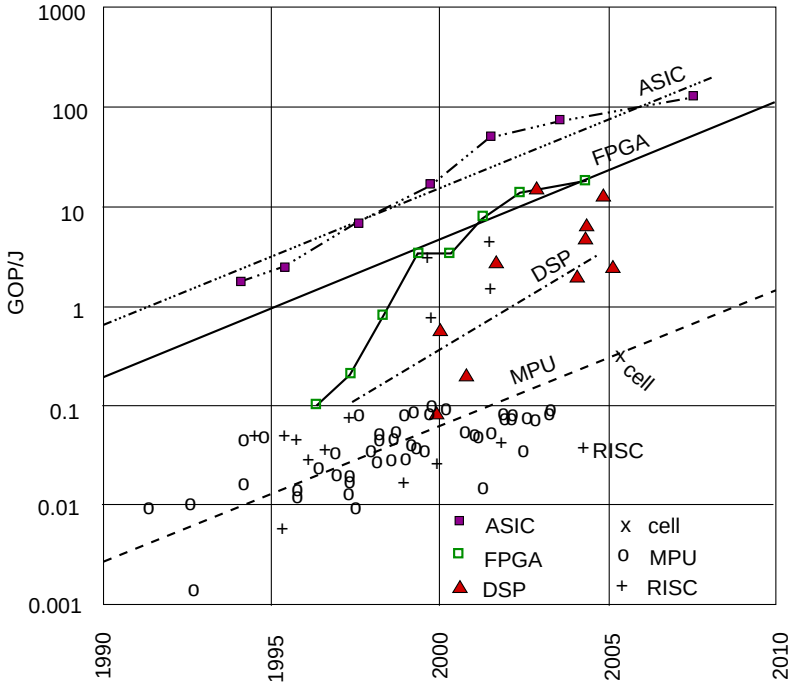


Abbildung 2.44: Energieeffizienz von Ausführungsplattformen (©De Man and Philips)

Dabei stellt sich heraus, dass die Energieeffizienz sich generell im Laufe der Jahre aufgrund der fortschreitenden Miniaturisierung innerhalb von Chips verbessert hat. Für eine bestimmte Fertigungstechnik bieten klassische Prozessoren die geringste Energieeffizienz. Mit DSPs ist diese größer, mit FPGAs noch größer und mit ASICs am größten. Mit ASIP kommt man der Effizienz von ASICs nahe.

Grundsätzlich ist es empfehlenswert, eine hohe Performance nicht über eine schnelle Taktung seriell ausgeführter einzelner Operationen zu realisieren. Hierfür wäre nämlich eine relative hohe Betriebsspannung erforderlich, was wiederum eine hohe Energieaufnahme bewirken würde. Statt dessen ist es besser, einen langsameren Takt vorzusehen und während jedes Taktes Operationen parallel auszuführen. Formal kann man diesen Ansatz wie folgt begründen:

Die bei üblicher CMOS-Chiptechnik in Wärmeleistung umgewandelte elektrische Leistung ist näherungsweise quadratisch von der Betriebsspannung abhängig:

$$P \sim V_{DD}^2 \tag{2.7}$$

Das quadratische Verhalten kann man damit erklären, dass die Leistung sich aus dem Produkt von Strom und Spannung ergibt und dass der Strom wiederum proportional zur Spannung sein kann.

Die maximale Taktfrequenz von CMOS-Schaltungen wächst näherungsweise linear mit der Betriebsspannung:

$$f \sim V_{DD} \tag{2.8}$$

Der Energiebedarf für ein sequentiell ausgeführtes Programm mit fester Laufzeit $t \sim \frac{1}{f}$ und (als konstant angenommener) Leistung P sei

$$E = P \cdot t \tag{2.9}$$

Führen wir jetzt β Operationen parallel aus, so haben wir für jede Operation β-mal mehr Zeit und können das Programm dennoch in derselben Zeit t abschließen. Da mehr Zeit zur Verfügung steht, ist es möglich, die Taktfrequenz auf die neue Taktfrequenz

$$f' = \frac{f}{\beta} \tag{2.10}$$

abzusenken und dementsprechend auch die Spannung:

$$V'_{DD} = \frac{V_{DD}}{\beta} \tag{2.11}$$

Durch die Spannungsreduktion reduziert sich die Leistung für eine einzelne Operation auf

$$P^+ = \frac{P}{\beta^2} \tag{2.12}$$

Bei paralleler Ausführung von β Operationen ergibt sich eine Gesamtleistung von

$$P' = \beta \cdot P^+ = \frac{P}{\beta} \tag{2.13}$$

Die Zeit für die Ausführung des Programms ist unverändert, d.h. die neue Laufzeit t' ist gleich der alten t. Die Energie für die Ausführung des Programms ist damit

$$E' = P' \cdot t = \frac{E}{\beta} \tag{2.14}$$

Diese (näherungsweise) Rechnung zeigt, dass die Parallelausführung aus energetischen Gründen der schnellen sequentiellen Ausführung vorzuziehen ist[16].

Dieses Argument spricht grundsätzlich für die Nutzung von VLIW-Prozessoren und von parallel rechnenden klassischen Mikroprozessoren in Form von so genannten Multicores. Dementsprechend ist die Steigerung der Taktrate von klassischen Prozessoren etwa im Jahr 2003 weitgehend gebremst worden und für die Zukunft werden nur noch geringe Steigerungen der Taktraten erwartet. Eine Steigerung der Performance gelingt damit überwiegend nur noch über eine Erhöhung der Anzahl gleichzeitig arbeitender Prozessoren. Diesen Trend zu Multiprozessorsystemen kann man auch bei den üblichen Mikroprozessoren beobachten.

In der Praxis können sich in einem Rechner mehrere Programme gleichzeitig im Speicher befinden, die jeweils mit einer Ausführungsumgebung zu **Prozessen** oder *threads* ergänzt werden. Diesen wird jeweils durch einen *dispatcher* Rechenzeit zugeteilt. Der Dispatcher schaltet zwischen der Ausführung von Prozessen um (*context switch*). Bei einem *context switch* werden alle Registerinhalte des anzuhaltenden Prozesses in einem ihm zugeordneten Datenblock *process control block (PCB)* gerettet und die des fortzuführenden Prozesses aus seinem PCB geladen. Abgesehen vom Zeitverhalten wird den Prozessen suggeriert, der Prozessor gehöre ihnen allein. Dabei unterscheiden wir zwischen Prozessen und *threads*. Bei Prozessen wird beim Kontextwechsel auch der dem Prozess sichtbare Speicherbereich verändert, wohingegen alle Threads, die zu einem Prozess gehören, denselben Speicher sehen können. Die Umschaltung zwischen *threads* geschieht dadurch schneller als die zwischen Prozessen. Weitere Informationen zur Verwaltung von Prozessen sind Teil der Vorlesung „Betriebssysteme und verteilte Systeme".

Bei modernen Prozessoren nutzt man die verschachtelte Ausführung von mehreren *threads*, um Zugriffszeit auf den Speicher (Latenzzeit) zu verstecken. Man schaltet sehr schnell (häufig: nach jedem Takt) zwischen der Ausführung von verschiedenen *threads* um. Während der Speicher noch beschäftigt ist, Informationen für den ersten *thread* zu holen, wird bereits der zweite ausgeführt, der seinerseits dann wieder Informationen aus dem Speicher anfordern kann. Übliche Mikroprozessoren sehen dieses so genannte Multithreading derzeit für zwei ineinander geschachtelt ausgeführte *threads* vor. In Kombination mit Multicore-Systemen ergibt sich so eine steigende Anzahl von typischerweise verfügbaren Cores (Prozessoren). Experimentell gibt es beispielsweise Intel's *single chip cloud computer* (SCC) dessen 24 physikalische Prozessoren sich aufgrund eines 2-fachen Multithreadings wie 48 logische Cores verhalten.

Bei der parallelen Verarbeitung klassifiziert man Prozessorsysteme häufig gemäß der Klassifikation von Flynn nach der Anzahl der Daten- und der Befehlsströme. Im Fall von einem Daten- und einem Befehlsstrom spricht man von SISD (*single instruction, single data*)-Maschinen. Dies sind die uns bislang vertrauten Maschinen bzw. Prozessoren. Betreibt man mehrere davon parallel, so kommt man zu MIMD (*multiple instruction, multiple data*)-Maschinen. Wendet man jeweils denselben Befehl auf unterschiedliche Daten an, so spricht man von SIMD (*single instruction, multiple data*)-Maschinen. Solche Maschinen sind sehr sinnvoll, wenn wie beispielsweise in der Bildverarbeitung auf die unterschiedlichen Daten (Pixel) derselbe Befehl (zum Beispiel zur Filterung) angewandt werden soll. SIMD-Maschinen sind häufig energieeffizient, weil man gegenüber dem Fall der SISD- oder MIMD-Maschinen Aufwand für das Holen von Befehlen spart. Die Energie-effizientesten unten den verfügbaren Prozessoren sind häufig SIMD-Prozessoren. Die Klasse MISD (*multiple instruction, single data*)-Maschinen ist nur mit Mühe zu füllen und wird meist als nicht vorhanden beschrieben.

Bei den Multicores wird vielfach die Frage diskutiert, ob die verschiedenen Cores alle von derselben Art sein sollen oder nicht. Im ersten Fall spricht man von **homogenen** Multicore-Systemen, im zweiten Fall von **heterogenen** Multicore-Systemen. Die verschiedenen Argumente zeigt die Abbildung 2.45.

	homogene Prozessoren	identische ISA, unterschiedl. Implementierung	heterogene Prozessoren
Leichte Verlagerung von Prozessen	+	+	-
Fehlertoleranz durch Verlagerung	+	+	-
Effizienz durch Optimierung für Anwendung(-sklassen)	-	((+))	+
Software-Entwurfsaufwand	+	(+)	-
Prozessor-Entwurfsaufwand	+	$\pm$	-

Abbildung 2.45: Argumente für (+) und gegen (-) bestimmte Prozessoren

[16] Laut persönlicher Kommunikation anläßlich eines Besuchs bei der Fa. IBM in Böblingen wurde kommuniziert, dass für die dort verwendete Technologie eher ein kubisches Verhalten beobachtet werden kann. Bei einem solchen Verhalten würden sich die Argumente für eine parallele Ausführung noch einmal verstärken.

Bei homogenen Multicore-Systemen kann zur Laufzeit Software relativ leicht auf einen anderen Core verlagert werden. Auch ist der Entwurfsaufwand relativ gering, da nur ein Prozessortyp entwickelt wird. Bei heterogenen Multicore-Systemen können einzelne Cores an bestimmte Aufgaben optimal angepasst werden, zum Beispiel an die Kompression von Bildern und die Dekodierung bei Telephonieanwendungen. Es zeigt sich, dass man damit die beste Energieeffizienz erreicht. Ein interessanter Kompromiss ist die big.LITTLE-Architektur der Fa. ARM. Hierbei haben alle Prozessoren denselben Befehlssatz (ISA), sind aber teilweise auf beste Energieeinsparung und teilweise auf maximale Rechenleistung hin entworfen. Beim Samsung Exynos 5 Octa Multicore (benutzt im Samsung Galaxy 4) gibt es jeweils vier Prozessoren von jeder Sorte, von denen aber immer nur max. vier in Betrieb sind. Ist das System wenig beschäftigt, sind dies die vier energieeffizienten Prozessoren. Logisch verhält sich dieses Multicore-System wie ein Vierkern-System. Zusätzlich sieht man vielfach spezielle Graphik-Coprozessoren vor, die bestimmte Aufgaben sehr effizient ausführen können. In der Großrechnerwelt sind dagegen eher homogene Prozessoren üblich, da hier die Verlagerung von Anwendungen wichtig und die Energieeinsparung weniger sind.

In der Zukunft erwartet man eine weitere Steigerung der Anzahl der Cores. Allerdings taucht die Frage auf, ob man überhaupt alle Prozessoren eines Chips gleichzeitig mit Strom versorgen kann. Problematisch ist die dabei enstehende thermische Energie, die abgeführt werden muss. Es zeigt sich, dass dies nicht mehr in jedem Fall möglich ist. Bereiche auf dem Chip, die in einer bestimmten Phase nicht mit Strom versorgt werden, nennt man *dark silicon*, weil sie praktisch nicht mehr “leuchten”. Die Tatsache, dass man künftig evtl. große Teile eines Chips aus thermischen Gründen nicht mehr mit Strom versorgen kann, hat weitreichende Konsequenzen, bis hin zu der Frage, ob künftige komplexe Halbleiterfabriken noch ökonomisch zu betreiben sind.

Aus technischen Gründen sind Multicore-Systeme entstanden. Die Erzeugung entsprechend zeitlich überlappt ausgeführter Programme stellt noch ein Problem dar. Fraglich ist, ob es Techniken geben wird, die derartige Programme effizient erzeugen können.

2.4 Speicherarchitektur

"*Die Wissenschaft Informatik befasst sich mit der Darstellung, Speicherung, Übertragung und Verarbeitung von Information*" [Ges06]. In diesem Kapitel geht es um die Speicherung. Dementsprechend ist diese in der Abbildung 2.46 hervorgehoben.

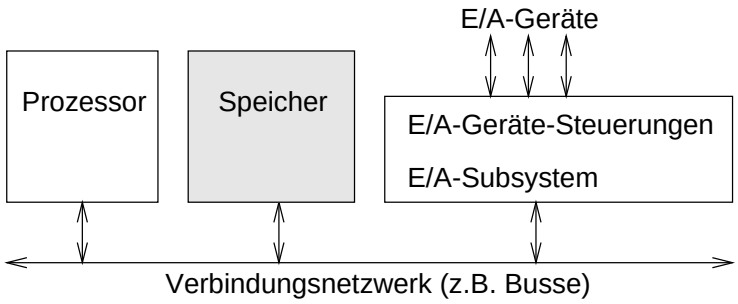


Abbildung 2.46: Speicherarchitektur

2.4.1 Speicherhierarchie

Beim Entwurf des Speichers gibt es viele Entwurfsziele zu beachten: wir wollen einen Speicher einer möglichst großen Kapazität mit möglichst kleinen Zugriffszeiten (sowohl im Sinne der **Latenz**, d.h. der Zugriffszeit auf das erste Wort, wie auch im Sinne des guten **Durchsatzes** (engl. *throughput*), d.h. eines kleinen Abstands zwischen aufeinanderfolgenden Worten). Die Speicherung sollte möglichst auch nach dem Ausschalten des Stromes erhalten bleiben, d.h. **persistent** und insgesamt eine hohe Datensicherheit bieten sein. Der Speicher sollte eine **geringe Energieaufnahme** haben und auch physikalisch **kompakt** sein. Leider sind diese Ziele teilweise nicht miteinander vereinbar oder nur näherungsweise zu erreichen.

Einen ersten Konflikt können wir am Beispiel der Kapazität, der Zugriffszeit und der Energieaufnahme ansehen. Es ist ein wesentliches Merkmal von Speichern, dass kleine Speicher schneller sind und pro Zugriff weniger Energie benötigen als große Speicher (siehe Abb. 2.47 [Mar10]). Hierfür gibt es zahlreiche technologische Gründe, wie z.B. Leitungen, die bei kleinen Speichern kürzer sind als bei großen.

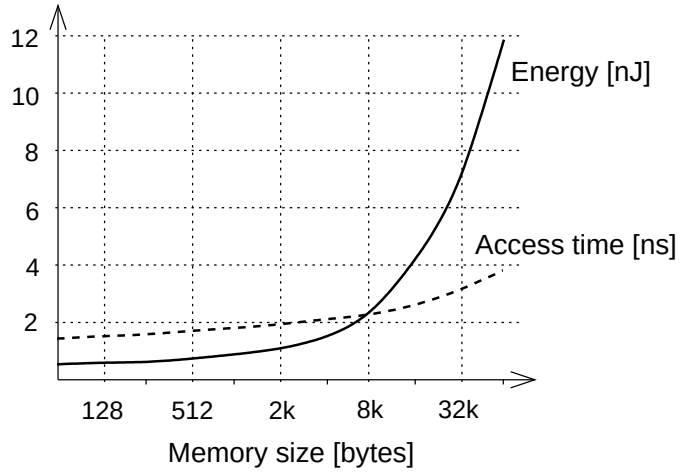


Abbildung 2.47: Abhängigkeit der Zugriffszeit und des Energiebedarfs von der Speichergröße

Leider verkürzen sich die Zugriffszeiten bei Speichern auch nur langsam. Potentiell wird damit die *Performance* von Systemen potentiell mehr und mehr durch den Speicher bestimmt. Der Speicher entwickelt sich also zum Flaschenhals. Man spricht deshalb auch von der so genannten "*memory wall*", der Wand gegen die man läuft, wenn man die *Performance* eines Systems erhöhen will.

Glücklicherweise besitzen viele Anwendungen ein anderes Merkmal: Anwendungen weisen üblicherweise eine gewisse **Lokalität der Speicherzugriffe** auf, es wird also nicht völlig zufällig auf den Speicher zugegriffen.

Diese beiden Merkmale versucht man nun üblicherweise in Form einer **Speicherhierarchie** auszunutzen: Man versucht, häufig benutzte Speicherinhalte in kleinen schnellen Speicher zu halten, wohingegen seltener benutzte Informationen in langsameren Speichern abgelegt werden. Ziel einer solchen Kombination von Speichern ist es, den Speicher insgesamt wie einen großen Speicher mit schnellen Zugriffen aussehen zu lassen. Diese Idee ist nicht neu. Vielmehr schreiben Burks, Goldstine und von Neumann 1946 [BGN46]:

"*Ideally one would desire an indefinitely large memory capacity such that any particular ... word ... would be immediately available - i.e. in a time which is ... shorter than the operation time of a fast electronic multiplier. ... It does not seem possible physically to achieve such a capacity. We are therefore forced to recognize the possibility of constructing a hierarchy of memories, each of which has greater capacity than the preceding but which is less quickly accessible*." .

Die Anzahl der Stufen wird dabei bestimmt durch die Unterschiede in der Zugriffsgeschwindigkeit zwischen der schnellsten und der langsamsten Stufe. Als Regel gilt: umso größer die Unterschiede, umso mehr Stufen sind sinnvoll. Beim aktuellen Stand können wir als Stufen der Hierarchie unterscheiden zwischen den Prozessorregistern, dem *Cache* (bzw. dem cache-ähnlichen TLB), dem Primär- oder Hauptspeicher, dem Sekundärspeicher und ggf. dem Tertiärspeicher. Den bislang betrachteten Speicher bezeichnen wir dabei als Haupt- oder Primärspeicher.

Dabei muss natürlich für jede Stufe auch eine passende Technologie zur Verfügung stehen. Eine technische Realisierung kann entweder eine dauerhafte Speicherung oder nur eine flüchtige Speicherung bieten. Erstere wird auch eine **persistente** Speicherung genannt, die zweite **volatile** Speicherung. Von einer dauerhaften Speicherung wird erwartet, dass die Informationen auch nach Ausschalten des Gerätes und Fortfall der Stromversorgung erhalten bleiben. Langfristig zu speichernde Informationen müssen natürlich persistent gespeichert werden. Derzeit (2013) sind als Hauptspeicher nutzbare Technologien (von Sonderfällen wie sehr langsamen Geräten abgesehen) in der Regel flüchtig. Als Speichertechnologien stehen derzeit v.a. statische RAMs (SRAMs), dynamische RAMs (DRAMs), Flash-Speicher, Magnetplatten, optische Speicher und Magnetbänder zur Verfügung. Es ist eine inhärente Eigenschaft aktuell verfügbarer Technologie, dass schnelle Speicher in der Regel flüchtig sind. Persistenter Flashspeicher weicht allerdings manche der bisherigen Grenzen auf. Auch gibt es Entwicklungen wie so genannte MRAMs, die eine persistente Speicherung mit den Zugriffszeiten von bisherigen Hauptspeichern erlauben könnten und die somit die bisherigen Strukturen massiv verändern könnten. Abbildung 2.48 zeigt eine Übersicht über verfügbare Speichertechnologien.

Name	Zugriffszeit	Durchsatz	Persistent?	Kosten/GB
SRAM	$<$ 1ns		nein	
DRAM	10..100 ns		nein	1..10 $/GB
Flash-Speicher	1..100 μs	$\sim$ 500 MB/s	ja	0,1..1 $/GB
Magnetplatten	1..100 ms	$<$ 300 MB/s	ja	0,01..0,1 $/GB
Optische Speicher			ja	$\sim$ 0,1 $/GB
Magnetbänder	1..100 s		ja	

Abbildung 2.48: Speichertechnologien (Ca.-Daten von 2013)

Abb. 2.49 zeigt die zur Zeit mögliche Zuordnung zwischen Hierarchieebenen und Technologien.

Name	SRAM	DRAM	Flash	Magnetplatte	Optische Speicher	Magnetbänder
Register	X					
Cache, TLB	X	((X))*				
Primärspeicher		X	((X))			
Plattencache		X	(X)			
Sekundärspeicher			X	X		
Tertiärspeicher					X	X

Abbildung 2.49: Hierarchieebenen und Technologien, *: bei langsamen Cacheebenen

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.

2.4.2 Primär- bzw. Hauptspeicherverwaltung

Wir beschäftigen uns zunächst mit dem Primär- bzw. Hauptspeicher. Wir verstehen darunter einen Speicher, dessen Modell zur Beschreibung der Bedeutung des Maschinenbefehlssatzes herangezogen wird. Wir erwarten eine Adressierbarkeit des Speichers durch die Adressenteile der Maschinenbefehle.

Ein Schlüssel für die Einführung der Speicherhierarchie ist die Unterscheidung zwischen Maschinen- und Prozessadressen gemäß nachfolgender Definition:

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[17].

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

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).

2.4.2.1 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 **Systeme ohne Speicherabstraktion** bezeichnet [Tan09]. 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 (siehe Abb. 2.4).

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. 2.50).

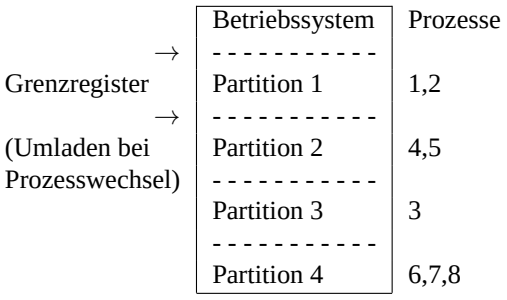


Abbildung 2.50: Aufteilung des Speichers in Core-Partitionen

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

Eine Ausdehnung über Partitionsgrenzen hinaus (siehe Abb. 2.51) ist meist nicht realisiert.

Eigenschaften dieser Organisationsform sind die folgenden:

- Es ist kein **Verschieben** (siehe Abb. 2.52) 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.

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

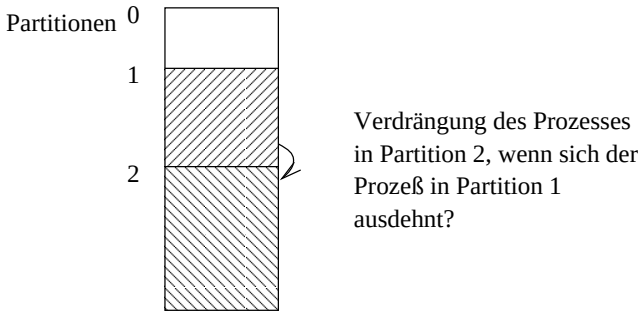


Abbildung 2.51: Zur Ausdehnung über Partitionsgrenzen

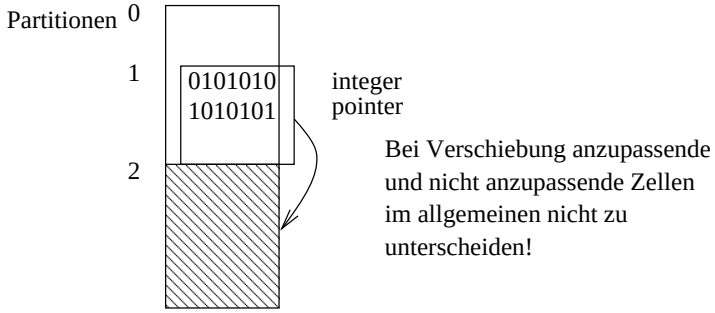


Abbildung 2.52: Zum Verschieben von Partitionen

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

- 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.

- Eine Verwendung der Adressen wie in Abb. 2.3 wäre sehr unpraktisch, da der physikalische Speicher genau die in Abb. 2.3 vorkommenden Adressbereiche realisieren müsste, also in eine Reihe von Segmenten zu partitionieren wäre.

Diese Form der Speicherorganisation ist durch eine starke so genannte **externe Fragmentierung** gekennzeichnet. Dies bedeutet, dass außerhalb des Speichers, der den Prozessen zugeteilt ist, viel Speicher ungenutzt bleiben kann. Man muss sich relativ genau überlegen, welcher Speicherbereich wie groß werden kann. Wird den Prozesen zu wenig Speicher zugeteilt, so besteht das Risiko des Prozessabbruchs, wenn der zugeteilte Speicher nicht ausreicht. Außerdem kann der Adressraum nie größer werden als der tatsächlich vorhandene Speicher. Mit komplexeren Techniken gelingt es dagegen, mehr Speicher zu adressieren, als tatsächlich physikalisch als Hauptspeicher vorhanden ist. Diese Techniken werden wir nachfolgend kennenlernen.

2.4.2.2 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** (engl. *pages*). Weiter erfolgt eine Einteilung des realen Adressraumes in gleich große Bereiche. Diese Bereiche heißen **Seitenrahmen** (engl. *frames*) 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. 2.53.

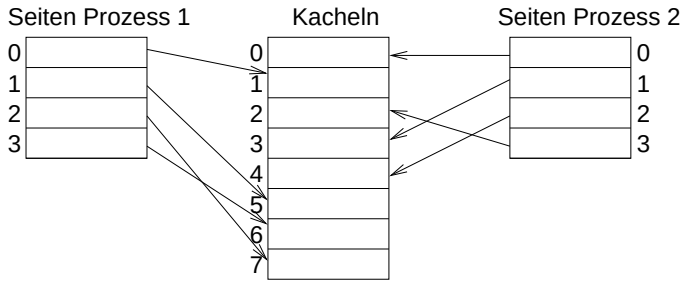


Abbildung 2.53: Zuordnung von Seiten zu Kacheln bei der Seitenadressierung

Jeder Prozess benutzt dabei virtuelle Adressen, die von den virtuellen Adressen anderer Prozesse zunächst einmal als unabhängig angesehen werden können. Die Menge der virtuellen Adressen eines Prozesses heißt dessen **virtueller Adressraum**.

Der große Vorteil der Seitenadressierung ist, dass jede Kachel jede Seite aufnehmen kann. Zu jedem Prozess muss zu jedem Ausführungszeitpunkt eine Abbildung **Seitennummern** $\rightarrow$ **Kachelnummern** definiert sein, mit Hilfe derer die reale Adresse gefunden werden kann. Bei der Neuzuordnung von Seiten kann sich diese Abbildung ändern. Die Abbildung von Seiten auf Kacheln kann dabei dynamisch sein. Man muss sich damit nicht mehr vorab überlegen, wie groß die einzelnen Speicherbereiche (wie z.B. *stack, heap, text*) werden können. Jeder Bereich des Speichers kann genutzt werden (keine **externe Fragmentierung**).

Lücken im virtuellen Adressraum (wie beim MARS) sind kein Problem bei der Speichernutzung. Die vorgestellte Seitenadressierung wäre beispielsweise ausreichend, um die Speicherzuordnung wie in Abb. 2.3 praktikabel werden zu lassen (siehe Abb. 2.54). Über die Seitentabellen kann man dafür sorgen, dass die verschiedenen Segmente auf einen einzigen Speicher mit lückenlosem Adressbereich abgebildet werden. In Abb. 2.54 sind die Seiten innerhalb der Segmente durch gestrichelte Linien getrennt. Wächst ein Segment, so kann freier realer Speicher zugeordnet werden.

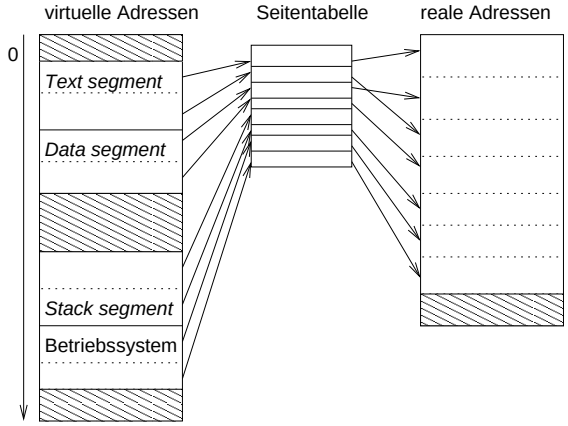


Abbildung 2.54: Abbildung der Speicherbereiche der Abb. 2.3 auf einen zusammenhängenden realen Speicherbereich mittels Paging

Die Seitenadressierung erledigt damit das so genannte Speicher-*Mapping*, d.h. die Abbildung der Speicherbereiche der Anwendung auf den realen Speicher.

Die virtuellen Adressen können unabhängig von den Adressen des vorhandenen Speichers gewählt werden und damit für jeden virtuellen Adressraum insbesondere bei Null beginnen. Compiler liefern typischerweise entsprechenden Code und dieser kann unmodifiziert in den Speicher geladen werden. Eine Adressanpassung beim Laden -wie dies bei der Identität der Fall war- ist nicht erforderlich. Eine Ausnahme davon ist das Binden unabhängig voneinander übersetzter Bibliotheken, die alle jeweils bei Null beginnen. Deren Adressen müssen beim Laden in einen gemeinsamen Adressraum abgebildet werden, die Adressen also entsprechend angepasst werden. Ein zeitraubendes Verschieben von Inhalten im Speicher ist nie erforderlich.

Ein gewisser Nachteil ist die so genannte **interne Fragmentierung** (d.h., Seiten sind nicht unbedingt komplett gefüllt).

Das Konzept der konkreten Umrechung von virtuellen Adressen auf reale zeigt die Abbildung 2.55.

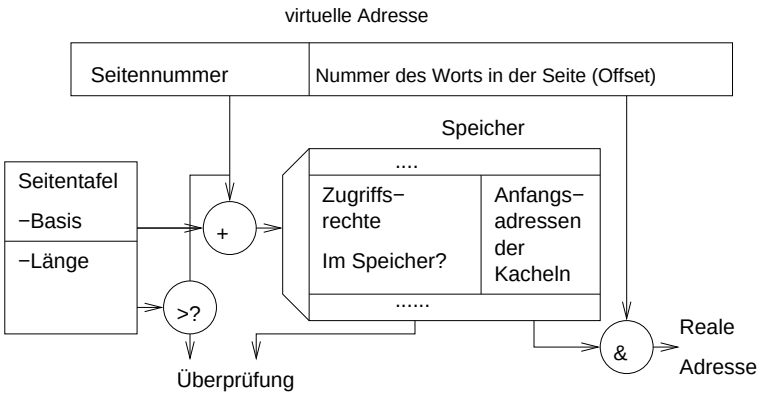


Abbildung 2.55: Konzept der Berechnung von realen Adressen

Die Seitennummer wird benutzt, um in der Seitentabelle die Anfangsadresse der Kachel zu finden, auf welche die Seite abgebildet ist. Der passende Eintrag in dieser Tabelle muss auf jeden Fall schnell gefunden werden, da die Umrechung bei jedem Maschinenbefehl mindestens einmal, bei Zugriff auf Speicherdaten auch mehrmals zu erfolgen hat. Grundsätzlich sind verschiedene Techniken zur Realisierung des schnellen Findens des Eintrags möglich. Abbildung 2.55 basiert auf der Annahme, dass die Seitentabelle zusammenhängend im Speicher angeordnet ist (dies ist zumindest für kleine Adressräume möglich). Ein Spezialregister kann dann den Anfang der Seitentabelle enthalten. Mittels Addition erhält man dann eine Adresse, mit der man direkt den Speicher adressieren und mit geringer Verzögerung die gesuchte Information erhalten kann. Üblicherweise ist die Kachelgröße eine Zweierpotenz und die Seitentabelle enthält daher nur die signifikantesten Bits der Anfangsadresse der Kachel. Konkateniert man diese Bits mit der Adresse innerhalb der Seite (dem *Offset*), so erhält man die vollständige reale Adresse.

Zusätzlich ist dabei zu überprüfen, ob die Seitennummer nicht größer ist als die Zahl der Einträge in der Seitentabelle. Zu diesem Zweck wird die Seitennummer verglichen mit der Zahl der Einträge in der Seitentabelle, die ebenfalls in einem Spezialregister gehalten wird.

Schließlich kann die Seitentabelle noch weitere Informationen über die Seiten enthalten. So können die Seiten mit **Zugriffsrechten** versehen werden. Diese beschreiben, in welcher Form ein Prozess auf eine Seite zugreifen kann. Damit kann verhindert werden, dass ein Prozess unberechtigt Informationen eines anderen Prozesses liest oder verändert. Man kann zwischen Schreib-, Lese- und Ausführungsrechten unterscheiden. Zwischen Lese- und Ausführungsrechten kann unterschieden werden, weil der Rechner immer „weiß“, ob er gerade einen Befehl oder Daten holen will. Auf diese Weise kann Software zur Ausführung bereitgestellt werden, ohne dass diese im Speicher gelesen werden kann.

Vielfach werden auch Peripheriegeräte auf Speicheradressen abgebildet (siehe nächstes Kapitel). Dann kann man denselben Mechanismus nutzen und z.B. einem Prozess das direkte Verwenden von Peripherie erlauben, einem anderen dieses verbieten. Allerdings werden häufig viele Geräte auf eine Seite abgebildet. Damit wird die Vergabe von Zugriffsrechten recht grob. Im einem Auto möchte man beispielsweise einem Programm Zugriff auf bestimmte Ventile erlauben, einem anderen (evtl. von einem anderen Zulieferer erstellten) Programm dagegen nur den Zugriff auf andere Ventile. Dies geht nicht, wenn beide Ventilmengen auf dieselbe Kachel bzw. Seite abgebildet sind. In solchen Situationen möchte man gern Mapping und Zugriffsschutz voneinander trennen.

Üblicherweise besitzt jeder Prozess seine eigene Seitentabelle. Bei der Prozessumschaltung werden dem entsprechend die Spezialregister mit den Informationen über diese Seitentabelle gefüllt. Die Inhalte der Spezialregister gehören mit zum Prozesskontext.

Die Voraussetzung einer zusammenhängenden Seitentabelle führt zu derer ineffizienten Nutzung, wenn im virtuellen Adressraum größere Lücken auftauchen. Für die Lücken beispielsweise zwischen *stack segment* und *data segment* (in Abb. 2.54 schraffiert gezeichnet) wird kein realer Speicher benötigt[18]. Allerdings sollte auch dafür gesorgt werden, dass für die Lücken im virtuellen Adressraum kein Platz in der Seitentabelle benötigt wird. Eine gewisse Abhilfe bieten hier hierarchische Seitentabellen. Hier werden einzelne Adressbits der virtuellen Adresse genutzt, um den Adressraum zu unterteilen und ggf. für ungenutzte Teile des Adressraums dann auch keine Einträge in Seitentabellen zur Verfügung zu stellen. Eine mögliche Strukturierung der Seitentabellen zeigt die Abbildung 2.56[19].

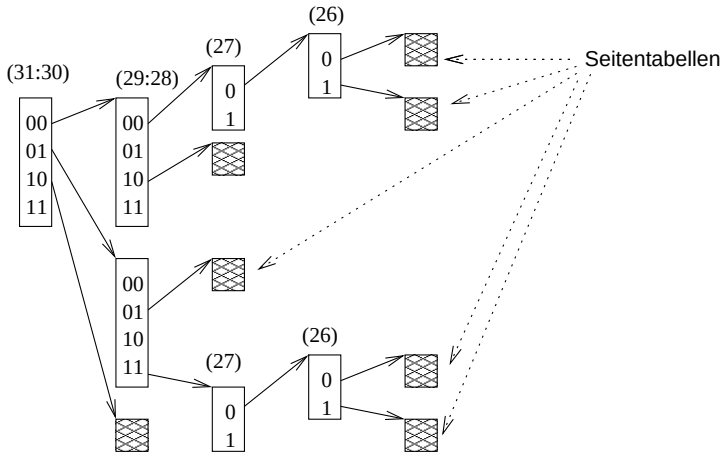


Abbildung 2.56: Hierarchische Seitentabellen

Selbst bei hierarchischen Seitentabellen kann der Speicherbedarf für Seitentabellen recht groß werden, insbesondere wenn es viele Prozesse mit großen Adressräumen gibt. Eine Möglichkeit der Reduktion des Speicherbedarfs ist die Verwendung so genannter „invertierter Seitentabellen“ (engl. *inverted page table*).

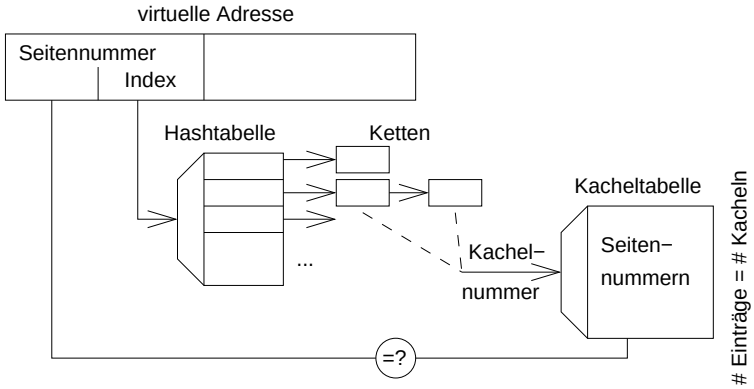


Abbildung 2.57: Invertierte Seitentabelle mit Zugriff über Hashing

Als wesentliche Datenstruktur haben wir hier eine Tabelle, die jeder Kachel die zugehörige Seite zuordnet (siehe Abb. 2.56). Wir können diese Tabelle **Kacheltabelle** nennen. Die Größe der Kacheltabelle ist recht überschaubar. Eine sequentielle Suche nach einer bestimmten Seite in dieser Tabelle wäre zu langsam. Deshalb benötigen wir eine weitere Datenstruktur, welche die Suche nach einem passenden Eintrag beschleunigt. Dafür sind so genannte **Hashtabellen** geeignet[20]. Beim Hashing benutzen wir einen Wert, den wir auf einfache Weise aus der Seitennummer ableiten, zur Indizierung in einer Tabelle, die uns den Zugriff auf eine relativ kleine Menge von möglichen Kandidaten für die korrekte Information liefert. Zum Beispiel können wir die Tabelle mit einigen weniger signifikanten Bits der Seitennummer adressieren. Wir erhalten so Verweise auf alle Einträge in der Kacheltabelle, deren zugeordnete Seiten in den gewählten Bits übereinstimmen. Diese Verweise können wir uns als eine Kette von Einträgen vorstellen. Für

[18] Wird eine Adresse benutzt, für die sich kein Eintrag in den Seitentabellen findet, so wird von vielen Betriebssystemen die Fehlermeldung „*segmentation fault*“ ausgegeben.

[19] Die entsprechenden Datenstrukturen sind ein Spezialfall so genannter **Vielweg-Suchbäume**, die üblicherweise in Vorlesungen über Algorithmen vorgestellt werden.

[20] Eine allgemeine Einführung in das Hashing gibt es in der Veranstaltung „Datenstrukturen, Algorithmen und Programmierung II“.

jedes Element der Kette benutzen wir die eingetragene Kachel, um die Kacheltabelle zu adressieren und auszulesen. Stimmt die Seitennummer aus der Kacheltabelle vollständig mit der Seitennummer aus der aktuellen virtuellen Adresse überein, so haben wir einen Treffer. Wenn nicht, so betrachten wir das nächste Element in der Kette. Gibt es ein solches nicht, so ist der virtuellen Adresse kein Speicher zugeordnet. Die Hashtabelle belegt natürlich zusätzlichen Speicherplatz. Umso kleiner die Hashtabelle ist, um so länger werden in der Regel die Ketten. Wir haben also einen Konflikt zwischen dem Wunsch, den zusätzlichen Speicher für die Hashdatenstruktur zu minimieren und dem Wunsch die Laufzeit, die sich durch das Abarbeiten der Ketten ergibt, klein zu halten.

Vielfach entsteht Bedarf, durch mehrere Prozesse auf denselben Speicher zuzugreifen. Man nennt dies *sharing*. Man kann dies dadurch realisieren, dass man gemeinsam genutzte Kacheln in die Seitentabellen mehrerer Prozesse einträgt (siehe Abb. 2.58).

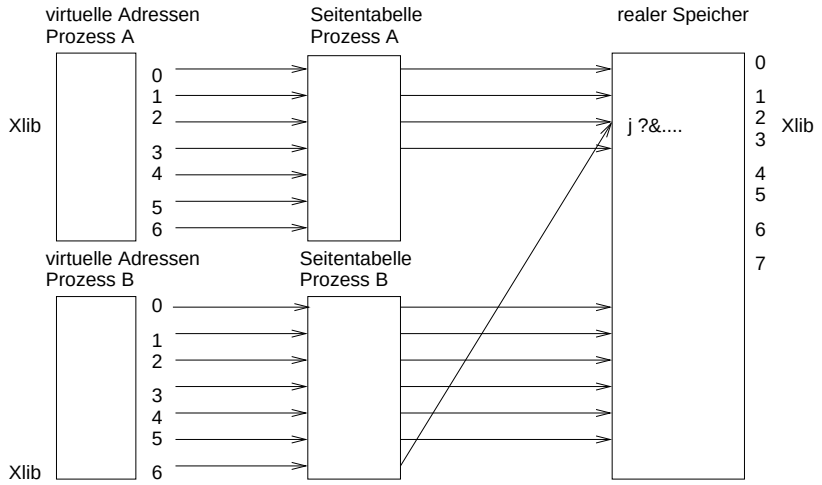


Abbildung 2.58: (Fehlerhafte) Idee zur Realisierung von *shared libraries*

Allerdings gibt es dabei ein Problem, wenn in den gemeinsam genutzten Kacheln wieder Adressen vorkommen. Diese Adressen müssen ja so gewählt werden, dass bei Benutzung der jeweiligen Seitentabellen auf die richtigen Informationen zugegriffen wird. Als Beispiel betrachten wir eine gemeinsam genutzte Softwarebibliothek (engl. *shared library*), die beispielsweise einen Sprung an eine (absolute) Adresse innerhalb derselben Bibliothek enthalten möge. Die im Sprungbefehl eingetragene Adresse muss bei Abbildung über die Seitentabelle des Prozesses `A` wieder in dieselbe Bibliothek verweisen. Im Beispiel wäre das die Seitennummer 2. Andererseits gilt dasselbe für den Prozess `B`. Damit bei Ausführung von `B` die Adressen korrekt werden, muss im Sprungbefehl im Beispiel die Seitennummer 6 eingetragen werden. In einer Speicherzelle kann aber nur eine Seitennummer eingetragen werden und diese ist unabhängig vom gerade ausgeführten Prozess. Das Fragezeichen steht in Abbildung 2.58 für das Problem, keine korrekte Seitennummer eintragen zu können. Deshalb kommen wir zu einer entscheidenden Einschränkung:

Nutzen Prozesse gemeinsamen Code, so muss dieser bei der Seitenadressierung bei allen beteiligten Prozessen auf dieselben virtuellen Adressen abgebildet werden (siehe Abb. 2.59). Im Beispiel kann jetzt im Adressenteil des Sprungbefehls eine 2 als Seitennummer eingetragen werden.

Diese Einschränkung erfordert Absprachen bei allen Bibliotheken, die auf einem Rechensystem als *shared library* ausgeführt werden[21].

Das Mapping hilft, den Speicher gut auszunutzen und nicht zu stark von den benutzbaren realen Adressen abhängig zu sein. Dieses Mapping ist selbst dann von Vorteil, wenn es keine anderen Speicher wie z.B. Plattenlaufwerke gibt. Diese Form der Seitenadressierung ist realzeitfähig, d.h. beim Zugriff auf den Speicher entstehen durch die Speicherverwaltung keine ungeplanten Zeitverzögerungen (von möglichen Verzögerungen beim Hashing für invertierte Seitentabellen einmal abgesehen). Sie wird daher zum Beispiel in manchen eingebetteten bzw. Realzeit-Systemen (Handys, PDAs, usw.) eingesetzt.

Gibt es jetzt einen Sekundärspeicher wie z.B. ein Plattenlaufwerk, so kann man einen Schritt weiter gehen. Benötigt jetzt zum Beispiel der Prozess `B` mehr Speicher, so kann der Inhalt der Seiten des Prozesses `A` auf eine Platte ausgelagert werden und die zugehörige Kachel kann dem Prozess `B` zusätzlich zur Verfügung gestellt werden. Greift später der Prozess `A` wieder selbst auf diese Seiten zu, so können sie wieder in den Speicher eingelagert werden. Dabei ist ein

[21] Bei 32-Bit Windows-Systemen ist hierfür die obere Hälfte der virtuellen Adressen vorgesehen.

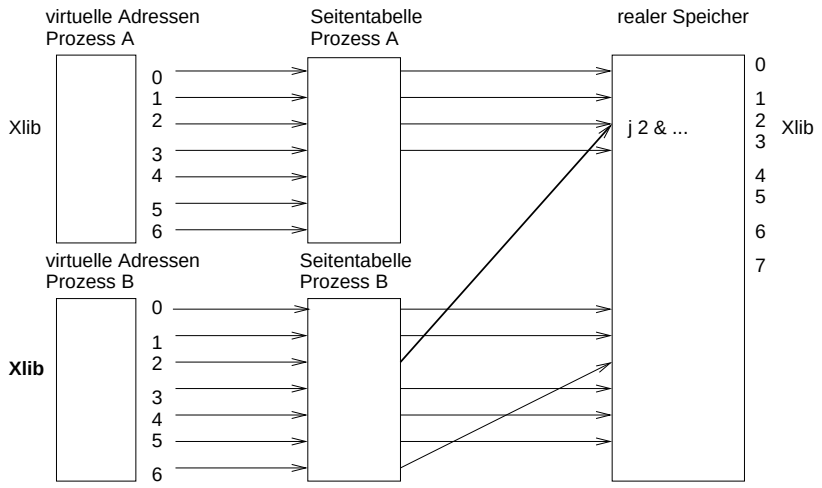


Abbildung 2.59: Mögliche Realisierung von *shared libraries*: Nutzung der Seite 2 in beiden Prozessen

Rückschreiben in beliebige Kachel möglich. Insgesamt kann jetzt mehr Speicher genutzt werden, als im Hauptspeicher tatsächlich vorhanden ist. Diese Form der Seitenadressierung wird der Deutlichkeit halber *demand paging* genannt. Demand paging ist möglicherweise die wichtigste Technik, um eine **Speicherhierarchie** zu realisieren. Wird nur von *paging* gesprochen, so ist im Allgemeinen unklar, ob *demand paging* oder die vorher beschriebene Form von *paging* gemeint ist. Während beim *demand paging* erst auf Anforderung Informationen von der Platte eingelagert werden, kann dieses bei *pre-paging* auch schon vorher geschehen. Man versucht dabei, zukünftige Zugriffe auf Seiten vorherzusagen. In der Praxis ist *pre-paging* darauf beschränkt, bei einem Einlagern einer Seite auch gleich deren Umgebung mit einzulagern.

Beim *demand paging* kann man noch versuchen, den Speicherbedarf für Seitentabellen im Hauptspeicher zu reduzieren, indem man diese selbst wieder im virtuellen Adressraum speichert. Allerdings muss ein Teil der Tabellen trotzdem im realen Speicher gehalten werden, damit man noch Adressen in reale wandeln kann.

2.4.2.3 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 diesem Sinne haben wir Segmente bereits mit der Abbildung 2.3 kennen gelernt. Bei der Segmentadressierung werden diesen Segmenten jetzt 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. 2.60). Da diese Basis von normalen Benutzern nicht geändert werden kann, heißt diese Organisation auch **Segmentadressierung mit verdeckter Basis**.

Abb. 2.61 zeigt, wie die Segmentadressierung eingesetzt werden kann, um die Segmente der Abbildung 2.3 auf realen Speicher abzubilden.

Bei Laden eines Segments in den Speicher ist keine Anpassung von Adressen an den jeweils genutzten Speicherbereich erforderlich, da diese Anpassung ja dynamisch zur Laufzeit erfolgt. Eine Ausnahme davon bildet das Laden von mehreren, voneinander unabhängig übersetzten Blöcken, die durch einen bindenden Lader (engl. *linking loader*) in dasselbe Segment gespeichert werden. Jede Übersetzungseinheit erzeugt in der Regel bei der Adresse Null beginnende Blöcke und beim Laden muss für alle Blöcke (bis auf den ersten) jeweils die relative Anfangsadresse des Blocks innerhalb des Segments auf die Adressen aufaddiert werden.

Segmente können sich zur Laufzeit an ihrem Ende ausdehnen und verkürzen. Falls eine Ausdehnung aufgrund belegten Speichers nicht möglich ist, muss das Segment in eine ausreichend große freie Lücke umkopiert werden. Das Finden einer Lücke kann beispielsweise mit dem *First-Fit*- und dem *Best-Fit*-Algorithmus ausgeführt werden. Beim

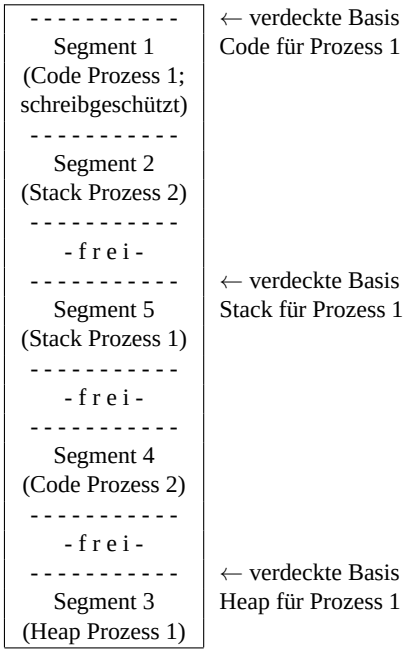


Abbildung 2.60: Speicherbelegung bei der Segmentierung mit verdeckter Basis

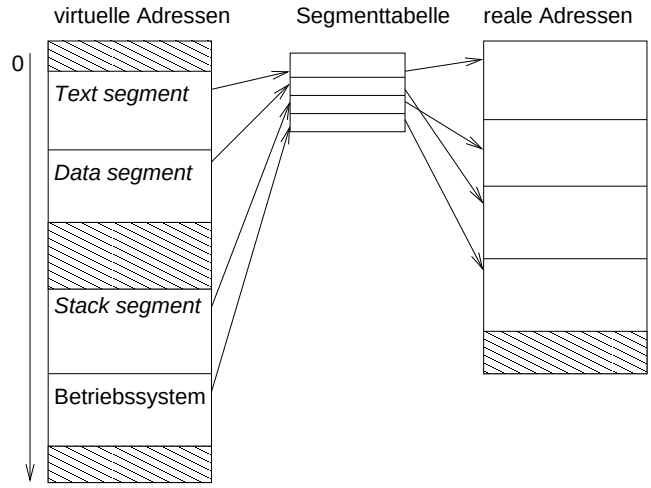


Abbildung 2.61: Abbildung der Speichersegmente der Abb. 2.3 auf realen Speicher mittels Segmentadressierung

First-Fit-Algorithmus wird der Speicherbereich sequenziell durchlaufen und die erste ausreichend große Lücke als neuer Speicherbereich des Segments ausgewählt. Beim *Best-Fit*-Algorithmus werden alle freien Speicherbereiche betrachtet und die Zuweisung erfolgt zu dem kleinsten, gerade ausreichend großen Speicherbereich (siehe Abb. 2.62 (a)). Ausführliche Untersuchungen sind zum Anteil der dabei im Mittel frei bleibenden Speicherbereiche durchgeführt worden (siehe z.B. Shore [Sho75]). Die Ergebnisse entsprechen dabei nicht immer den intuitiven Erwartungen. So ist der *Best-Fit*-Algorithmus häufig nicht besser als der *First-Fit*-Algorithmus, da er relativ kleine, kaum nutzbare Lücken hinterlässt.

Sofern keine ausreichend große Lücke mehr vorhanden ist, aber insgesamt immer noch ausreichend viel Spoeicher frei ist, müssen die Segmente **kompaktiert** werden, d.h. die Lücken werden durch Zusammenschieben der Segmente eliminiert. Dabei müssen ggf. größere Menge an Informationen im Speicher verschoben werden (siehe Abb. 2.62 (b)). Dies erfordert viel Zeit und ist daher ein ernsthafter Nachteil dieser Form der Speicherorganisation.

Aufgrund der möglichen Existenz von größeren Lücken spricht man bei der Segmentadressierung auch von der **externen Fragmentierung** des Speicher.

Reicht der Hauptspeicher insgesamt nicht mehr aus, so kann man bei vorhandenem größerem und langsameren Spei-

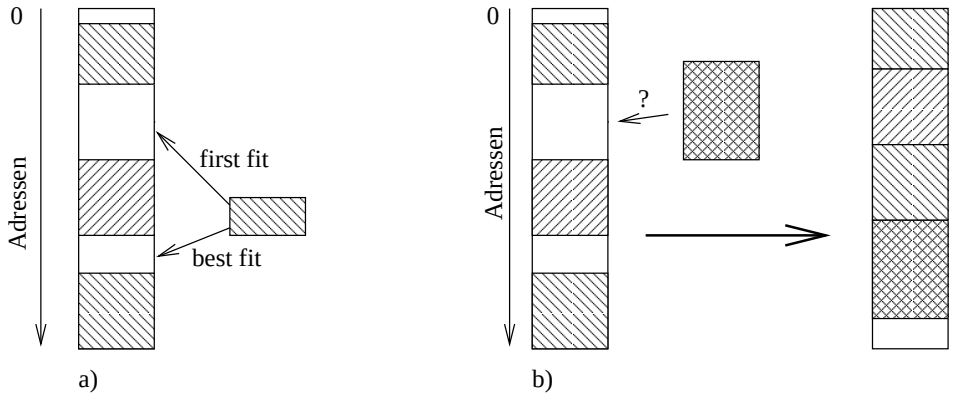


Abbildung 2.62: (a) Wahl einer Lücke; (b) Kompaktierung

cher (z.B. einem Plattenspeicher) Segmente auf diesen Speicher **auslagern**. Das Rückschreiben kann an eine beliebige ausreichend große Lücke erfolgen, da die Adressanpassung ja wieder dynamisch erfolgt.

Ein **Speicherschutz** ist ebenfalls möglich. Aufgrund der Adressanpassung kann auf Adressen unterhalb des Speicherbereichs des Segments ohnehin nicht zugegriffen werden. Zur Vermeidung des Zugriffs oberhalb des zugeordneten Speichers reicht eine Abfrage aus: diese prüft, ob benutzte Adressen größer sind als es der Segmentlänge entspricht. Diese Segmentlänge kann man mit der Segmentbeschreibung speichern. Man kann Segmente weiterhin mit Zugriffsrechten (*read, write, execute*) versehen. Die Vergabe dieser Rechte erfolgt im Gegensatz zur Seitenadressierung auf der Basis der logischen Zusammenhänge. Es ist kein Kopieren der Rechte in die Beschreibung aller betroffenen Seiten erforderlich.

Der Mechanismus zur Abbildung von virtuellen Adressen auf reale Adressen ist im Falle der Segmentadressierung gegenüber der Seitenadressierung 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. 2.63).

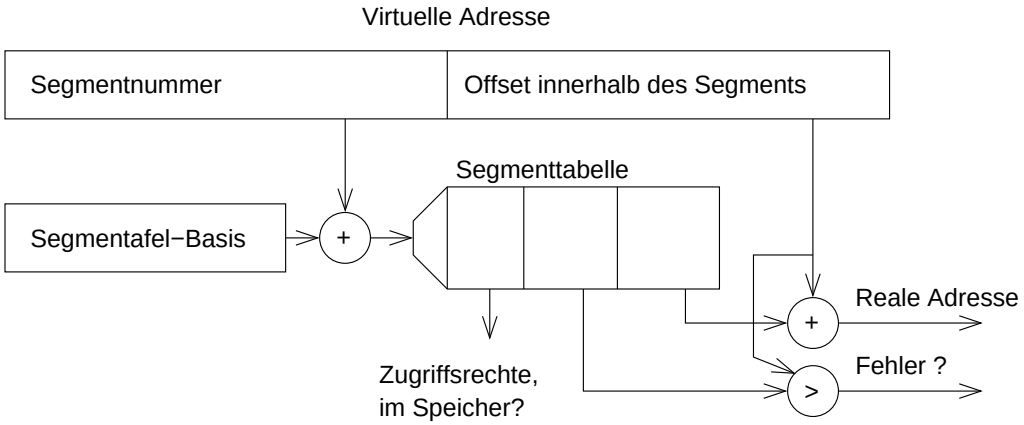


Abbildung 2.63: Prinzip der Adressabbildung bei der Segmentadressierung

Im Prinzip hätte man gern einen möglichst großen Adressraum für Segmentnummern. Dies würde es erlauben, jedem Speicherblock seine eigene Nummer zuzuordnen und die Zuordnung vielleicht sogar nie wieder rückgängig zu machen. Man könnte damit sogar Speicherobjekte in Archiven identifizieren (dies wurde so in dem Unix-Vorgänger MULTICS realisiert). Allerdings ist der Vorrat am Segmentnummer in der Praxis recht begrenzt. Es gibt dabei zwei Formen der Segmentierung: bei der ersten werden die Segmentnummern in den normalen Speicherworten abgespeichert. Bei 32-Bit Speicherworten reicht die Anzahl der Adressbits dabei weder für die gewünschte Größe des Segmentnummernraums noch für die gewünschte Größe des Adressraums innerhalb der Segmente. Dies ist die Organisationsform älterer Großrechner z.B. der Fa. IBM. Die Segmentadressierung ist hier ein Mittel der Speicherverwaltung, aber man muss teilweise schon Objekte (wie z.B. den Stack) auf mehrere Segmente abbilden, weil die maximale Größe eines Segments sonst nicht ausreichen würde. Diese erste Form der Segmentierung findet sich auch beim PowerPC, allerdings mit potenziell größerem Adressbereich. Bei der zweiten Form der Segmentierung benutzt man zur Darstellung der Segmentnummern separate Register. Damit ist ein ausreichend großer Segmentadressenraum sichergestellt.

Diese Form wird bei den Intel Pentium-Prozessoren eingesetzt.

Benutzt man separate Segmentregister, so entfällt das bei der Seitenadressierung vorgestellte Problem mit *shared libraries*: Unterschiedliche Prozesse können dann denselben Speicherbereich mit unterschiedlichen Segmentnummern ansprechen, da sich diese Segmentnummern ja in Registern befinden und die Registerinhalte können je nach ausgeführtem Prozess unterschiedlich sein, im Gegensatz zu den Seitennummern im Speicher. Bei der ersten Form der Segmentierung bleibt das Problem allerdings erhalten, da sich die Segmentnummern ja in Speicherzellen befinden.

2.4.2.4 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. 2.64).

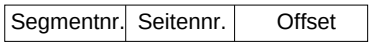


Abbildung 2.64: Aufteilung von Segmenten in Seiten

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.

Der Mechanismus zur Abbildung von virtuellen Adressen auf reale Adressen ist im Falle der Segmentadressierung mit Seitenadressierung noch einmal zu modifizeren. Die Abbildung ist jetzt zweistufig (siehe Abb. 2.65).

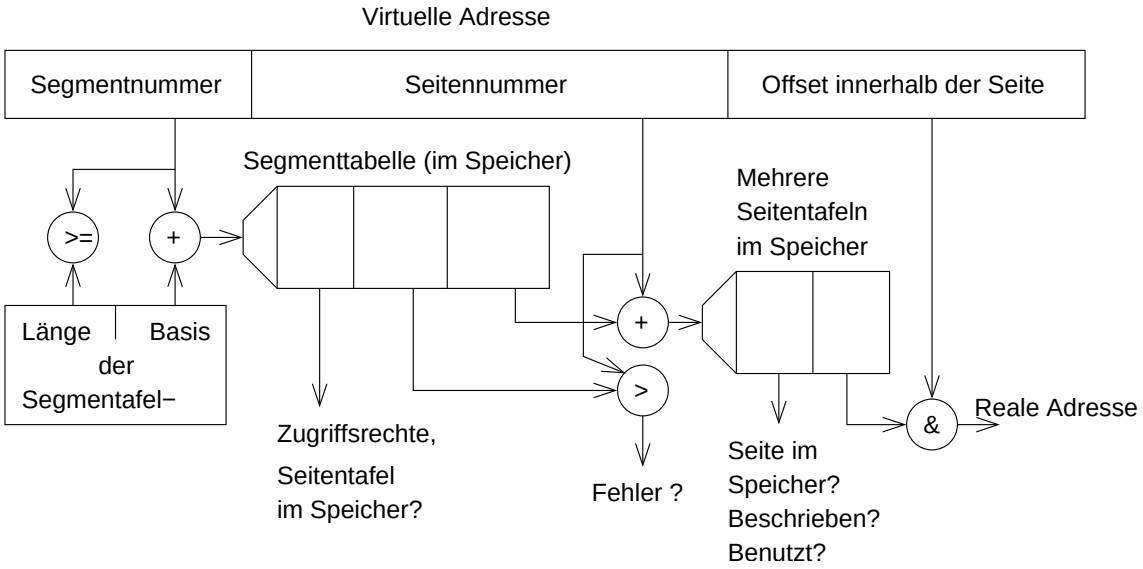


Abbildung 2.65: Segmentadressierung mit Seitenadressierung

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.

Segmentierung mit Seitenadressierung unter Verwendung separate Segmentregister wird von der Hardware der Pentium-Prozessoren bereitgestellt. Allerdings nutzen Windows-Systeme die damit mögliche günstige Realisierungsmöglichkeit von *shared libraries* nicht.

2.4.3 *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 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* (ATM). Sie haben eine Verzögerung von wenigen Nanosekunden 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*.

2.4.3.1 *Direct Mapping*

In diesem Fall adressiert die Seitennummer oder ein Teil davon einen kleinen, schnellen Speicher.

Überwiegend liegt dabei eine Situation vor, in welcher der Bereich möglicher Seitennummern so groß ist, dass nur ein Teil der Seitennummer zur Adressierung verwendet werden kann. Diese Situation zeigt die Abbildung 2.66.

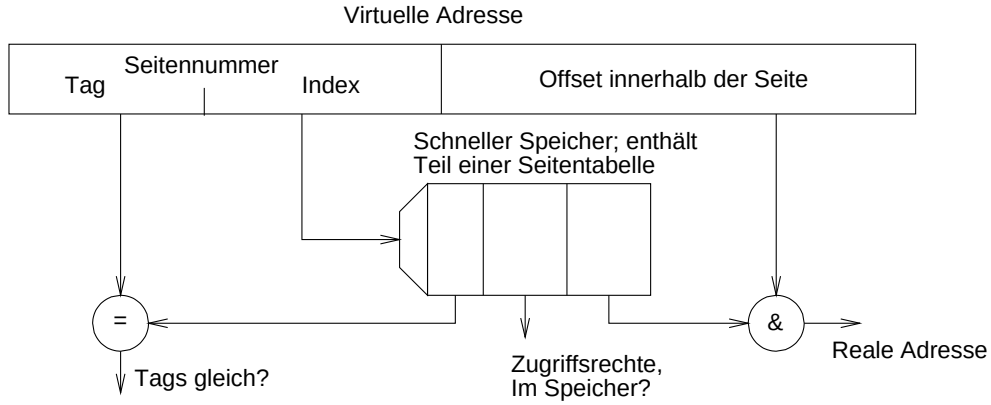


Abbildung 2.66: *Direct mapping* für übliche Seitentafel-Größen

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. Bei 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. 2.67).

In diesem Fall erfolgt die Umsetzung über den Hauptspeicher. Nachteilig ist, dass bei abwechselndem Zugriff auf derartige 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 und die Zeiten für das Umladen bei Prozesswechseln bleiben moderat.

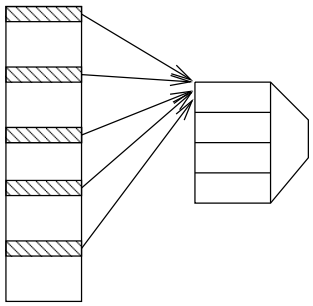


Abbildung 2.67: Seiten, die demselben TLB-Eintrag zugeordnet werden

Aufgrund der Konflikte in Abb. 2.67 hätte man gern mehrere **Tag**-Einträge und würde gern parallel suchen, ob der passende Eintrag dabei ist.

2.4.3.2 Mengen-assoziative Abbildung

Dies wird bei der **Mengen-assoziativen Abbildung** (engl. *set associative mapping*) realisiert (siehe Abb. 2.68).

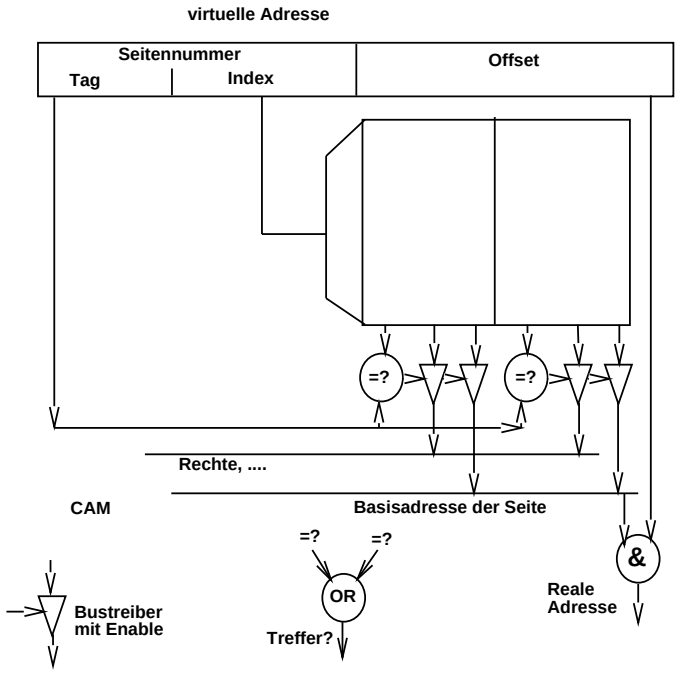


Abbildung 2.68: Mengen-assoziative Adressabbildung (Setgröße=2)

Mit einem Teil der Seitennummer, Index genannt, wird der Speicher wie beim *direct mapping* adressiert. Im Unterschied zum *direct mapping* gibt es aber jetzt für jeden Index mehrere Einträge über zugeordnete Seiten. Unter den adressierten Einträge wird parallel nach einem Eintrag gesucht, der im verbleibenden Teil der Seitennummer, *tag* genannt, übereinstimmt. Sofern der Vergleich der *tags* positiv ausfällt, wird der Eintrag aus dem Speicher weiterleitet. Hierzu kann beispielsweise ein so genannter „Bustreiber mit *enable*" dienen, der den Eintrag weiterleitet, wenn an seinem *enable*-Steuereingang eine '1' anliegt. Fällt der Vergleich für keinen Eintrag positiv aus, so erfolgt eine Umsetzung über die Tabelle im Hauptspeicher. In diesem Fall wird einer der „alten" Einträge im TLB durch die Information zur gerade benutzten Seite überschrieben. Die Auswahl des zu überschreibenden Eintrags kann dabei mit der *least-recently-used* (LRU)-Regel erfolgen: es wird immer der Eintrag ersetzt, auf den am Längsten nicht zugegriffen wurde.

Bei Prozesswechsel werden die Einträge als ungültig erklärt.

Vorteile der Mengen-assoziative Adressabbildung liegen in der immer noch einfachen Realisierbarkeit und der weit-

gehenden Freiheit von Anomalien des *direct mappings*.

Im Fall $||Index|| = 0$ entsteht der (voll) assoziative Speicher.

2.4.3.3 Assoziativspeicher, *associative mapping*

Beim Assoziativspeicher wird der Speicher nicht mehr addressiert, sondern es erfolgt nur noch ein paralleler Vergleich des *tags* der aktuellen virtuellen Adresse mit den gespeicherten *Tags*. Dies ist mit einem so genannten **inhaltsadressierbaren Speicher** (engl. *content addressable memory* (CAM)) möglich. Damit ist eine gleichzeitige Suche nach der Seitennummer in allen Einträgen des TLB realisiert. Abbildung 2.69 zeigt die Verwendung eines Assoziativspeichers.

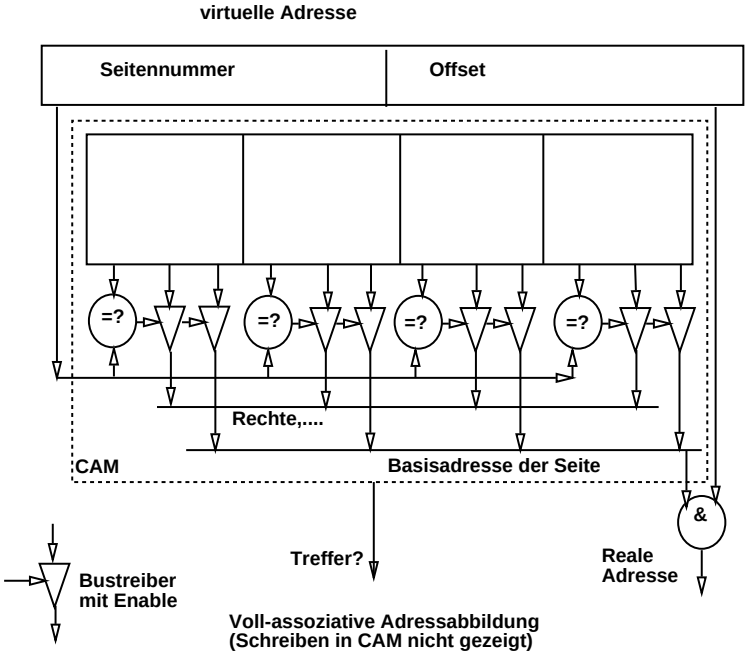


Abbildung 2.69: (Voll)-assoziative Adressabbildung

Der Indexteil ist dabei vollständig entfallen. CAMs können für eine kleine Zahl von Einträgen (z.B. bis zu 64) hergestellt werden. Bei sehr viel größeren Anzahlen von Einträgen würde der Aufwand zu hoch werden.

Falls kein Treffer gefunden wird erfolgt die Umsetzung wieder über den Hauptspeicher und die Information über die Seite verdrängt anschließend einen „alten“ Eintrag im TLB, z.B. nach der LRU-Regel.

Bei Prozesswechsel ist die Information im TLB offensichtlich zu löschen, da sich die Abbildung von Seiten- auf Kachelnummern ändert.

Zu den Vorteilen der voll-assoziativen Abbildung gehört, dass es keine Abhängigkeit von bestimmten Folgen von Seitenreferenzen gibt, im Gegensatz zum *direct mapping*. Es ergeben sich gute Trefferraten selbst bei kleinem Speicher und großem Adressraum, wenn sich der Prozess lokal verhält (Angaben von 98 % Treffern)

Eine Realisierung des *associative mappings* findet sich beispielsweise in der *Memory Management Unit* (MMU) Motorola MC 68851[22].

Die MMU MC 68851 enthält einen voll-assoziativen TLB mit 64 Einträgen. Es erfolgt eine Adressumrechung über die Tabellen in Speicher, wenn die MMU keinen passenden Eintrag findet. Die Seitentabelle ist dabei zur Vermeidung der Speicherung von Informationen in ungenutzten Lücken des Adressbereichs baumartig organisiert, wie in Abb. 2.56 bereits gezeigt.

Das am wenigsten signifikante Byte wird dabei von der virtuellen zur realen Adresse gleich weitergereicht. Die kleinste mögliche Seitengröße beträgt damit 256 Byte.

[22] Diese MMU wurde hier ausgewählt, weil sie ursprünglich diskret hergestellt wurde, d.h. als Zusatzbaustein zum Prozessor. Heutige MMUs sind im Prozessorbaustein integriert und damit etwas weniger leicht identifizierbar.

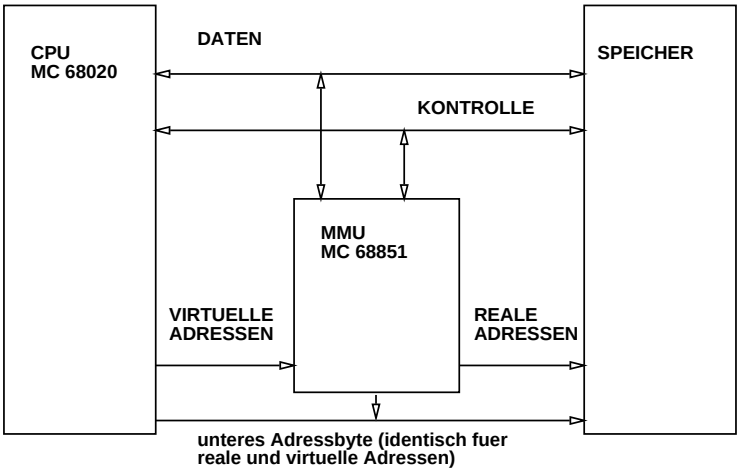


Abbildung 2.70: Adressabbildung mittels MMU

Die MMU erlaubt eine Erweiterung der virtuellen Adressen um eine 3-Bit **Prozess**- bzw. **Adressraum-Identifikation** (PID). Damit kann zwischen den virtuellen Adressen verschiedener Prozesse unterschieden werden.

2.4.4 *Caches*

2.4.4.1 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 (siehe Abb. 2.71).

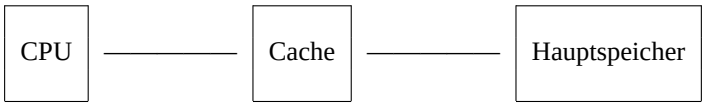


Abbildung 2.71: Anordnung eines Caches (im engeren Sinn)

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. 2.72) . 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.
- 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.

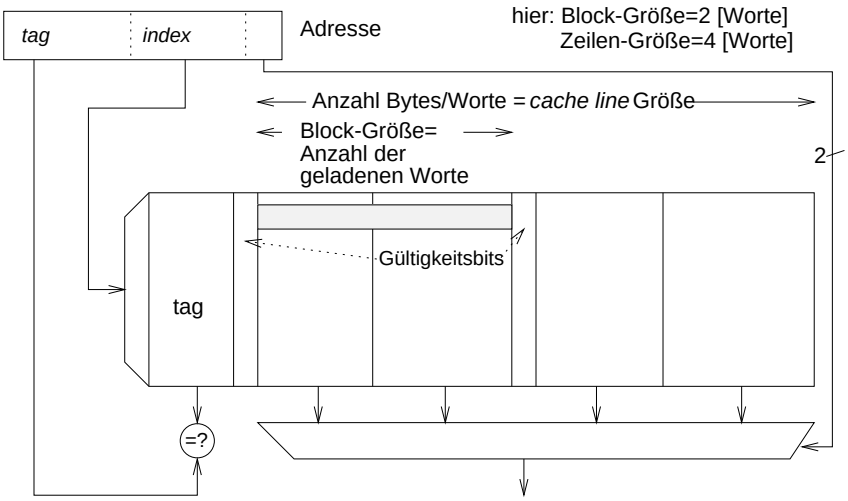


Abbildung 2.72: Allgemeine Organisation eines Caches

- 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 aufeinander folgender Adressen (*nibble mode*, *block mode* der Speicher)
  2. *Interleaving*, d.h. aufeinander folgende 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. 2.72 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.

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 aufeinander folgende 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 2.4.5).

- *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 Vergleiche für große Datenblöcke, die nicht vollständig nachgeladen werden.

2.4.4.2 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. 2.73.

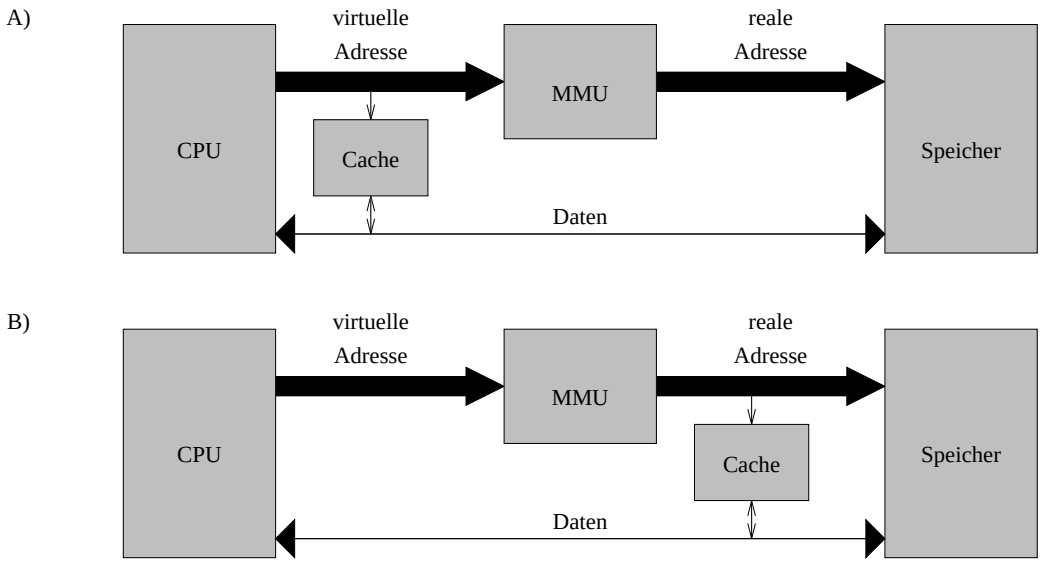


Abbildung 2.73: Anordnungsmöglichkeiten von Caches

Abhängig von der Anordnung ergeben sich unterschiedliche Geschwindigkeiten: virtuelle Caches sind meist schneller als reale: bei virtuellen Caches sind die Zugriffe auf den Cache und auf den TLB voneinander nicht kausal abhängig, können also nebenläufig ausgeführt werden.

Allerdings muss man sich überlegen, wann Caches oder Teile davon jeweils ungültig werden. Entsprechende Aktionen sind in Abschnitt 2.4.4.4 beschrieben.

2.4.4.3 Schreibverfahren

Bezüglich des Rückschreibens in den 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.

2.4.4.4 Cache-Kohärenz

2.4.4.4.1 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 Prozesswechseln 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. 2.74 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	(teilweise) ungültig	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 2.74: Vergleich von Caches

2.4.4.4.2 Systeme mit mehreren Caches: Viele moderne Rechnerarchitekturen bieten mehrere Cache-Ebenen: zum Beispiel 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. Wie wird man diese beiden Caches organisieren? Möglich ist z.B. folgende Wahl: der interne Befehlscache 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*. .

2.4.5 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[23]:

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
   (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.

   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.

2.4.6 Sekundärspeicher-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 teilweise in einem nichtflüchtigen Teil des Hauptspeichers realisiert (NVRAM).

2.4.7 Sekundärspeicher

Unter dem Begriff Sekundärspeicher verstehen wir hier die erste Stufe der Speicherhierarchie jenseits des Primärspeichers. In der Regel wird keine direkte Adressierbarkeit über Maschinenbefehle gegeben sein. Typischerweise erwarten wir eine persistente Speicherung.

Zur technischen Realisierung stehen dafür derzeit v.a. Flashspeicher und Plattenlaufwerke (hard discs) auf der Basis magnetischer Speicherung zur Verfügung.

[23] Beschreibungen weiterer Verfahren findet man bei Baer [Bae80] sowie in vielen Büchern über Betriebssysteme (siehe z.B. Tanenbaum [Tan05]).

2.4.7.1 Magnetplatten

Abb. 2.75 (links) zeigt den schematischen Aufbau eines Plattenlaufwerks. Die Aufteilung in Spuren und Sektoren zeigt der rechte Teil derselben Abbildung (nach [HP02]).

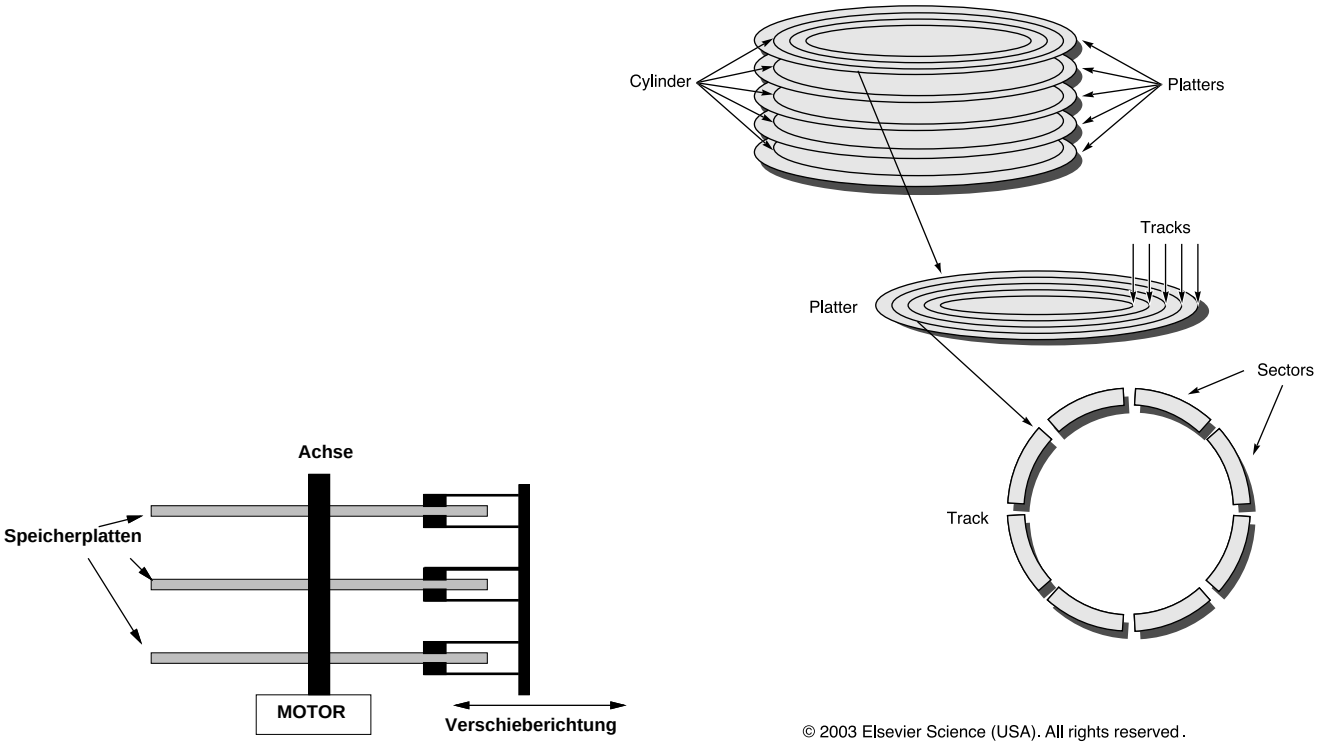


Abbildung 2.75: Plattenlaufwerk

2.4.7.1.1 Disc-Arrays: Eine spezielle Organisationsform von Plattenlaufwerken bilden die so genannten **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*) . RAID basiert auf den Prinzipien der Duplizierung oder Spiegelung (engl. *mirroring*) und der Paritätsinformation (engl. *parity*) zur Verbesserung der Fehlertoleranz in Kombination mit dem Prinzip der Verteilung von Daten zur Verbesserung der Zugriffsgeschwindigkeit. Dabei gibt es eine große Anzahl von Kombinationsmöglichkeiten mit teilweise recht subtilen Unterschieden. Folgende Verfahren sind normiert [HP07, CLG+94, Koz04]:

- RAID 0

  Bei RAID 0 wird so genanntes *Striping* eingesetzt. Beim *Striping* unterteilt man Dateien in kleinere Einheiten, die nicht unbedingt mit den üblichen Blöcken übereinstimmen müssen. Der *Striping*- Parameter kennzeichnet dabei die Länge der Einheiten. Bei kleinem Striping-Parameter (z.B ein Byte) erreicht man eine hohe Übertragungsrate, verwendet aber evtl. auf jeder der Platten nur kleine Bereiche.

  Bei RAID 0 werden die Stripes einer Datei werden auf mehreren Platten abgelegt (siehe Abb. 2.76). Bei diesem Verfahren werden mehrere kleine Laufwerke zu einem großen logischen Laufwerk zusammengefasst. Da keine Redundanz vorhanden ist, ist das Verfahren störanfällig und dürfte eigentlich nur AID-0 heißen.

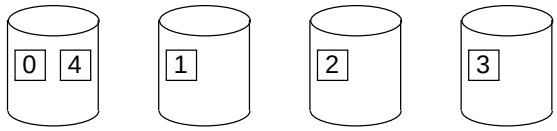


Abbildung 2.76: Verteilung von *Stripes* auf verschiedenen Platten bei RAID 0

- RAID 1

  Bei diesem Verfahren wird mit **gespiegelten Platten** (engl. *mirrored discs*) gearbeitet, d.h. dieselben Daten werden auf zwei Platten geschrieben (siehe Abb. 2.77). Die Lesegeschwindigkeit kann so erhöht werden. Fällt eine aus, so kann mit den Daten der anderen Platte weitergearbeitet werden. Nachteil ist der doppelte Plattenaufwand.

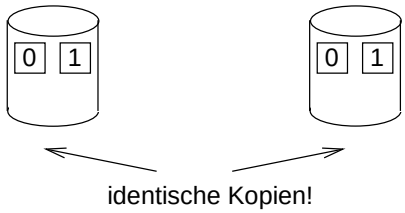


Abbildung 2.77: Spiegelung von Daten auf verschiedenen Platten bei RAID 1

- RAID 2

  Bei RAID 2 verwendet man einen *Striping*-Parameter von einem Bit: die **Bits** eines Datenblocks werden auf mehreren Platten verteilt (engl. *bit interleaving*). Es werden Prüfbits auf spezielle Prüfbit-Platten geschrieben. Dabei werden wie bei fehlerkorrigierenden Hauptspeichern fehlerkorrigierenden Codes (engl. *error-correcting codes* (ECC)) eingesetzt. So können trotz des Ausfalls von Platten die Daten regeneriert werden. Dabei müssen fehlerhafte Plattenlaufwerke nicht selbst einen Fehler bemerken und signalisieren. 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. RAID 2 erfordert recht spezielle Hardware. Es wird daher kommerziell nicht eingesetzt.

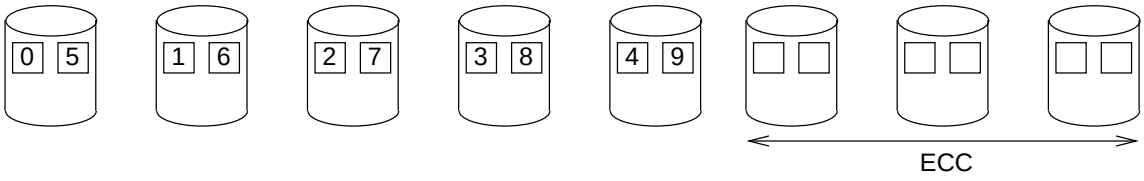


Abbildung 2.78: Verwendung fehlerkorrigierender Codes bei RAID 2

- RAID 3

  Bei RAID 3 wird ausgenutzt, dass Plattenlaufwerke aufgrund interner Fehlererkennungsmaßnahmen in der Regel selbst erkennen, ob sie defekt sind. Dann reicht ein einzelnes zusätzliches Paritätsbit aus, um trotz des Ausfalls einer Platte eine Datei korrekt zu lesen. Dieses Paritätsbit wird auf einer speziellen Platte gespeichert. Bei RAID 3 werden alle Köpfe gleichzeitig genutzt, um Dateiinformationen zusammenzusetzen. Es besitzt daher eine relativ gute Leseperformance.

  Unter RAID 3 versteht man die Verwendung eines einzelnen Paritätsbits mit einem *Striping*-Parameter von einer gewissen Anzahl von Bytes.

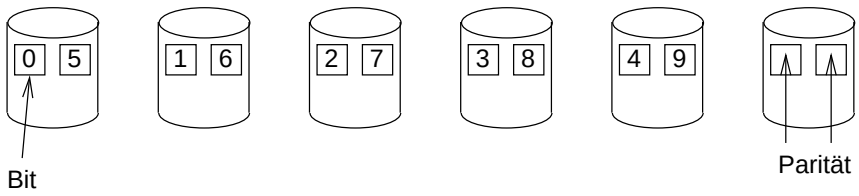


Abbildung 2.79: Benutzung eines einzelnen Paritätsbits auf einer speziellen Platte bei RAID 3

- RAID 4

  Wie RAID 3, jedoch mit einem *Striping*-Faktor von einem Block oder mehr. RAID 3 unterscheidet sich von RAID 3 durch die Benutzung von Blöcken statt Bytes für den Striping-Parameter und von RAID 5 durch die Benutzung einer dedizierten Paritätsplatte statt einer verteilten Parität.

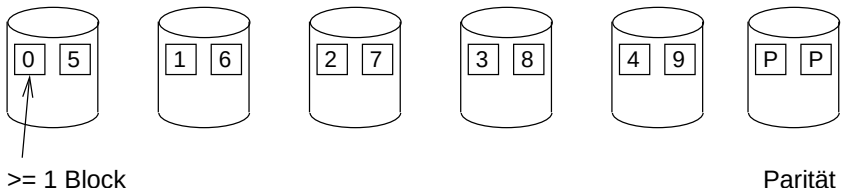


Abbildung 2.80: Verteilung von Blöcken auf verschiedenen Platten bei RAID 4

- RAID 5

  Bei RAID 5 wird die Paritätsinformation über verschiedene Platten verteilt. So wird die Paritätsplatte als Performance-Engpass vermieden.

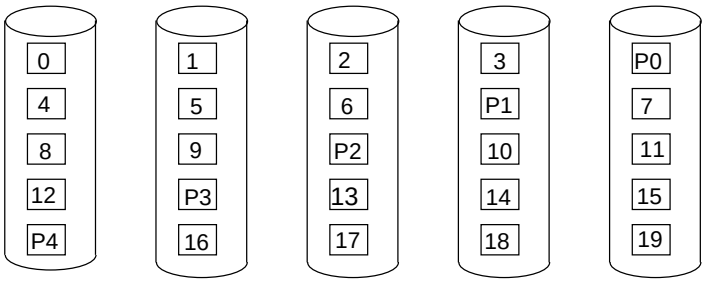


Abbildung 2.81: Verteilung der Paritätsinformation auf verschiedenen Platten bei RAID 5

Geschachtelte RAIDs

Durch Kombination von verschiedenen RAID-Varianten kann man versuchen, die besten Eigenschaften der RAID-Varianten auszunutzen [Koz04]. So ist es beispielsweise möglich, RAID 0 und RAID 1 miteinander zu kombinieren. Es ist dies auf zwei Weisen möglich. Dies sei am Beispiel von 10 Plattenlaufwerken gezeigt:

- Man kann zwei Mengen von je 5 Plattenlaufwerken bilden. Innerhalb jeder Menge realisiert man *Striping* und sorgt dafür, dass die eine Menge die andere spiegelt. Man nennt dies *mirrored striped discs* und notiert dies als RAID 1+0 oder RAID 10.
- Andererseits könnte man 5 Mengen von je zwei Plattenlaufwerken bilden und innerhalb jeder Menge die Platten spiegeln. Über diese 5 Mengen könnte man dann ein *Striping* realisieren. Man nennt dies *striped mirrored discs* und notiert es als RAID 0+1 oder RAID 01.

RAID 3 kann mit RAID 0 zur Verbesserung der Performance und mit RAID 1 zur Verbesserung der Fehlertoleranz verbunden werden. Je nach den Details entstehen dadurch die Organisationsformen RAID 0+3, 3+0, 1+3 bzw. 3+1, auch als RAID 03, 30, 13 bzw. 31 bezeichnet.

RAID 5 kann mit RAID 0 zur Verbesserung der Performance und mit RAID 1 zur Verbesserung der Fehlertoleranz verbunden werden. Je nach den Details entstehen dadurch die Organisationsformen RAID 0+5, 5+0, 1+5 bzw. 5+1, auch als RAID 05, 50, 15 bzw. 51 bezeichnet.

Eine Übersicht über die Eigenschaften einiger RAID-Varianten bietet die Tabelle 2.15.

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 [Tan05]) ausreichend Information.

[1] Es werden eigentlich nur 2 Platten benutzt.

RAID-Variante	Technik	Fehler toler.	Daten-Platten	Prüf-platten
0	*nonredundant*	0	8	0
1	*mirrored*	1	8[1]	8
0+1	Kombination aus RAID 0 und 1	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* [HP07]	2	8	2

Tabelle 2.15: Vergleich von RAID-Varianten

2.4.7.2 *Flash*-Speicher

Flash-Speicher werden üblicherweise ebenfalls als Sekundärspeicher eingesetzt. Aus Flash-Speichern werden zunehmend so genannte *solid state discs* konstruiert, die ohne bewegliche Teile auskommen. Näheres siehe Folien!

2.4.8 Tertiärspeicher

Unter Tertiärspeicher wollen wir hier Speicher verstehen, der nicht permanent mit unserem System verbunden ist. Statt dessen muss das Speichermedium in der Regel mechanisch zunächst mit dem System verbunden werden, also z.B. eine CD oder ein Magnetband manuell oder durch einen Automaten eingelegt werden. Dementsprechend ergeben sich Zugriffszeiten im Sekunden- oder sogar Minutenbereich. Diese Stufe wird derzeit üblicherweise von optischen Speichern und Magnetbändern gebildet.

2.4.8.1 Optische Speicher

- **CD-ROM/CD-R/CD-RW**

  Die CD-ROM, entsprechend „akustischen” CDs, erreicht standardmäßig eine Speicherkapazität von 650 MByte, in neueren Versionen auch etwas mehr. Zugriffszeiten liegen in der Größenordnung von ca. 100 ms.

  Den grundsätzlichen Aufbau zeigt die Abb. 2.82[24].

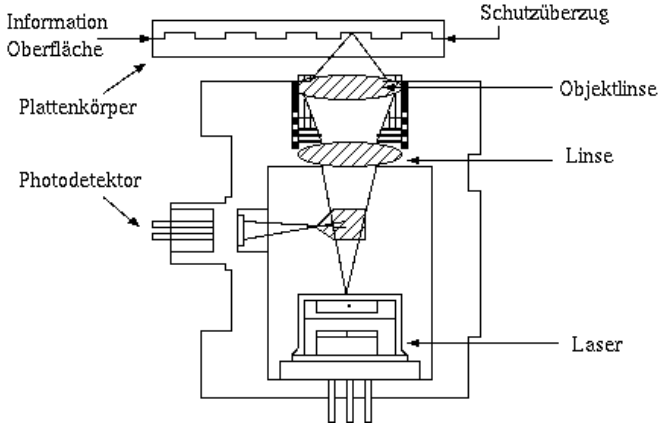


Abbildung 2.82: Grundsätzlicher Aufbau eines CD-ROM-Laufwerkes

  Die Intensität des reflektierten Lichts ist dabei abhängig von den Vertiefungen auf den Datenträgern.

- **DVD**

[24] Quelle: [aia.wu-wien.ac.at/inf_wirt/03.04.01.html]

Ziel der Entwicklung der DVD war es, gut 2 Stunden MPEG-komprimiertes Video mit Standard-Fernseh-Qualität auf einem Speichermedium speichern zu können. Bei der DVD wird die Kapazität von CD-ROMs mit verschiedenen Mitteln erhöht. Zunächst einmal werden die Abstände der Vertiefungen und deren Größe gegenüber der CD-ROM reduziert (siehe Abb. 2.83. Die Reduktion wurde u.a. durch eine etwas kürzere Wellenlänge des Lichts der eingesetzten Laser erreicht: als Wellenlänge wurden 650 nm gewählt.

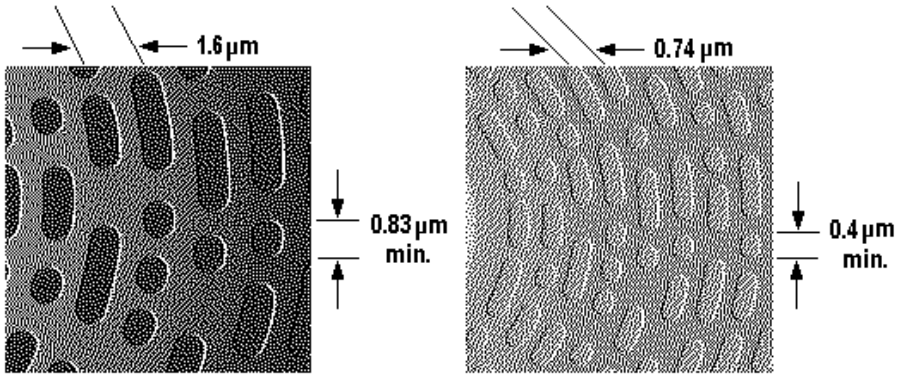


Abbildung 2.83: Vergleich der Vertiefungen bei der DVD und der CDROM

Durch diese Maßnahmen ergibt sich 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. 4,7 GB erlauben die Aufzeichnung von etwa 2 Stunden und 15 Minuten Video.

Zusätzlich kann man durch Umfokussieren zwei Ebenen je Seite einer DVD nutzen (siehe Abb. 2.84). Dadurch kommt man auf eine Kapazität von 8,5 GB je Seite einer DVD.

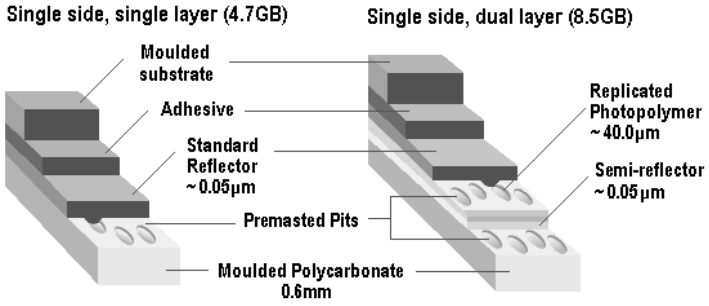


Abbildung 2.84: Ausnutzung zweier Ebenen je DVD-Seite

Schließlich kann man noch beide Seiten einer DVD nutzen und kommt so auf bis zu 17 GB je DVD (siehe Abb. 2.85). Die Dicke der Schutzschicht über den beschriebenen Ebenen beträgt 0,6 mm, die gesamte Dicke etwa 1,2 mm.

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.

- ***Blu-Ray Discs* (BD)**

  Aufgrund der Einführung von hochauflösendem Fernsehen reichen DVDs für die Speicherung von 2 Stunden Video in einer Speicherebene nicht mehr aus. Für eine Spielzeit von 2 Stunden und 15 Minuten werden bei hochauflösendem Fernsehen 22 GB benötigt. Daher musste die Speicherkapazität weiter erhöht werden. Zur Reduktion der Abstände auf dem Speichermedium musste die Wellenlänge der eingesetzten Laser verkürzt werden. Die beim DVD-Nachfolger eingesetzte Wellenlänge von 400 nm liegt im Bereich blauen Lichts. Dies ist der Grund für die Wahl des Namens des Nachfolgers der DVD. Daher die korrekte Schreibweise mit „e" nicht als Marke geschützt werden konnte, wurde die Schreibweise ohne „e" gewählt. Zur Erhöhung der Kapazität musste die Dicke der Schutzschicht über der beschriebenen Fläche auf 100 μm reduziert werden [Blu04].

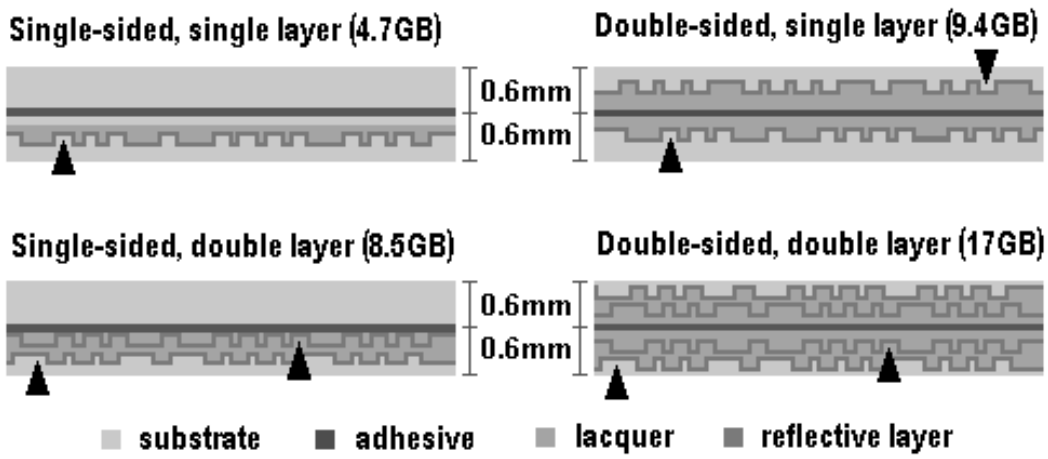


Abbildung 2.85: Speicherkapazitäten von DVDs

2.4.8.2 Magnetbänder

Für die Datensicherung ist prinzipiell der Einsatz von Bandlaufwerken mit sequenziellem Zugriff auf die Daten möglich. Aufgrund der Trennung von Speichermedium und Aufnahme- und Wiedergabeelektronik gibt es ein Potenzial zur Kostenreduktion gegenüber Plattenlaufwerken.

Verbreitete Formate sind die folgenden [Beh99]:

- **QIC-Kassetten**

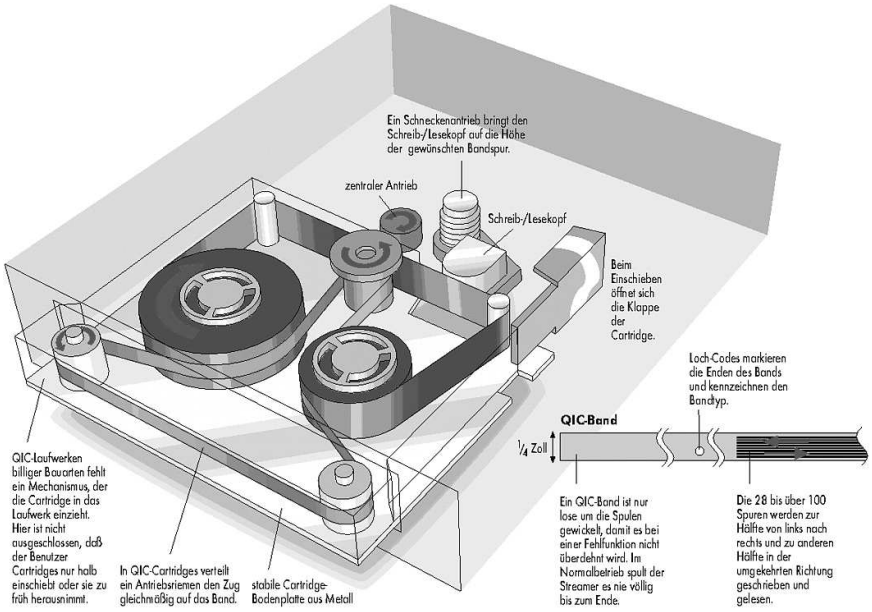


Abbildung 2.86: QIC-Laufwerk (©c't)

QIC steht für *quarter inch cartridge*. QIC-Kassetten (siehe Abb. 2.86) 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. 2.87) 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.

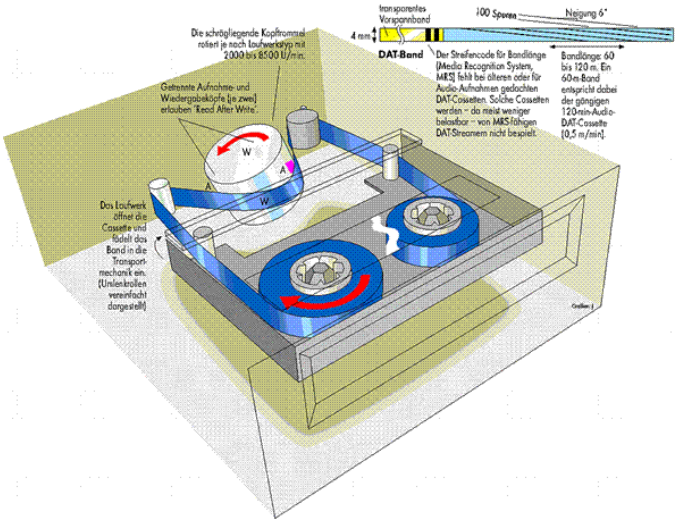


Abbildung 2.87: DAT-Laufwerk (©c't)

- **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).

- **LTO-4 Bänder**

  Näheres siehe Folien!

Die Steigerungsraten der Speicherkapazität lag in den letzten Jahren aber immer deutlich unter den Steigerungsraten für Plattenlaufwerke. Aufgrund des großen Marktes konnten die Preise für Plattenlaufwerke außerdem stärker gesenkt werden als die Preise für Bandlaufwerke. Als Folge davon ergibt sich für Bandlaufwerke laut Hennessy/Patterson ab etwa 2001 kein wesentlicher Preisvorteil mehr gegenüber Plattenlaufwerken [HP02], wobei dies auch teilweise anders gesehen wird.

2.5 Kommunikation, Ein-/Ausgabe

"*Die Wissenschaft Informatik befasst sich mit der Darstellung, Speicherung, Übertragung und Verarbeitung von Information*" [Ges06]. In diesem Kapitel geht es um die Übertragung bzw. die Kommunikation.

Rechensysteme ohne Kommunikations- bzw. Ein- und Ausgabemöglichkeiten[25] (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. 2.88 sind der Bus, die Steuerungen und die E/A-Geräte eines Rechensystems hervorgehoben.

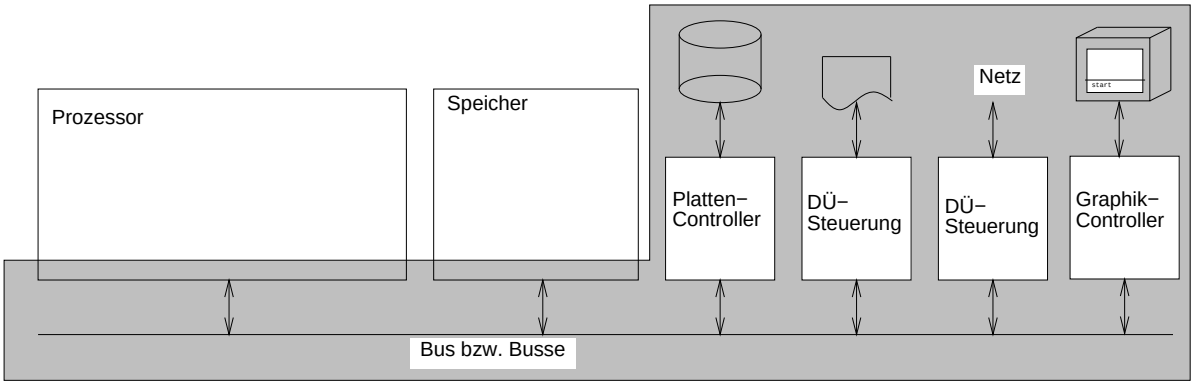


Abbildung 2.88: Rechner mit Kommunikations- bzw. Ein-/Ausgabesystem

2.5.1 Bussysteme

Zunächst beschäftigen wir uns mit Bussen und deren Anschluss an die Controller. Die Bus-Organisation müssen wir uns im Folgenden genauer ansehen.

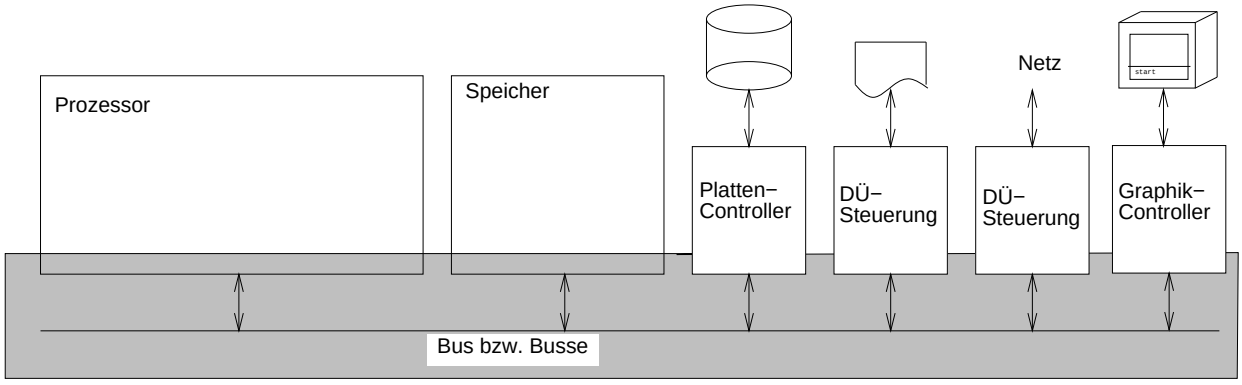


Abbildung 2.89: Zum Thema des Abschnitts 4.1

2.5.1.1 Topologien

Aufgrund der jeweils vorhandenen Entwurfsbeschränkungen und Zielvorstellungen ist eine Vielzahl von Bus-Topologien im Gebrauch.

[25] 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.

- In einer Reihe von Fällen findet man in der Praxis Bussysteme, die auch bei detaillierter Betrachtung weitgehend das Schema der Abb. 2.88 widerspiegeln. Ein Beispiel dafür ist der Bus der 68000er-Prozessorfamilie. Eine Ursache hierfür ist die begrenzte Anzahl von Pins, die an derartigen Prozessoren vorhanden ist. An diesem Bus werden bei einfachen Systemen sowohl der Speicher als auch Gerätesteuerungen direkt angeschlossen (siehe Abb. 2.90).

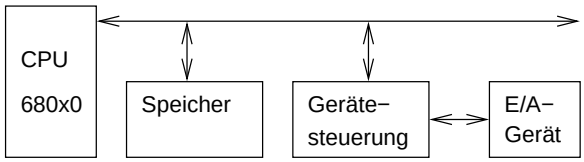


Abbildung 2.90: Bus der 68000er-Prozessorfamilie (vereinfacht)

Die in Abb. 2.90 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.

- Systeme, die auf Mikroprozessoren basieren, gehen häufig einen Weg zur Entkopplung von Speicher- und E/A-Bus. In der Regel wird hierzu ein sog. **Busadapater** bzw. eine **Bus-Brücke** realisiert (siehe Abb. 2.91).

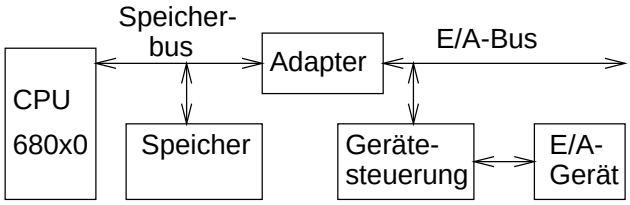


Abbildung 2.91: Separate Speicher- und E/A-Busse

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. 2.92 zeigt die Grundstruktur älterer PCI-basierter Systeme.

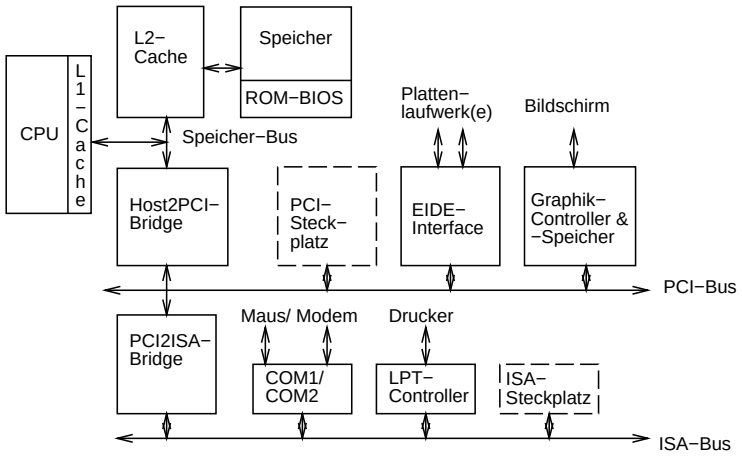


Abbildung 2.92: Blockschaltbild eines PCI-Systems

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. Abb. 2.93 zeigt den Speicherzugang für den Chipsatz VIA Apollo KT133. Zentral für die Systemleistung ist die sog. *north bridge*, die Prozessor, Graphikkarte, Speicher und PCI-Bus miteinander verbinden.

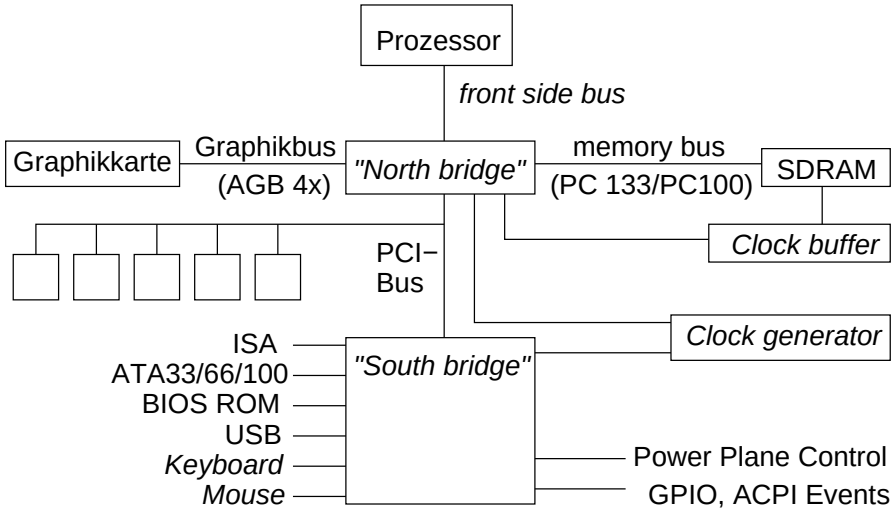


Abbildung 2.93: Bustopologie mit sog. *north bridge* und *south bridge*

- Der PCI-Bus wird für manche Karten (wie z.B. Gigabit-Ethernetkarten) zu einem Engpass. Als Nachfolger wurde der PCI-Express-Bus eingeführt. Dabei handelt es sich eigentlich nicht mehr um einen Bus, sondern um Punkt-zu-Punkt-Verbindungen über serielle Leitungen. Die Punkt-zu-Punkt-Verbindungen werden wie bei Netzwerken über zentrale Schalter (engl. *switches*) hergestellt (siehe Abb. 2.94).

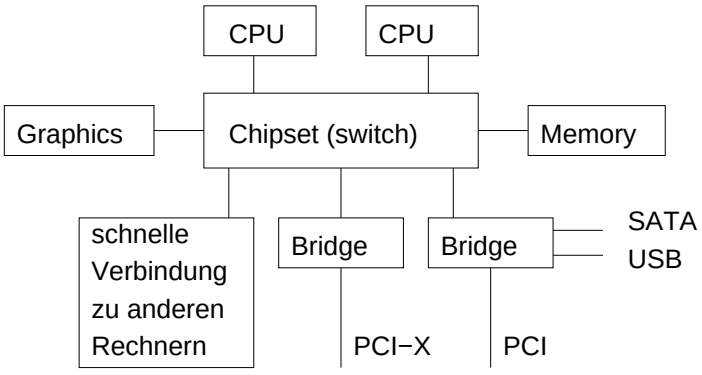


Abbildung 2.94: Topologie eines PCI-Express-Systems [Bha]

2.5.1.2 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. 2.95 a)).

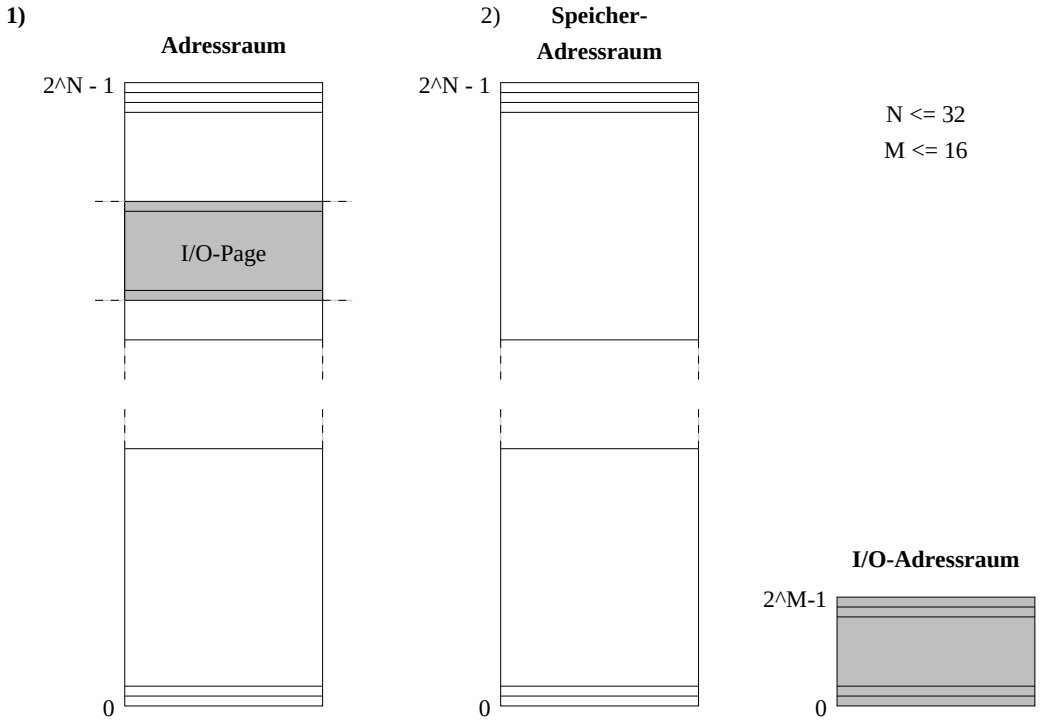


Abbildung 2.95: Adressierung von E/A-Geräten und Gerätesteuerungen

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. 2.95 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.

2.5.1.3 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 [Hay98].

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)

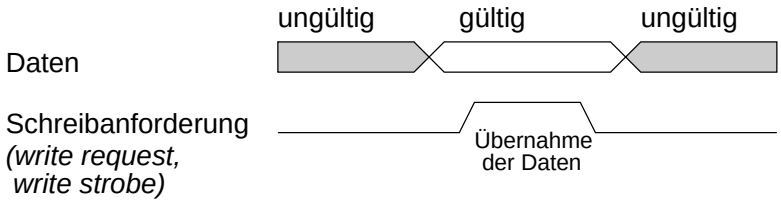


Abbildung 2.96: Schreiben beim unidirektionalen Timing

Die Anforderung wird vom Sender generiert (siehe Abb. 2.96) 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. 2.97). Ein *address strobe*-Signal zeigt an, dass gültige Adressen und eine gültige Busoperation vorhanden 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.

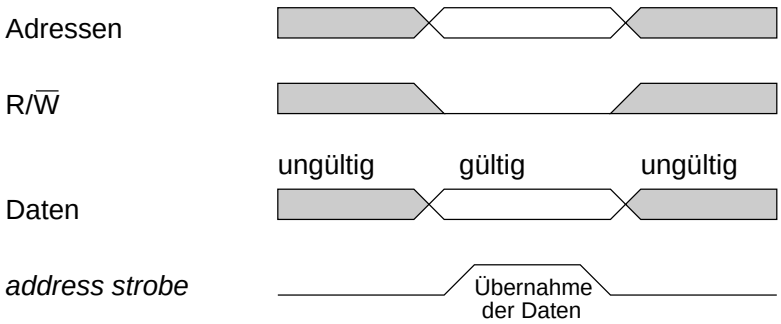


Abbildung 2.97: Schreiben beim synchronen Bus

In der Praxis werden manche der Leitungen (insbesondere *address strobe*) invertiert sein, was aber am Prinzip nichts ändert.

- **Lesen** (Initiierung durch Empfänger)

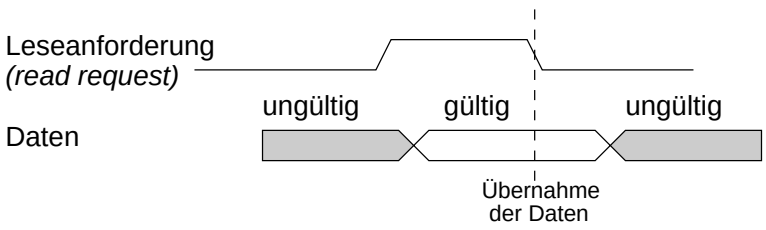


Abbildung 2.98: Lesen beim synchronen Bus

Die Anforderung wird vom Empfänger generiert und zeigt an, dass gelesen werden soll (siehe Abb. 2.98). 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. 2.99).

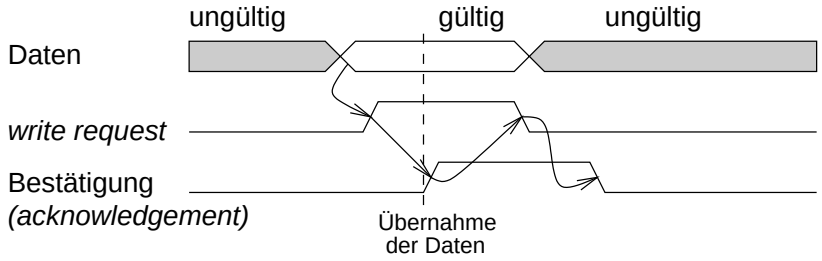


Abbildung 2.99: Schreiben beim bidirektionalen Timing

Bei Bussen werden wieder meist zusätzlich Adressenleitungen vorhanden sein (siehe Abb. 2.100). Adressen, Daten und das $R/\overline{W}$-Signal müssen gültig sein, solange *address strobe* =’1’ ist.

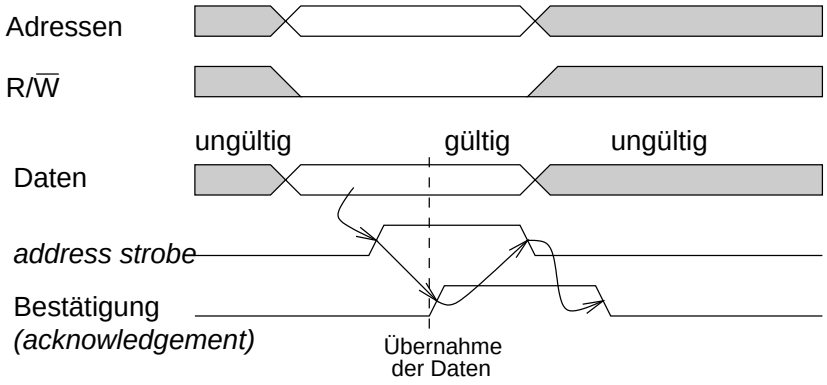


Abbildung 2.100: Schreiben beim asynchronen Bus

Mit Hilfe der Signale *address strobe* und *data acknowledge* werden letztlich **Automaten** innerhalb des Senders und des Empfängers synchronisert. Die entsprechenden Zustandsdiagramme zeigt Abb. 2.101. 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.

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 (siehe Vorlesung „Grundlagen der Theoretischen Informatik“) benutzt werden, die eine äquivalente Beschreibungsform von Automaten darstellen.

Bei Speichern muss die Bestätigung in der Regel außerhalb der eigentlichen Speicherbausteine erzeugt werden (siehe Abb. 2.102)

- **Lesen** (Initiierung durch Empfänger)
  Nach dem Senden der Anforderung kann noch eine große Zeit vergehen, bis der Sender die Bestätigung schickt (siehe Abb. 2.103).

  Die Daten müssen zumindest innerhalb des Zeitintervalls, in dem die Bestätigung = ’1’ ist, gültig sein.

  Bei Bussen werden meist zusätzlich Adressenleitungen vorhanden sein. Abb. 2.104 zeigt typische Signale in diesem Fall.

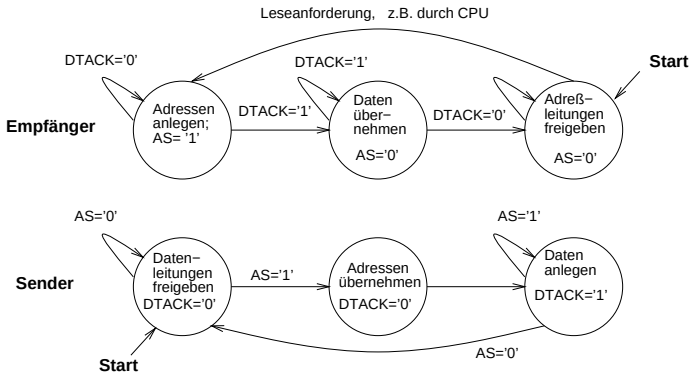


Abbildung 2.101: Zustandsdiagramm eines asynchronen Busses (AS=*address strobe*)

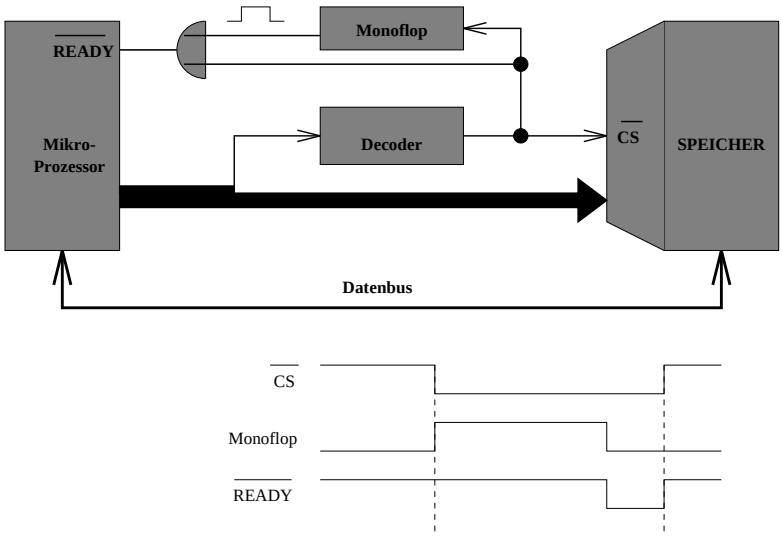


Abbildung 2.102: Erzeugung der Bestätigung bei Speicherbausteinen (READY=*acknowledge*)

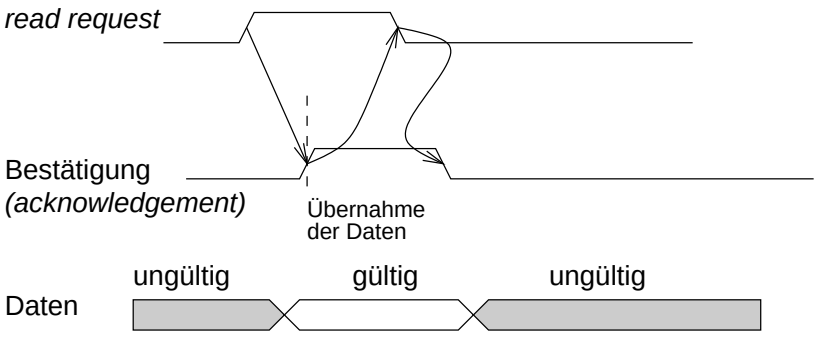


Abbildung 2.103: Lesen bei birektionalem Timing

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 2.16.

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.

Bähring [Bäh02b] 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 Ant-

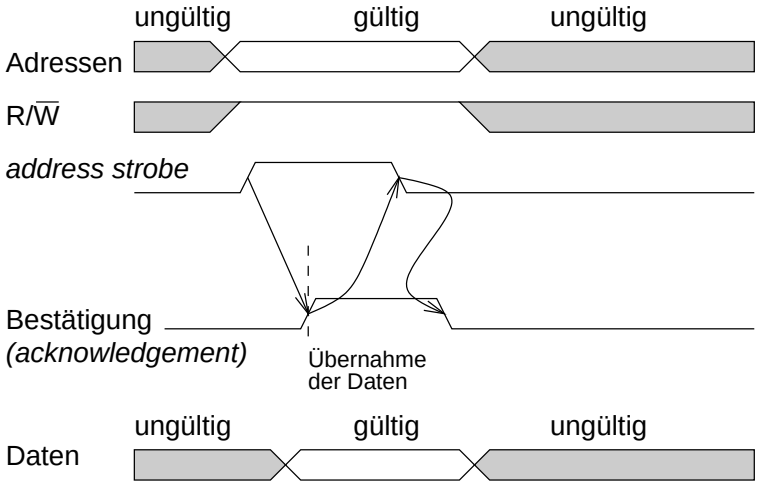


Abbildung 2.104: Lesen beim asynchronen Bus

	unidirektional	bidirektional
Vorteile	einfach; bei konstanten Antwort-zeiten 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 2.16: Unidirektionales und bidirektionales Timing

worten gewartet werden muss, ist die Übertragungsgeschwindigkeit wie beim asynchronen Bus durch die Laufzeit der *request*- und *acknowledge*-Signale begrenzt.

2.5.1.4 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.

2.5.1.4.1 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.

2.5.1.4.2 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 [Hay98]) :

Gegeben sei eine Gerätesteuerung, bei der unter Adresse 1 der gegenwärtige Status und unter Adresse 2 die aktuelle Eingabe eingelesen werden kann (siehe Abb. 2.105).

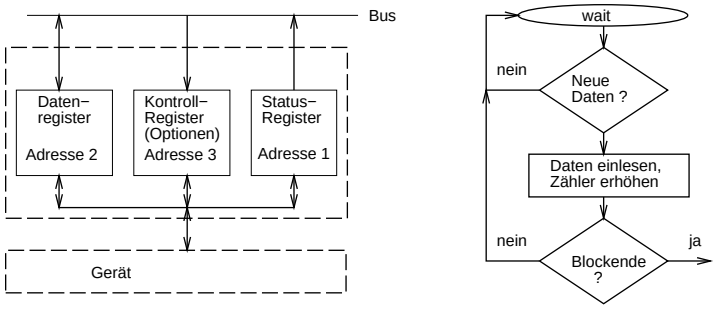


Abbildung 2.105: Realisierung von *busy waiting*

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[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)

Angewandt 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.

2.5.1.4.3 ***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 praktisch 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*).

2.5.1.4.4 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. 2.106):

- Prozess A überträgt E/A-Auftrag mittels SVC an Betriebssystem
- Dispatcher schaltet auf neuen Prozess um

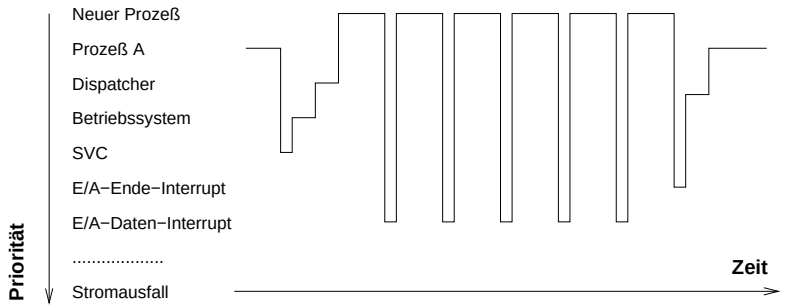


Abbildung 2.106: Ablauf bei der Interrupt-Verarbeitung

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.

2.5.1.4.5 *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. 2.107.

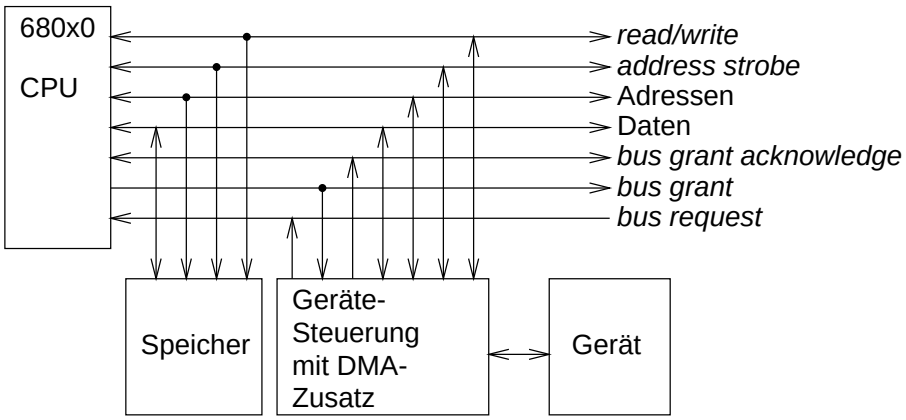


Abbildung 2.107: Bus des MC 680x0

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**.

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.

2.5.1.5 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**[26]).

[26] Siehe auch Abb. 5.2-1, 5.2-2, 5.2-3 des Buches von Bähring.

Es gibt verschiedene Methoden der Arbitrierung ([HP95], Abschn. 8.4):

1. ***Daisy Chaining***[27]:

   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[28] *... 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].

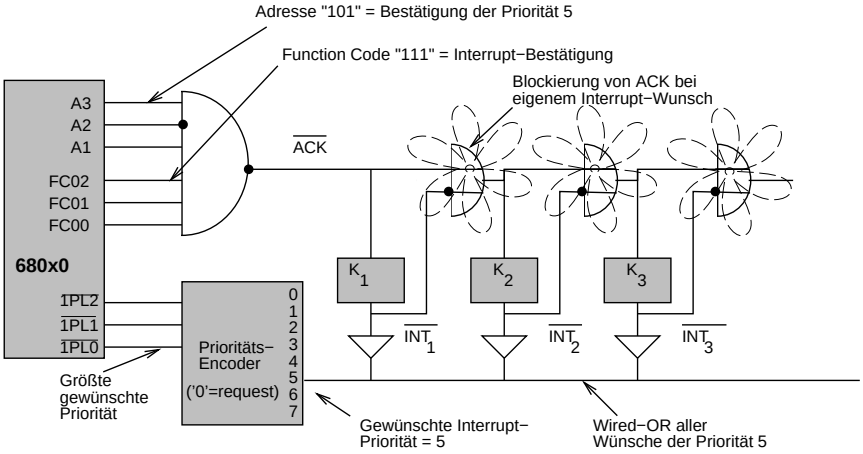


Abbildung 2.108: *daisy chaining* bei Motorola 68000er-Prozessoren

2. ***Centralized, parallel arbitration*** (siehe Abb. 2.109):

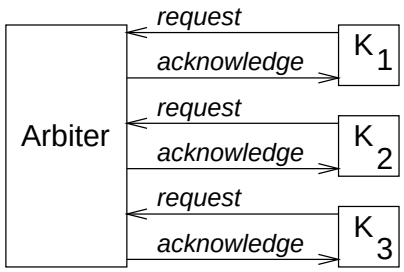


Abbildung 2.109: *Centralized parallel arbitration*

   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].

3. ***Distributed arbitration by self-selection*** (siehe Abb. 2.110):

   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. 2.110 zeigt eine mögliche Realisierung.

[27] Deutsch: Gänseblümchen-Ketten
[28] Siehe Abb. 2.108 [Bäh02a, Bäh02b].

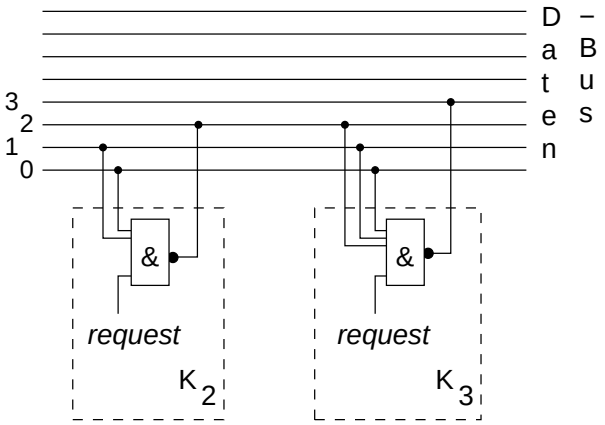


Abbildung 2.110: *Distributed arbitration by self-selection*

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***:

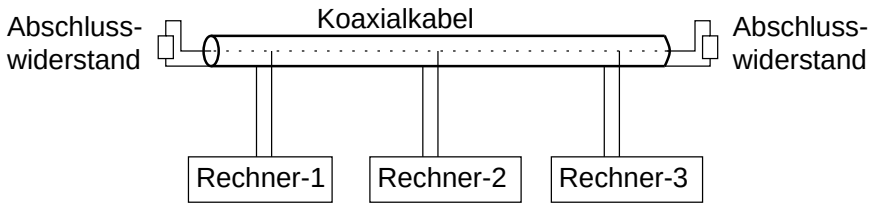


Abbildung 2.111: *Distributed arbitration by collision detection* beim Ethernet

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 109).

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.

6. **Zeitgesteuerte Arbitrierung**: Sofern eine Verarbeitung innerhalb einer bestimmten Zeit garantiert werden soll, ist eine zeitgesteuerte Zuordnung des Übertragungsmediums sinnvoll.

2.5.1.6 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 abwickeln 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.
- **Vollduplex- und Halb-Duplex-Betrieb:** Beim Voll-Duplex-Betrieb von Bussen (oder anderen Übertragungsmedien) können Informationen in beide Richtung gleichzeitig übertragen werden. 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.

2.5.1.7 Standardbusse

Es gibt eine Vielzahl von Standard-Bussen, die jeweils aufgrund bestimmter Zielvorstellungen und technischer Möglichkeiten definiert wurden [MSS91, HP08]. Wir werden hier zwischen Speicherbussen, lokalen Bussen und Peripheriebussen unterscheiden (vgl. Abb. 2.92). Tatsächlich kann ein bestimmter Standard mehreren Zwecken dienen und für einen bestimmten Zweck können aus Kompatibiltätsgründen mehrere Standards dienen.

2.5.1.7.1 Speicherbusse Bei der Darstellung beginnen wir mit den Speicherbussen (vgl. Abb. 2.92). Speicherbusse sind die schnellsten Busse überhaupt. Sie sind vielfach Hersteller-spezifische Busse. In der Regel besteht keine Notwendigkeit der Kompatibilität mit Bussen anderer Hersteller.

2.5.1.7.2 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.

- 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.

- 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). AGP ist im Grunde kein Bus, sondern ein Standard für eine Punkt-zu-Punkt-Verbindung.

- PCI-Express

  Über die *switches* des PCI-Express-Standards (siehe Abb. 2.94) werden wie beim Internet-Verkehr Pakete übertragen. Der PCI-Bus reicht auch für moderne Multimedia-Anforderungen nicht mehr aus: gewöhnliche Dateitransfer-Aufgaben können zeitkritische Multimedia-Transfers ausbremsen, bei denen sich Verzögerungen evtl. als Störungen in Bild oder Ton auswirken können. Man muss also zwischen Transfers unterschiedlicher Dringlichkeiten unterscheiden. Durch die Paket-basierte Übertragung sind *split transactions* realisiert. Für die Software soll die Änderung der physikalischen Transfers transparent sein, d.h. für PCI entwickelte Betriebssysteme sollen ohne Änderungen funktionieren (auch wenn vielleicht nicht alle Vorteile ausgenutzt werden können). Die Karten können eine unterschiedliche Anzahl von seriellen Leitungen (engl. *lanes*) nutzen. Während einfache Karten z.B. nur eine *lane* nutzen, werden Graphikkarten häufig 16 *lanes* verwenden. Den jeweils nutzbaren lanes entsprechen die Kontakte in den Steckplätzen bzw. Einschubkarten. Die unterschiedlichen Versionen erlauben jeweils eine Verdopplung der Datenrate (siehe www.pcisig.com).

Die Tabelle 2.17 enthält charakteristische Daten einer Reihe von lokalen Bussen.

	ISA-Bus	PCI-Bus (Std.)	AGP (4x)	PCI-Express 1.0	PCI-Express 3.0	
Einführungsjahr	1981			2004	2010	
Bustakt [Mhz]	4,77	33	266	2500	8000	
Transfer-Rate (Block-Mode) [MB/s]	9,54	111	1000	4000/Verbindung bei 16 *lanes*	15754/Verbindung bei 16 *lanes*	
synchron/asynchron	synchron	synchron		Paketvermittlung		
Datenleitungen	8/16	32 (64)		serielle Leitungen (*lanes*)		
Adressenleitungen	20	32 (64)		serielle Leitungen (*lanes*)		
Bemerkung	8086-Bus		Punkt-zu Punkt-Verb.	Punkt-zu Punkt-Verbindungen		

Tabelle 2.17: Daten einiger Busse

2.5.2 Peripherie-Schnittstellen

Daten müssen sowohl zu lokalen peripheren Geräten wie auch zwischen Rechnern ausgetauscht werden. Da die Techniken zum Anschluss von Peripherie-Geräten und zur Datenübertragung zu anderen Rechnern sich immer weiter angenähert haben, werden sie hier zusammenhängend behandelt. Generell werden bei der Übertragung über größere Entfernungen Informationen aus Aufwandsgründen nicht mehr wortparallel, sondern höchstens noch byteparallel, in der Regel aber bitseriell übertragen. Einige hierfür benutzte Standards sind nachfolgend beschrieben.

2.5.2.1 Asynchrone serielle Schnittstellen

Asynchrone serielle Schnittstellen stellen eine einfache Technik zur Realisierung von Kommunikation dar. Sie würden früher v.a. zum Anschluss von Terminals, aber auch zum Anschluss von Druckern, Fax-Geräten, Modems u.a.m. an einen Rechner benutzt. In vielen Geräten wurden externe Schnittstellen durch andere abgelöst. jedoch finden sich diese Schnittstellen aufgrund ihrer Einfachheit teilweise innerhalb von Geräten oder werden in Einzelfällen auch noch als externe Schnittstellen genutzt. Darüber hinaus sind mit diesen Schnittstellen Begriffe und Verfahrensweisen eingeführt worden, die auch auf andere Schnittstellen Anwendung finden. Es ist daher sinnvoll, auch weiterhin diese Schnittstellen zu behandeln.

Grundlage der üblichen asynchronen Übertragung ist das in Abb. 2.112 dargestellte Format.

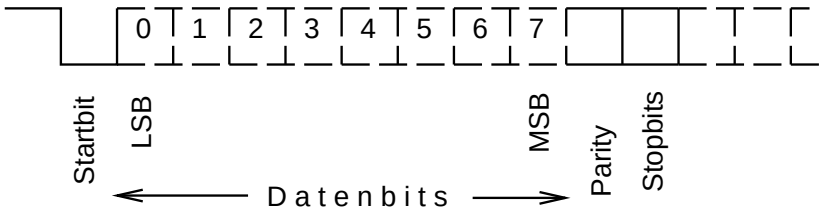


Abbildung 2.112: Zeichenrahmen bei der asynchronen Datenübertragung

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 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).

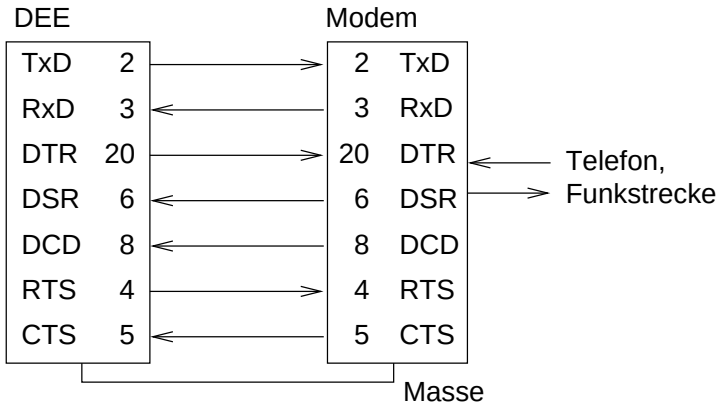


Abbildung 2.113: Anschluss einer DEE an ein Modem

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. 2.113 benötigt (Masseverbindungen nicht gezeigt).

In beide Richtungen können gleichzeitig über die Leitungen TxD und RxD Daten gesendet 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 zeigt der Kommunikationspartner dem DEE an, 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. 2.114 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.

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 man 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. 2.115 notwendig.

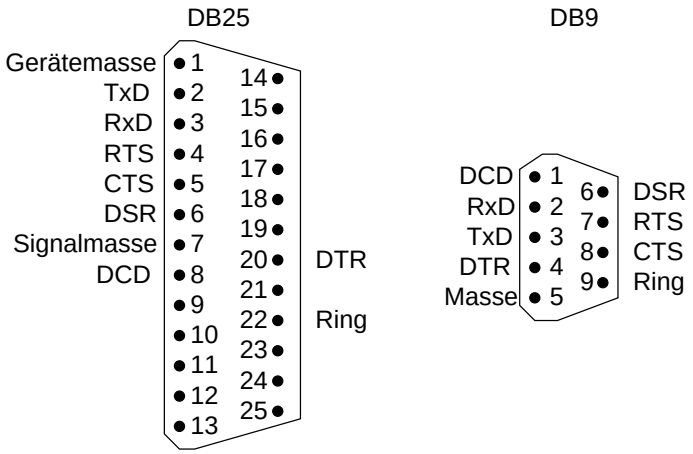


Abbildung 2.114: Steckerbelegung der V.24-Schnittstelle

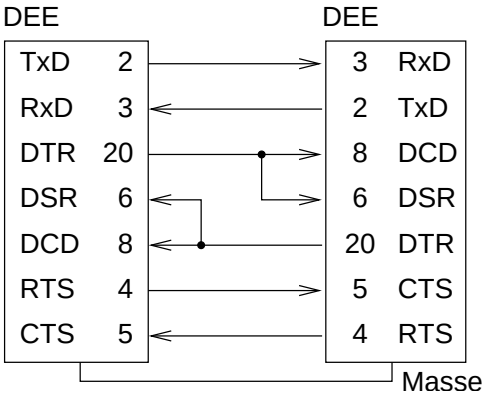


Abbildung 2.115: Verkabelung bei Anschluss zweier DEE einschließlich Hardware-Flusskontrolle

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. 2.116).

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.

2.5.2.2 Synchrone serielle Schnittstellen

Bei **synchronen seriellen Schnittstellen** wird die Synchronisation zwischen Sender und Empfänger nicht vor der Übertragung jedes Zeichens, sondern vor der Übertragung eines Blocks hergestellt. Dazu werden einem Block einige zwischen Sender und Empfänger verabredete Synchronisationszeichen vorangestellt. Abgeschlossen wird der Block ebenfalls durch einige verabredete Zeichen.

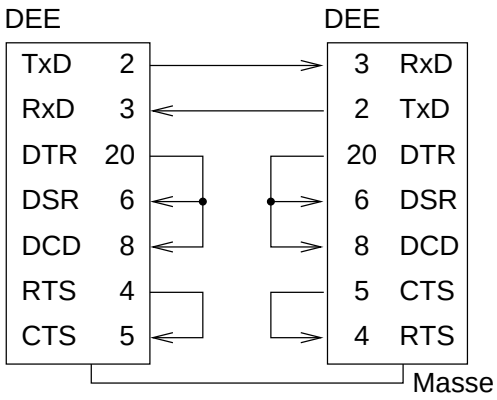


Abbildung 2.116: Verkabelung bei Anschluss zweier DEE ohne Hardware-Flusskontrolle

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. 2.117.

Abbildung 2.117: Zeichenorientierte Übertragung nach Bähring)

   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. 2.118.

Abbildung 2.118: Bitorientierte Übertragung (nach Bähring)

   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 bedingungslos. 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.

2.5.2.3 USB-Bus (*Universal serial bus)*

Der USB-Bus ist ein serieller Bus. Der USB-Standard basiert auf der Annahme, dass es einen Rechner (den **Wurzelknoten**) gibt, der alle Geräte kontrolliert. Aus diesem Grund gibt es den Steckertyp A, der eine Verbindung zum Rechner herstellen soll, sowie den Typ B, dessen zugehörige Buchse für Peripheriegeräte vorgesehen ist (siehe Abb. 2.119 [CHPI+00]).

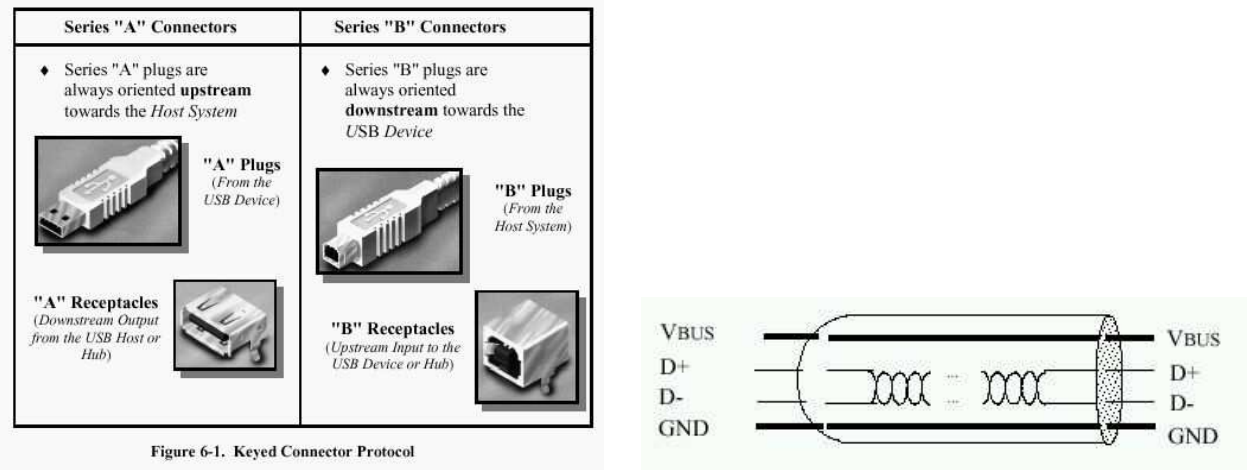


Abbildung 2.119: USB-Stecker und -Buchsen sowie der interne Aufbau der Kabel (nach [CHPI+00])

Intern enthalten die Kabel ein verdrilltes Leitungspaar für die (Halbduplex-) Datenübertragung sowie eine Masse- und eine Betriebsspannungsleitung. Über die Betriebsspannungsleitung können Geräte ohne eigene Stromversorgung betrieben werden. Der Strombedarf sollte mittels spezieller Datenpakete angemeldet werden, um eine Überlastung zu verhindern.

Zwecks Anschlusses einer großen Zahl von Geräten ist ein hierarchisch aufgebautes *Hub*-System vorgesehen, bei dem jeder Hub die Zahl der verfügbaren Anschlüsse vervielfacht.

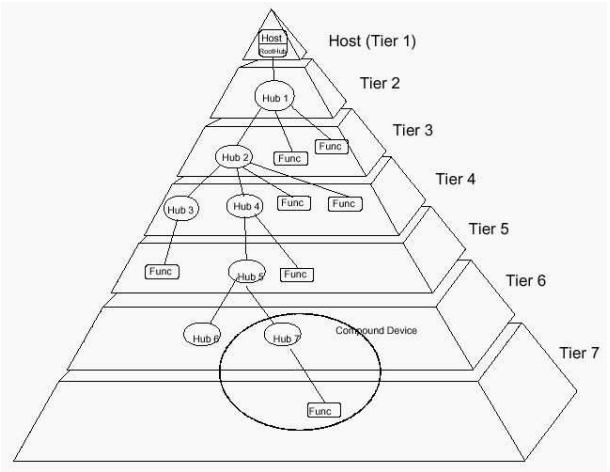


Abbildung 2.120: USB-Hierarchie (nach [CHPI+00])

Dabei darf eine maximale Anzahl von 5 hintereinander geschalteten Hubs und eine Länge von 5 Metern pro Hub nicht überschritten werden. Damit auch Laien die Längenbeschränkung von 5 Metern nicht verletzen, sollten eigentlich nur Kabel gefertigt werden, welche jeweils einen Stecker vom Typ A und einen Stecker vom Typ B besitzen. Verlängerungskabel, die einen Stecker vom Typ A und eine Buchse vom Typ A enthalten, die also als Verlängerungskabel geeignet sind, sollten eigentlich nicht hergestellt werden. In kurzen Längen sind sie dennoch im Handel. Solange nicht mehrere in Kombination benutzt werden, entsteht dadurch kein Problem.

Die Abfrage der verschiedenen Geräte erfolgt per Polling vom Wurzelknoten aus[29]. Die einzelnen Geräte können

[29] Die Beschränkung auf Polling scheint ab USB 3.0 aufgehoben zu sein.

mithin nicht selbst das Übertragungsmedium anfordern. Aus diesem Grund erfolgt ist USB im Prinzip zur Vernetzung verschiedener Rechner ungeeignet. Eine Verbindung von zwei Rechnern über ein Kabel mit zwei USB-A-Steckern würde damit aufgrund des gewählten Übertragungsprotokolls nicht funktionieren. Zusätzlich würde ein solches Kabel über die integrierte Stromversorgungsleitung die Spannungsversorgungen der beiden Rechner miteinander kurzschließen, was beide Rechner gefährden könnte. Aus diesem Grund soll es gemäß Standard solche Kabel nicht geben. Es gibt sie dennoch, aber von ihrem Einsatz muss dringend abgeraten werden. Sollen Rechner dennoch über USB miteinander vernetzt werden, so müssen so genannte USB-Brücken eingesetzt werden, die im Kabel selbst wieder einen Prozessor und Speicher enthalten. Von einem Rechner gesendete Daten werden in diesem Speicher zwischengepuffert, bis der andere Rechner sie mittels Polling abholt. Jede Verbindung zwischen zwei Rechnern muss einen solchen Speicher enthalten. Solche Lösungen bleiben teuer und verhältnismäßig langsam.

Der USB-Bus erreicht Datenraten von 1,5 MBit/s (für einfache Geräte wie z.B. Mäuse oder Tastaturen), 12 MBit/s und, 480 MBit/s (nur USB 2.0). Es werden drei verschiedene Übertragungsmodi angeboten:

- max 64 Byte, garantiert pro Zeitintervall
- *bulk transfer* (für *file-transfer*), nur wenn Bandbreite vorhanden;
- isochroner Transfer, zeitgenau, keine Fehlerbehandlung

Die Daten werden mittels NRZI-Kodierung übertragen. Ist eine ’1’ zu übertragen, so bleibt der Pegel erhalten. Ist eine ’0’ zu übertragen, so erfolgt ein Wechsel des Pegels. Ggf. müssen zusätzliche Nullen erzeugt werden, um aus einem eingehenden Datenstrom wieder die einzelnen Bits trotz leicht schwankender Übertragungsrate zu separieren und um den um Takt erzeugen zu können.

USB 3.0: Die Einführung von USB 3.0 (ab 2010) brachte folgende Veränderungen:

- Höhere Geschwindigkeit (bis 5 Gbit/s)
- Voll-Duplex
- Abkehr vom reinen Polling
- Mögliche Stromentnahme:
  - Unangemeldet 150 mA statt bisher 100mA
  - Angemeldet 900 mA statt bisher 500 mA
- Zusätzliche Kontakte für Duplexfähigkeit und als Masseanschluss
- Abwärtskompatibel: alte Stecker passen in neue Buchsen

2.5.2.4 *Serial ATA (SATA), eSATA*

SATA (serial attached ATA) ist die serielle Variante der älteren ATA-Schnittstelle zum Anschluss von Plattenlaufwerken. eSATA ist eine Variante davon zum Anschluss von externen Plattenlaufwerken mit verhältnismäßig hoher Übertragungskapazität.

2.5.2.5 *SAS*

SAS ist die serielle Variante der älteren SCSI-Schnittstelle zum Anschluss von Plattenlaufwerken.

2.5.2.6 FireWire (IEEE Std. 1394)

FireWire ist ein weiterer Standard für serielle Busse. Das Grundkonzept von FireWire geht zurück auf Ideen, welche die Firma Apple ab 1987 entwickelte. FireWire ist ursprünglich bis 400 MBit/s normiert (IEEE 1394a), ist aber inzwischen mit 800 MBits/s verfügbar und ist für Erweiterungen bis 1,6 GBit/s und 3,2 GBit/s vorgesehen (IEEE 1394b).

Selbst 4,9 Gbit/s sollen erreicht werden. FireWire erlaubt Isochron-Betrieb mit garantierten Übertragungsraten. Aufgrund dieser „Echtzeitfähigkeit" ist FireWire besonders zum Anschluss von Kameras und anderen Multimediageräten geeignet. FireWire bietet ebenfalls eine Stromversorgung für angeschlossene Geräte. FireWire-Hubs sind meist keine externen, separaten Geräte, sondern in den übrigen Geräten integriert. Der größte Unterschied zu USB liegt in der Tatsache, dass FireWire auch Verbindungen zwischen gleichberechtigten Geräten (so genannte *peer-to-peer*-Verbindungen) ermöglicht. Damit ist FireWire per Entwurf auch zum Aufbau kleiner Netze geeignet.

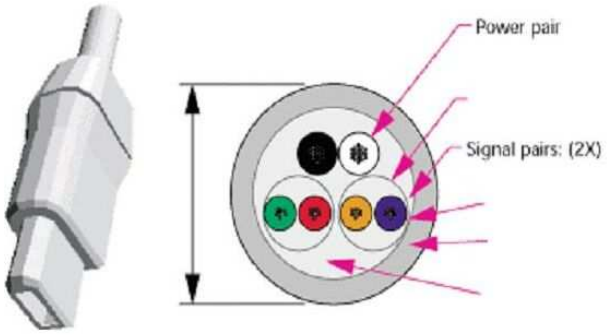


Abbildung 2.121: FireWire-Kabel (nach //www.1393ta.org)

2.5.2.7 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 er 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. 2.122).

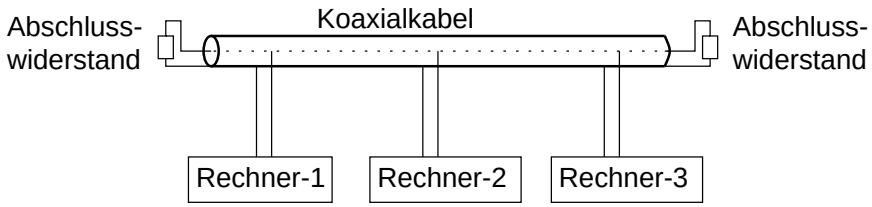


Abbildung 2.122: Physikalische Ethernet-Verbindung mit Abschlusswiderständen

  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. 2.123); Kabel wird an Anschlussstelle durchbohrt; Anschluss an AUI-Buchse des Rechners.

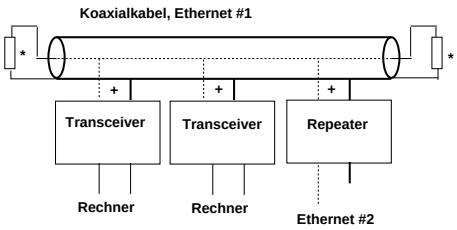


Abbildung 2.123: 10BASE5; +: kurze Verbindung zum Rechner; *: Abschlusswiderstand

– **10BASE2, ThinWire, Cheapernet:**
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. 2.124).

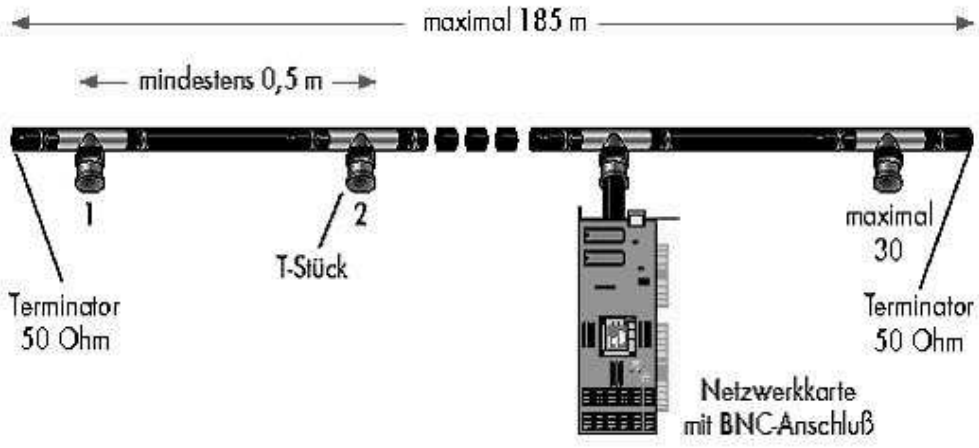


Abbildung 2.124: 10Base2: Detail des Rechneranschlusses (©c't)

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*) & STP-Kabel, Anschluss über RJ-45-Stecker (siehe Abb. 2.125).

– **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

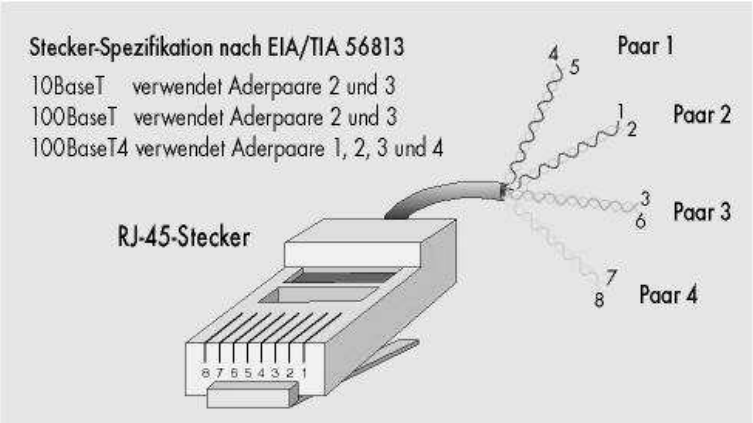


Abbildung 2.125: RJ-45-Stecker (©c't, 1999)

- **1000 Mbit/s, *(Gigabit Ethernet)***

  CSMA/CD erfordert, dass 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 2.126: Paketerweiterung (*extension*) beim Gigabit-Ethernet (©c't, 1999)

  – **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. 2.127) pro Verbindung.

Abbildung 2.127: *Twinax*- und *Quad*-Kabel (©c't, 1999)

  – **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.

- **10 Gbit-Ethernet**: Das kollisionsbehaftete CSMA/CD-Verfahren wird abgeschafft. 10 Gigabit Ethernet über Glasfaser ist im Gebrauch. Es löst andere Techniken für Weitverkehrsnetze ab. Lokal wird es als Backbone und zur Anbindung von *storage-area networks* (SANs) eingesetzt. Im Sommer 2006 wurde der IEEE-Standard für 10 Gigabit-Ethernet über Kupferkabel verabschiedet. Die Übertragungsstrecken werden auf ca. 100 m beschränkt sein. Alternativen sind derzeit:

  - Kat. 6-Kabel (mit bisherigen Steckern)
  - Kat. 7-Kabel (mit neuen Steckern)

- **40/100 Gbit-Ethernet**: 40/100 Gbit-Ethernet wurde 2010 standardisiert.

2.5.2.8 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. 2.128).

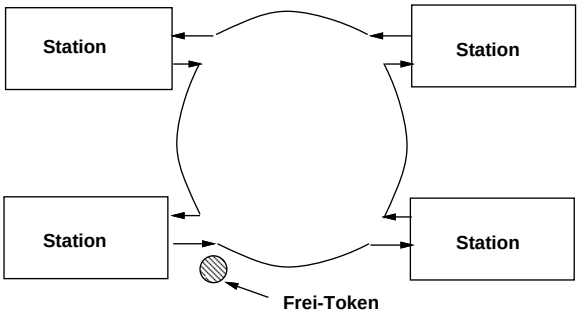


Abbildung 2.128: Ruhezustand

Stationen mit Sendewunsch wandeln ein „Frei-Token" in ein „Belegt-Token" um und fügen die Empfängeradresse sowie die zu übertragende Nachricht an (Abb. 2.129).

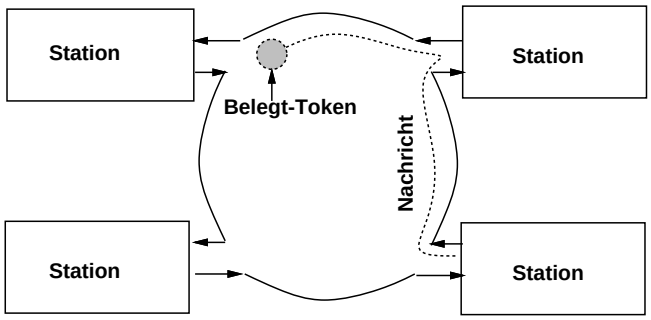


Abbildung 2.129: Erzeugung einer Nachricht

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. 2.130).

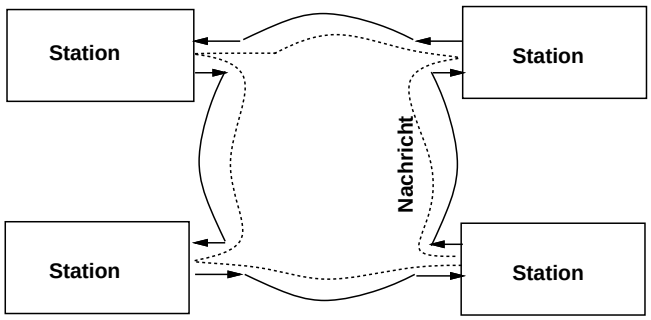


Abbildung 2.130: Vollständiger Umlauf

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 $\approx$ 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 [Sta07].

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.

2.5.3 Drahtlose lokale Netze (WLANs)

Drahtlose lokale Netzwerke gewinnen eine immer größere Bedeutung, insbesondere aufgrund der Diskussion *information anytime, anywhere* zur Verfügung zu stellen. Folgende Techniken werden derzeit eingesetzt:

- **Wireless LAN:** IEEE 802.11 a/b/g/n als Varianten der Netzwerk-Norm IEEE 802:

  Datenraten:

  - IEEE 802.11b: 1, 2, 5,5 oder 11 Mbit/s im 2,4 GHz-Band (meistens benutztes Band)
  - IEEE 802.11a: bis 54 Mbit/s im 5 GHz-Band
  - IEEE 802.11g: bis 54 MBits/s, abwärtskompatibel zu 802.11a und 802.11b.
  - IEEE 802.11n: bis 248 Mbit/s, 2,4 und 5 GHz. Benutzung mehrerer Antennen (MIMO) zur Erzielung einer Richtwirkung.

  Abb. 2.131 zeigt die Beziehungen zwischen Datenraten und Entfernungen bei typischen baulichen Gegebenheiten.

  Funkkanäle sind störanfälliger als drahtgebundene Kommunikation; aufgrund zusätzlicher Korrekturmaßnahmen in der Regel: effektive Datenrate WLAN $<$ Datenrate von 10BASE-T oder effektive Datenrate WLAN $<<$ Datenrate von 100BASE-Tx. (siehe auch c't 6/1999, S.222 ff)

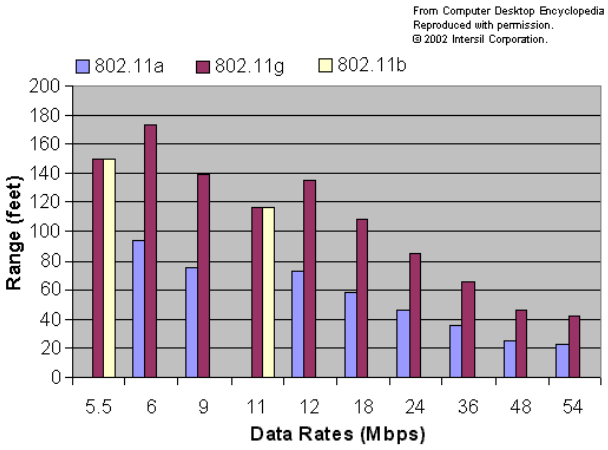


Abbildung 2.131: Datenraten verschiedener Varianten des Standards IEEE 802.11, ©Intersil, 2002

- **Bluetooth:** Benannt nach Harald Blauzahn (dänischer König von 940-985), der das Land einigte.

  Datenrate:

  - synchron: 64kbit/s für Sprachanwendung
  - asynchron symmetrisch: 432,6 kbit/s (Netzwerkanwendung)
  - asynchron asymmetrisch: 721/57,6 kbit/s downloads

  Reichweiten:

  - Pico bluetooth: 10cm..10m, 1mW Sendeleistung
  - Mega bluetooth: $>$ 100m, 100mW Sendeleistung

  Frequenzband: 2402-2480 GHz, lizenzfrei für industrielle, medizinische und wissenschaftliche Zwecke bereitgestellt; gemeinsam genutzt mit IEEE 802.11 und Mikrowellenöfen.

  Kanäle: 79, je 1 MHz; schneller Wechsel (nach jeder Nachricht) zwischen Kanälen, um die Wahrscheinlichkeit von Störungen zu reduzieren.

  Weitere Eigenschaften: Vollduplex-Übertragung, Spezielle Stromsparmodi, optionale Verschlüsselung, optionale Authentifizierung, verschiedene Profile ermöglichen Interoperabilität, Hardware relativ preisgünstig (Einbau z.B. in Köpfhörer).

  Weitere Informationen: http://www.bluetooth.com, c't 2/2003

- **DECT:** Die DECT-Technik ist eigentlich für Schnurlos-Telefone vorgesehen. Allerdings scheuen viele Anwender vor der Installation von zwei verschiedenen Funksystemen zurück, sodass Geräte, welche auf der Basis von DECT Daten übertragen, im Einsatz sind.

- **GSM, UMTS, HSDPA, LTE:** Für die Übertragung über größere Entfernungen sind die Mobilfunkstandards geeignet.

2.6 Rechner in Eingebetteten Systemen

2.6.1 Übersicht

Bis in die späten 80er Jahre war die Informationsverarbeitung mit großen *Mainframe*-Rechnern und riesigen Bandlaufwerken verbunden. Während der 90er Jahre hat sich die Informationsverarbeitung zu den *Personal Computern*, den PCs, verlagert. Dieser Trend zur Miniaturisierung geht weiter, und die Mehrzahl informationsverarbeitender Geräte werden in naher Zukunft kleine, teilweise auch tragbare Computer sein, die in größere Produkte integriert sind. Das Vorhandensein von Prozessoren in diesen umgebenden Produkten, wie z.B. in Telekommunikationsgeräten, wird weniger sichtbar sein als beim klassischen PC. Daher wird dieser Trend als der **verschwindende Computer** bezeichnet. Allerdings bedeutet dieser Begriff nicht, dass die Computer tatsächlich verschwinden werden, sondern vielmehr, dass sie überall sein werden. Diese neue Art von Anwendungen der Informationsverarbeitung wird auch *ubiquitous computing* (**allgegenwärtiges Rechnen**), *pervasive computing* [Han01], [Bur01] oder *ambient intelligence* [MA03] genannt. Diese drei Begriffe beschäftigen sich mit unterschiedlichen Nuancen der zukünftigen Informationsverarbeitung. *Ubiquitous Computing* konzentriert sich auf die langfristige Zielsetzung, Informationen jederzeit und überall zur Verfügung zu stellen, wohingegen *pervasive computing* sich mehr mit praktischen Aspekten, wie etwa der Ausnutzung bereits vorhandener Technologie, beschäftigt. Im Bereich *ambient intelligence* findet man einen Schwerpunkt auf der Kommunikationstechnologie im Wohnbereich der Zukunft sowie im Bereich der intelligenten Gebäudetechnik. Eingebettete Systeme sind einer der Ausgangspunkte dieser drei Gebiete und sie steuern einen Großteil der notwendigen Technologie bei.

Definition: Eingebettete Systeme[30] **sind informationsverarbeitende Systeme, die in ein größeres Produkt integriert sind**, und die normalerweise nicht direkt vom Benutzer wahrgenommen werden.

Basistechniken eingebetteter Systeme bilden zusammen mit der Kommunikationstechnik die Ausgangsbasis für die Realisierung von *ubiquitous computing* und *pervasive computing*.

Die folgende Liste beinhaltet Anwendungsbereiche, in denen eingebettete Systeme eingesetzt werden:

- **Automobilbereich:** Moderne Autos können nur noch verkauft werden, wenn sie einen beträchtliche Anteil an Elektronik enthalten, z.B. Airbag-Steuerungs-Systeme, elektronische Motorsteuerungen, ABS, Klimaanlagen, Navigationsgeräte mit GPS-Unterstützung und viele andere.
- **Bordelektronik im Flugzeug:** Einen Großteil des Gesamtwertes eines Flugzeugs machen heute die informationsverarbeitenden Systeme aus. Dazu gehören Flugkontrollsysteme, Anti-Kollisions-Systeme, Piloten-Informations-Systeme und andere. Die Verlässlichkeit der Systeme ist in diesem Bereich von allerhöchster Wichtigkeit.
- **Eisenbahntechnik:** Bei Zügen, Lokomotiven und stationären Sicherheitssystemen ist die Situation ähnlich wie bei Autos und Flugzeugen. Auch hier tragen die Sicherheitssysteme einen Großteil zum gesamten Wert bei, und die Verlässlichkeit hat eine ähnlich hohe Priorität.
- **Telekommunikation:** Die Verkaufszahlen von Handys sind in den vergangenen Jahren so stark gestiegen wie in kaum einem anderen Bereich. Schlüsselaspekte bei der Entwicklung von Handys sind das Beherrschen der Sendetechnik, digitale Signalverarbeitung und ein geringer Energieverbrauch.
- **Medizinische Systeme:** Im Bereich der medizinischen Geräte gibt es durch die Verwendung von informationsverarbeitenden Geräten ein großes Innovationspotential.
- **Militärische Anwendungen:** Informationsverarbeitung wird seit vielen Jahren in militärischen Ausrüstungsgegenständen verwendet. Eine der ersten Computeranwendungen war die automatische Auswertung von Radarsignalen.
- **Authentifizierungs-Systeme:** Eingebettete System können zur Benutzer-Authentifizierung verwendet werden. Beispielsweise können neuartige Zahlungssysteme die Sicherheit gegenüber klassischen Systemen deutlich erhöhen.

  Andere Authentifizierungssyteme sind etwa Fingerabdrucksensoren oder Gesichtserkennungs-Systeme.
- **Unterhaltungselektronik:** Video- und Audio-Geräte sind ein besonders wichtiger Sektor der Elektronikindustrie. Die Anzahl informationsverarbeitender Systeme auf diesem Gebiet erhöht sich ständig. Neue Dienste

[30] Dieser Abschnitt enthält u.a. Auszüge aus dem Buch „Eingebettete Systeme“ [Mar07].

und bessere Qualität werden durch moderne Methoden der digitalen Signalverarbeitung erreicht. Viele Fernseher, Multimedia-Handys und Spielekonsolen beinhalten Hochleistungsprozessoren und -Speichersysteme. Diese stellen eine besondere Gattung eingebetteter Systeme dar.

- **Fabriksteuerungen:** Im Bereich der Fabriksteuerungen werden eingebettete Systeme traditionell seit Jahrzehnten eingesetzt. Die Sicherheit solcher Systeme ist sehr wichtig, wohingegen der Energieverbrauch ein weniger wichtiges Kriterium ist.

- **Intelligente Gebäude:** Informationsverarbeitende Systeme können verwendet werden, um den Komfort in Gebäuden zu verbessern, deren Energieverbrauch zu verringern oder um die Sicherheit zu erhöhen. Vorher unabhängige Teilsysteme müssen zu diesem Zweck miteinander verbunden werden. Der Trend geht dahin, Klimaanlagen, Lichtsteuerungen, Zugangskontrollen, Abrechnungssysteme sowie Informationsverteilung und -bereitstellung in ein einziges System zu integrieren. So kann man z.B. Energie sparen, indem man Klimaanlage, Heizung und Beleuchtung herunterfährt, wenn die betreffenden Räume leer sind. Verfügbare Räume können an geeigneten Stellen angezeigt werden, wodurch sowohl die Suche nach einem Raum für ein spontanes Meeting als auch die Aufgabe der Reinigungskräfte vereinfacht wird. Der Geräuschpegel der Klimaanlage kann an die aktuelle Situation im Raum angepasst werden. Eine intelligente Steuerung der Jalousien kann die Beleuchtung und die Nutzung der Klimaanlage optimieren. Für leere Räume können größere Temperaturschwankungen akzeptiert werden, außerdem kann die Beleuchtung entsprechend reduziert werden. Im Notfall kann eine Liste der belegten Räume am Eingang des Gebäudes angezeigt werden (vorausgesetzt, der notwendige Strom steht noch zur Verfügung).

  Anfangs werden solche Systeme wohl hauptsächlich in modernen Bürogebäuden zu finden sein.

- **Robotik:** Im Bereich der Robotik werden eingebettete Systeme ebenfalls seit langem eingesetzt. Ein wichtiger Bereich auf diesem Gebiet ist die Mechanik. Viele der oben genannten Charakteristiken treffen auch auf die Robotik zu. Es gibt Roboter, die Tieren oder dem Menschen nachempfunden sind.

Die Größe des Marktes für eingebettete Systeme kann aus einer Reihe von Perspektiven analysiert werden. Wenn man z.B. die Anzahl der momentan im Betrieb befindlichen komplexen Prozessoren betrachtet, so wurde geschätzt, dass mehr als 90% dieser Prozessoren in eingebetteten Systemen verwendet werden. Viele dieser eingebetteten Prozessoren sind 8-Bit-Prozessoren, trotzdem sind 75% aller 32-Bit-Prozessoren ebenfalls in eingebettete Systeme integriert [Sti00]. Bereits 1996 wurde geschätzt, dass der durchschnittliche US-Amerikaner jeden Tag mit 60 Mikroprozessoren in Berührung kommt [CW96]. Diese Zahlen sind viel größer als man normalerweise annimmt, da vielen Menschen nicht bewusst ist, dass sie Prozessoren verwenden.

2.6.2 Herausforderungen

Der Entwurf eingebetteter Systeme stellt eine große Zahl von Herausforderungen. Unter diesen Herausforderungen befinden sich u.a.:

- Die **Verlässlichkeit:** Viele eingebettete Systeme sind sicherheitskritisch und müssen deshalb verlässlich arbeiten. Atomkraftwerke sind ein Beispiel für extrem sicherheitskritische Systeme, die zumindest zum Teil von Software gesteuert werden. Verlässlichkeit ist auch in anderen Systemen wichtig, so etwa in Autos, Zügen, Flugzeugen usw. Ein Hauptgrund, warum diese Systeme sicherheitskritisch sind, ist die Tatsache, dass sie direkt mit ihrer Umgebung in Verbindung stehen und einen unmittelbaren Einfluss auf diese Umgebung haben.

- Die **Realzeitfähigkeit:** Ein großer Teil der eingebetteten Systeme besteht aus Realzeitsystemen. Realzeitsysteme müssen nicht nur eine gültige Lösung berechnen, sie müssen dies auch in einer maximal erlaubten Zeit bewerkstelligen.

- Die **Energieeffizienz:** Viele eingebettete Systeme sind in tragbare Geräte integriert, die ihre Energie über Batterien beziehen. Nach Vorhersagen [ITR11] wird die Kapazität von Batterien nur langsam wachsen. Allerdings wachsen die Anforderungen an die Rechenkapazität, insbesondere für Multimedia-Anwendungen, sehr stark an, und die Kunden erwarten trotzdem lange Batterielaufzeiten. Daher muss die verfügbare elektrische Energie sehr effizient eingesetzt werden.

2.6.3 Hardware in the loop

In vielen eingebetteten Systemen, besonders in Regelungssystemen, wird eingebettete Hardware wie in Abb. 2.132 gezeigt in einer „Schleife“ verwendet (engl. *Hardware in the loop*).

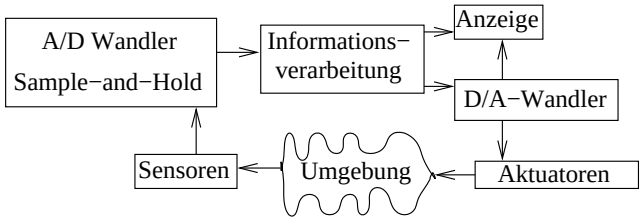


Abbildung 2.132: *Hardware in the loop*

In dieser Schleife werden Informationen aus der physikalischen Umgebung durch **Sensoren** verfügbar gemacht. Typischerweise erzeugen Sensoren kontinuierliche Folgen analoger Werte. Wir werden uns auf die Betrachtung von digitalen Systemen beschränken, die diskrete Folgen von Werten verarbeiten. Zwei Arten von Schaltungen führen diese Konvertierung durch: *Sample-and-hold*-Schaltungen und Analog-Digital-Wandler (A/D-Wandler). In der Regel benutzt man dabei zunächst *sample-and-hold*-Schaltungen, um von zeitkontinuierlichen Wertemengen auf zeitdiskrete Folgen von Werten überzugehen. Anschließend erfolgt dann eine Wertediskretisierung im A/D-Wandler. Nach der Umwandlung können die Informationen digital weiterverarbeitet werden. Die erzeugten Ergebnisse können z.B. angezeigt, aber auch zur Steuerung der physikalischen Umgebung verwendet werden. Letzteres geschieht mit der Hilfe von **Aktuatoren**. Da die meisten Aktuatoren analog sind, benötigt man in der Regel auch eine Digital-Analog-Wandlung.

Sensoren gibt es für die meisten physikalischen Größen und teilweise auch für andere Größen, wie beispielsweise chemische Stoffe. Die Zahl möglicher Sensoren ist so groß, dass sie hier nicht dargestellt werden kann.

Alle üblichen digitalen Computer arbeiten mit diskreten Zeitwerten[31]. Das bedeutet, dass sie diskrete Folgen von Werten verarbeiten können. Kontinuierliche Werte müssen folglich in den diskreten Wertebereich umgewandelt werden. Dies wird von sogenannten *Sample-and-Hold-Schaltungen* übernommen. Abbildung 2.133 (links) zeigt eine einfache *Sample-and-Hold*-Schaltung.

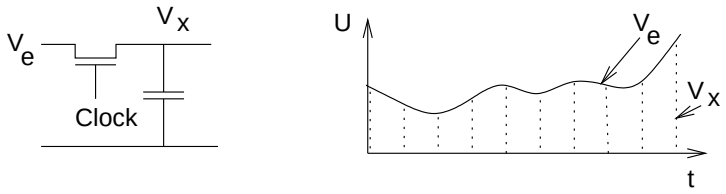


Abbildung 2.133: *Sample-and-Hold*-Schaltung

Im Wesentlichen besteht die Schaltung aus einem getakteten Transistor und einem Kondensator. Der Transistor verhält sich wie ein Schalter. Immer, wenn der Schalter durch das Taktsignal geschlossen wird, wird der Kondensator aufgeladen, so dass seine Spannung praktisch der Eingangsspannung V_e entspricht. Wenn der Schalter wieder geöffnet wird, bleibt diese Spannung so lange erhalten, bis der Schalter wieder geschlossen wird. Jede in dem Kondensator gespeicherte Spannung kann als ein Element der diskreten Folge von Werten V_x angesehen werden, die aus der kontinuierlichen Folge V_e erzeugt wurden (s. Abb. 2.133, rechts).

Eine ideale *Sample-and-Hold*-Schaltung wäre in der Lage, die Spannung des Kondensators in einer beliebig kurzen Zeit zu ändern. So könnte die Eingangsspannung zu einem bestimmten Zeitpunkt auf den Kondensator übertragen werden und jedes Element der diskreten Folge würde der Eingangsspannung zu einem bestimmten Zeitpunkt entsprechen. In der Praxis muss der Transistor allerdings für eine gewisse Zeit leitend bleiben, damit sich der Kondensator tatsächlich laden oder entladen kann.

Da wir uns auf digitale Computersysteme beschränken, muss nach der Diskretisierung der Zeiten auch eine Diskretisierung der Werte (auch **Quantisierung** genannt) vorgenommen werden. Diese Umwandlung von analogen zu digitalen Werten wird von A/D-Wandlern durchgeführt. Es gibt eine Vielzahl von A/D-Wandlern mit verschiedenen Geschwindigkeits- und Genauigkeits-Eigenschaften.

[31] Quantencomputer schließen wir hier aus.

Die Unterscheidung zwischen vielen digitalen Werten ist mit Hilfe eines A/D-Wandlers nach dem Prinzip der sukzessiven Approximation möglich. Die Schaltung ist in Abb. 2.134 gezeigt.

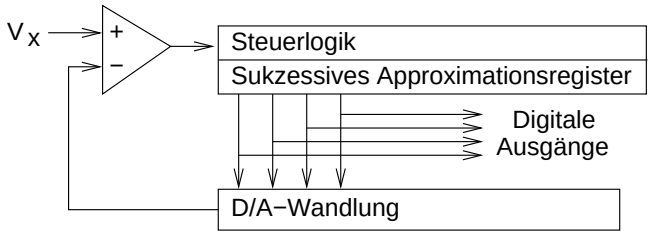


Abbildung 2.134: A/D-Wandlerschaltung mit sukzessiver Approximation

Die Grundidee dieser Schaltung basiert auf der binären Suche. Anfangs wird das höchstwertige Ausgabe-Bit auf '1' gesetzt, alle anderen Bits auf '0'. Dieser digitale Wert wird dann in einen analogen Wert umgewandelt, der $0,5\times$ der maximalen Eingangsspannung entspricht[32]. Wenn V_x diesen analogen Spannungswert übersteigt, bleibt das höchstwertige Bit auf '1', ansonsten wird es auf '0' zurückgesetzt.

Dieser Vorgang wird mit dem nächsten Bit wiederholt. Es bleibt auf '1', wenn die Eingangsspannung entweder im zweiten oder im vierten Viertel des Eingangsspannungsbereichs liegt. Dies wird mit allen weiteren Bits wiederholt.

Aufgrund der begrenzten Auflösung kann allerdings der ursprüngliche analoge Spannungswert nicht exakt digital dargestellt werden. Die Abweichung zwischen dem ursprünglichen analogen Wert und dem Wert, der durch die Digitalzahl dargestellt wird, heißt **Quantisierungsrauschen**.

Bei der Vielzahl von Klassen eingebetteter Systeme gibt es auch viele unterschiedliche Anforderungen an die Kommunikation. Im Allgemeinen ist das Verbinden von Hardwarekomponenten eingebetteter Systeme alles andere als einfach. Einige häufig vorkommende Anforderungen sind:

- **Echtzeit-Verhalten:** Diese Anforderung hat weitreichende Auswirkungen auf den Entwurf des Kommunikations-Systems. Einige kostengünstige Lösungen, wie etwa Ethernet, erfüllen diese Anforderung nicht.
- **Effizienz:** Die Verbindung zweier Hardwarekomponenten kann recht teuer sein. Beispielsweise ist eine direkte Punkt-zu-Punkt-Verbindung in einem großen Gebäude nahezu unmöglich. Im Automobilbereich hat sich gezeigt, dass separate Kabel, welche die Steuereinheiten mit der externen Peripherie verbinden, sehr teuer und schwer sind. Einzelne Kabel machen es auch schwierig, neue Komponenten anzuschließen. Die Kosten haben auch einen Einfluss auf den Entwurf der Stromversorgung externer Geräte. Häufig wird eine zentrale Stromversorgung eingesetzt, um Kosten zu sparen.
- **Angemessene Bandbreite und Kommunikations-Verzögerung:** Die geforderten Bandbreiten eingebetteter Systeme unterscheiden sich sehr stark. Die zur Verfügung stehende Bandbreite muss den Anforderungen genügen, ohne das System unnötig zu verteuern.
- **Unterstützung für Ereignis-gesteuerte Kommunikation:** Systeme, die auf dem regelmäßigen Abfragen von Geräten (sog. *Polling*) basieren, besitzen ein sehr gut vorhersagbares Echtzeitverhalten. Die Verzögerungen können bei dieser Art der Kommunikation allerdings zu groß sein. Oft wird eine schnelle, ereignisgesteuerte Kommunikation benötigt. Notfallbedingungen sollten beispielsweise sofort weitergeleitet werden und nicht solange unbemerkt bleiben, bis ein zentrales Systeme alle Geräte nach neuen Nachrichten abfragt.
- **Robustheit:** Eingebettete Systeme sollen bei extremen Temperaturen oder in der Nähe von Quellen elektromagnetischer Strahlung eingesetzt werden. Automotoren etwa sind Temperaturen von -20 bis zu +180 °C ausgesetzt. Spannungspegel und Taktfrequenzen können von solch hohen Temperaturschwankungen beeinflusst werden. Trotzdem muss eine verlässliche Kommunikation gewährleistet werden.
- **Fehlertoleranz:** Trotz aller Bemühungen, eine robuste Kommunikation zu erreichen, können Fehler auftreten. Eingebettete Systeme sollten auch nach einem solchen Fehler funktionsfähig bleiben. Neustarts mit wahrnehmbaren Verzögerungszeiten, wie man sie vom PC her gewohnt ist, sind inakzeptabel. Wenn eine Kommunikation also fehlgeschlagen ist, sind Wiederholungsversuche notwendig. Hier entsteht ein Konflikt mit der ersten

[32] Glücklicherweise kann die Umwandlung von digitalen zu analogen Werten (D/A-Wandlung) sehr effizient und schnell realisiert werden (s. Seite 119).

Anforderung: wenn man mehrere Kommunikationsversuche zulässt, ist es schwierig, das Echtzeitverhalten zu garantieren.

- **Wartbarkeit, Diagnosefähigkeit:** Es ist offensichtlich, dass man eingebettete Systeme in einer annehmbar kurzen Zeit reparieren können muss.
- **Verschlüsselung:** Um sicherzustellen, dass private und vertrauliche Informationen geheim bleiben, kann es notwendig sein, bei der Kommunikation Verschlüsselungstechniken einzusetzen.

Eine einfache Technik zur Verbesserung der elektrischen Robustheit ist die differenzielle Übertragung, die wir beispielsweise schon im Zusammenhang mit SCSI kennen gelernt haben.

Zur Informationsverarbeitung kann man derzeit v.a. drei Zieltechnologien einsetzen: spezielle anwendungsspezifische Schaltkreise (ASICs) (die sich nur bei sehr hohen Stückzahlen lohnen), konfigurierbare Hardware sowie Prozessoren. Unter den Prozessoren wird u.a. eine große Anzahl von **Mikrocontrollern** eingesetzt. Es sind dies einfache programmierbare Prozessoren, die für ihre Aufgaben in eingebetteten Systemen optimiert sind. In komplexeren Systemen befinden sich aber auch so genannte *multi-processor systems on a chip* (MPSoCs). Bei MPSoCs sind ganze Systeme auf einem Chip integriert.

D/A-Wandler sind nicht sehr kompliziert. Abbildung 2.135 zeigt den Schaltplan eines einfachen D/A-Wandlers.

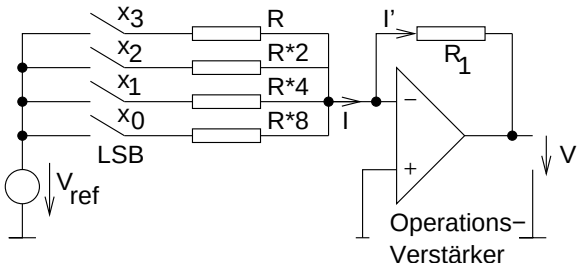


Abbildung 2.135: D/A-Wandler

Der Operationsverstärker in Abb. 2.135 verstärkt die Spannungsdifferenz zwischen seinen beiden Eingängen um einen sehr großen Faktor (mehrere Zehnerpotenzen)[33]. Über den Widerstand R_1 wird die entstehende Spannung wieder in den Eingang '–' eingespeist. Wenn zwischen den Eingängen eine kleine Spannungsdifferenz anliegt, wird sie invertiert, verstärkt und wieder an den Eingang gelegt, wodurch die Eingangsspannungdifferenz reduziert wird. Aufgrund des großen Verstärkungsfaktors wird die Spannung zwischen den Eingängen praktisch auf Null reduziert. Da der Eingang '+' mit Masse verbunden ist, ist die Spannung zwischen Eingang '–' und Masse praktisch gleich Null. Man sagt, der '–'-Eingang bilde eine virtuelle Masse. Eine virtuelle Masse kann man dabei als einen Leitungsknoten interpretieren, dessen Spannung in Bezug auf die Masseleitung praktisch gleich Null ist, der mit dieser Leitung aber nicht direkt verbunden ist.

Die Idee des D/A-Wandlers besteht nun darin, einen Strom zu erzeugen, der proportional zum Wert ist, der von dem Bitvektor x dargestellt wird. Dieser Strom wird dann in eine äquivalente Spannung umgewandelt.

Nach den Kirchhoffschen Gesetzen ist der Strom I hier die Summe aus den Strömen durch die Widerstände. Der Strom durch einen der Widerstände ist gleich Null, wenn das entsprechende Vektorelement von x gleich '0' ist. Wenn es dagegen '1' ist, entspricht der Strom wegen der entsprechend gewählten Widerstandswerte der Gewichtung dieses Bits.

$$\begin{aligned} I &= x_3 * \frac{V_{ref}}{R} + x_2 * \frac{V_{ref}}{2*R} + x_1 * \frac{V_{ref}}{4*R} + x_0 * \frac{V_{ref}}{8*R} \\ &= \frac{V_{ref}}{R} * \sum_{i=0}^{3} x_i * 2^{i-3} \end{aligned} \tag{2.15}$$

Nach Kirchhoff und wegen der virtuellen Masse am Eingang '–' gilt auch: $V + R_1 * I' = 0$.

[33] Die Folien enthalten hierzu weitere Erklärungen.

Der Strom, der in die Eingänge des Operationsverstärkers fließt, ist praktisch gleich Null. Somit sind die beiden Ströme I und I' identisch: $I = I'$. Daher:

$$V + R_1 * I = 0 \tag{2.16}$$

Aus den Gleichungen 2.15 und 2.16 ergibt sich:

$$-V = V_{ref} * \frac{R_1}{R} * \sum_{i=0}^{3} x_i * 2^{i-3} = V_{ref} * \frac{R_1}{8 * R} * nat(x) \tag{2.17}$$

nat ist die natürliche Zahl, die durch den Bitvektor *x* dargestellt wird. Offensichtlich ist die Spannung am Ausgang proportional zur durch *x* dargestellten Zahl. Positive Ausgangsspannungen und Bitvektoren, die Zahlen im Zweierkomplement darstellen, erfordern kleine Erweiterungen des D/A-Wandlers.

Aktuatoren gibt es wieder in so unterschiedlichen Ausführungen, dass hier auch kein repräsentativer Überblick gegeben werden kann.

2.6.4 Programmierung von Eingebetteten Systemen am Beispiel Lego Mindstorms

Für die Programmierung der Lego-Mindstorm-Roboter sei hier auf die Folien verwiesen.

Anhang A

Der SPIM-Simulator

SPIM S20: A MIPS R2000 Simulator

"$\frac{1}{25}^{th}$ the performance at none of the cost"

James R. Larus
larus@cs.wisc.edu
Computer Sciences Department
University of Wisconsin–Madison
1210 West Dayton Street
Madison, WI 53706, USA
608-262-9519

A.1 SPIM

SPIM S20[1] is a simulator[2] that runs programs for the MIPS R2000/R3000 RISC computers[3]. SPIM can read and immediately execute files containing assembly language. SPIM is a self-contained system for running these programs and contains a debugger and interface to a few operating system services.

The architecture of the MIPS computers is simple and regular, which makes it easy to learn and understand. The processor contains 32 general-purpose 32-bit registers and a well-designed instruction set that make it a propitious target for generating code in a compiler.

However, the obvious question is: why use a simulator when many people have workstations that contain a hardware, and hence significantly faster, implementation of this computer? One reason is that these workstations are not generally available. Another reason is that these machine will not persist for many years because of the rapid progress leading to new and faster computers. Unfortunately, the trend is to make computers faster by executing several instructions concurrently, which makes their architecture more difficult to understand and program. The MIPS architecture may be the epitome of a simple, clean RISC machine.

In addition, simulators can provide a better environment for low-level programming than an actual machine because they can detect more errors and provide more features than an actual computer. For example, SPIM has a X-window interface that is better than most debuggers for the actual machines.

Finally, simulators are an useful tool for studying computers and the programs that run on them. Because they are

[1] Dieses Kapitel wird im Skript eingefügt, obwohl wir den MARS-Simulator nutzen. Der Grund dafür ist, dass diese Beschreibung viele auch für die Simulation mit MARS nützliche Informationen enthält. Die idraw-Abbildungen des Originals wurden in das xfig-Format konvertiert.

[2] I (am) grateful to the many students at UW who used SPIM in their courses and happily found bugs in a professor's code. In particular, the students in CS536, Spring 1990, painfully found the last few bugs in an "already-debugged" simulator. I am grateful for their patience and persistence. Alan Yuen-wui Siow wrote the X-window interface.

[3] For a description of the real machines, see Gerry Kane and Joe Heinrich, *MIPS RISC Architecture*, Prentice Hall, 1992.

implemented in software, not silicon, they can be easily modified to add new instructions, build new systems such as multiprocessors, or simply to collect data.

A.1.1 Simulation of a Virtual Machine

The MIPS architecture, like that of most RISC computers, is difficult to program directly because of its delayed branches, delayed loads, and restricted address modes. This difficulty is tolerable since these computers were designed to be programmed in high-level languages and so present an interface designed for compilers, not programmers. A good part of the complexity results from delayed instructions. A *delayed branch* takes two cycles to execute. In the second cycle, the instruction immediately following the branch executes. This instruction can perform useful work that normally would have been done before the branch or it can be a `nop` (no operation). Similarly, *delayed loads* take two cycles so the instruction immediately following a load cannot use the value loaded from memory.

MIPS wisely choose to hide this complexity by implementing a *virtual machine* with their assembler. This virtual computer appears to have non-delayed branches and loads and a richer instruction set than the actual hardware. The assembler *reorganizes* (rearranges) instructions to fill the delay slots. It also simulates the additional, *pseudoinstructions* by generating short sequences of actual instructions.

By default, SPIM simulates the richer, virtual machine. It can also simulate the actual hardware. We will describe the virtual machine and only mention in passing features that do not belong to the actual hardware. In doing so, we are following the convention of MIPS assembly language programmers (and compilers), who routinely take advantage of the extended machine. Instructions marked with a dagger (†) are pseudoinstructions.

A.1.2 SPIM Interface

SPIM provides a simple terminal and a X-window interface. Both provide equivalent functionality, but the X interface is generally easier to use and more informative.

`spim`, the terminal version, and `xspim`, the X version, have the following command-line options:

`-bare`
: Simulate a bare MIPS machine without pseudoinstructions or the additional addressing modes provided by the assembler. Implies `-quiet`.

`-asm`
: Simulate the virtual MIPS machine provided by the assembler. This is the default.

`-pseudo`
: Accept pseudoinstructions in assembly code.

`-nopseudo`
: Do not accept pseudoinstructions in assembly code.

`-notrap`
: Do not load the standard trap handler. This trap handler has two functions that must be assumed by the user's program. First, it handles traps. When a trap occurs, SPIM jumps to location 0x80000080, which should contain code to service the exception. Second, this file contains startup code that invokes the routine `main`. Without the trap handler, execution begins at the instruction labeled `__start`.

`-trap`
: Load the standard trap handler. This is the default.

`-trap_file`
: Load the trap handler in the file.

`-noquiet`
: Print a message when an exception occurs. This is the default.

`-quiet`
: Do not print a message at an exception.

`-nomapped_io`
Disable the memory-mapped IO facility (see Section A.5).

`-mapped_io`
Enable the memory-mapped IO facility (see Section A.5). Programs that use SPIM syscalls (see Section A.1.5) to read from the terminal should not also use memory-mapped IO.

`-file`
Load and execute the assembly code in the file.

`-s seg size` Sets the initial size of memory segment *seg* to be *size* bytes. The memory segments are named: `text`, `data`, `stack`, `ktext`, and `kdata`. For example, the pair of arguments `-sdata 2000000` starts the user data segment at 2,000,000 bytes.

`-lseg size` Sets the limit on how large memory segment *seg* can grow to be *size* bytes. The memory segments that can grow are: `data`, `stack`, and `kdata`.

A.1.2.1 Terminal Interface

The terminal interface (`spim`) provides the following commands:

`exit`
Exit the simulator.

`read "file''`
Read *file* of assembly language commands into SPIM's memory. If the file has already been read into SPIM, the system should be cleared (see `reinitialize`, below) or global symbols will be multiply defined.

`load "file''`
Synonym for `read`.

`run <addr>`
Start running a program. If the optional address *addr* is provided, the program starts at that address. Otherwise, the program starts at the global symbol `__start`, which is defined by the default trap handler to call the routine at the global symbol `main` with the usual MIPS calling convention.

`step <N>`
Step the program for *N* (default: 1) instructions. Print instructions as they execute.

`continue`
Continue program execution without stepping.

`print $N`
Print register *N*.

`print $fN`
Print floating point register *N*.

`print addr`
Print the contents of memory at address *addr*.

`print_sym`
Print the contents of the symbol table, i.e., the addresses of the global (but not local) symbols.

`reinitialize`
Clear the memory and registers.

`breakpoint addr`
Set a breakpoint at address *addr*. *addr* can be either a memory address or symbolic label.

`delete addr`
Delete all breakpoints at address *addr*.

`list`
: List all breakpoints.

`.`
: Rest of line is an assembly instruction that is stored in memory.

`<nl>`
: A newline reexecutes previous command.

`?`
: Print a help message.

Most commands can be abbreviated to their unique prefix e.g., `ex`, `re`, `l`, `ru`, `s`, `p`. More dangerous commands, such as `reinitialize`, require a longer prefix.

A.1.2.2 X-Window Interface

The X version of SPIM, `xspim`, looks different, but should operate in the same manner as `spim`. The X window has five panes (see Figure A.1). The top pane displays the contents of the registers. It is continually updated, except while a program is running.

The next pane contains the buttons that control the simulator:

quit
Exit from the simulator.

load
Read a source file into memory.

run
Start the program running.

step
Single-step through a program.

clear
Reinitialize registers or memory.

set value
Set the value in a register or memory location.

print
Print the value in a register or memory location.

breakpoint
Set or delete a breakpoint or list all breakpoints.

help
Print a help message.

terminal
Raise or hide the console window.

mode
Set SPIM operating modes.

The next two panes display the memory contents. The top one shows instructions from the user and kernel text segments.[4] The first few instructions in the text segment are startup code (`__start`) that loads `argc` and `argv` into registers and invokes the `main` routine.

[4]These instructions are real—not pseudo—MIPS instructions. SPIM translates assembler pseudoinstructions to 1–3 MIPS instructions before storing the program in memory. Each source instruction appears as a comment on the first instruction to which it is translated.

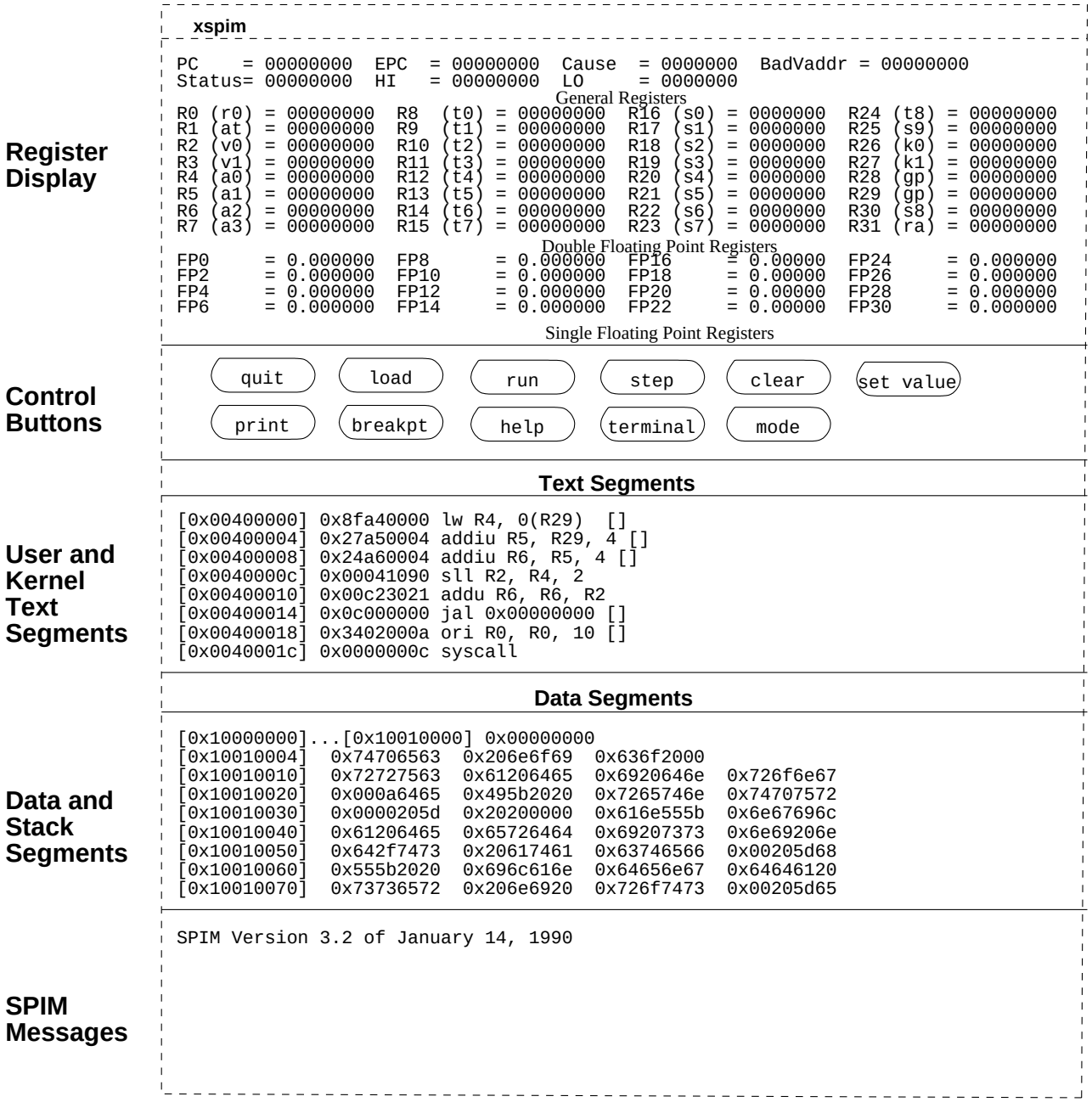


Abbildung A.1: X-window interface to SPIM.

The lower of these two panes displays the data and stack segments. Both panes are updated as a program executes.

The bottom pane is used to display messages from the simulator. It does not display output from an executing program. When a program reads or writes, its IO appears in a separate window, called the Console, which pops up when needed.

A.1.3 Surprising Features

Although SPIM faithfully simulates the MIPS computer, it is a simulator and certain things are not identical to the actual computer. The most obvious differences are that instruction timing and the memory systems are not identical. SPIM does not simulate caches or memory latency, nor does it accurate reflect the delays for floating point operations or multiplies and divides.

Another surprise (which occurs on the real machine as well) is that a pseudoinstruction expands into several machine instructions. When single-stepping or examining memory, the instructions that you see are slightly different from the source program. The correspondence between the two sets of instructions is fairly simple since SPIM does not reorganize the instructions to fill delay slots.

A.1.4 Assembler Syntax

Comments in assembler files begin with a sharp-sign (#). Everything from the sharp-sign to the end of the line is ignored.

Identifiers are a sequence of alphanumeric characters, underbars (_), and dots (.) that do not begin with a number. Opcodes for instructions are reserved words that are **not** valid identifiers. Labels are declared by putting them at the beginning of a line followed by a colon, for example:

```
       .data
item:  .word 1
       .text
       .globl main             # Must be global
main:  lw $t0, item
```

Strings are enclosed in double-quotes ("). Special characters in strings follow the C convention:

```
   newline        \n
   tab            \t
   quote          \"
```

SPIM supports a subset of the assembler directives provided by the MIPS assembler:

`.align n`
: Align the next datum on a 2^n byte boundary. For example, `.align 2` aligns the next value on a word boundary. `.align 0` turns off automatic alignment of `.half`, `.word`, `.float`, and `.double` directives until the next `.data` or `.kdata` directive.

`.ascii str`
: Store the string in memory, but do not null-terminate it.

`.asciiz str`
: Store the string in memory and null-terminate it.

`.byte b1, ..., bn`
: Store the *n* values in successive bytes of memory.

`.data <addr>`
: The following data items should be stored in the data segment. If the optional argument *addr* is present, the items are stored beginning at address *addr*.

Service	System Call Code	Arguments	Result
print_int	1	`$a0` = integer	
print_float	2	`$f12` = float	
print_double	3	`$f12` = double	
print_string	4	`$a0` = string	
read_int	5		integer (in `$v0`)
read_float	6		float (in `$f0`)
read_double	7		double (in `$f0`)
read_string	8	`$a0` = buffer, `$a1` = length	
sbrk	9	`$a0` = amount	address (in `$v0`)
exit	10		

Tabelle A.1: System services.

`.double d1, ..., dn`
Store the *n* floating point double precision numbers in successive memory locations.

`.extern sym size`
Declare that the datum stored at `sym` is `size` bytes large and is a global symbol. This directive enables the assembler to store the datum in a portion of the data segment that is efficiently accessed via register `$gp`.

`.float f1, ..., fn`
Store the *n* floating point single precision numbers in successive memory locations.

`.globl sym`
Declare that symbol `sym` is global and can be referenced from other files.

`.half h1, ..., hn`
Store the *n* 16-bit quantities in successive memory halfwords.

`.kdata <addr>`
The following data items should be stored in the kernel data segment. If the optional argument *addr* is present, the items are stored beginning at address *addr*.

`.ktext <addr>`
The next items are put in the kernel text segment. In SPIM, these items may only be instructions or words (see the `.word` directive below). If the optional argument *addr* is present, the items are stored beginning at address *addr*.

`.space n`
Allocate *n* bytes of space in the current segment (which must be the data segment in SPIM).

`.text <addr>`
The next items are put in the user text segment. In SPIM, these items may only be instructions or words (see the `.word` directive below). If the optional argument *addr* is present, the items are stored beginning at address *addr*.

`.word w1, ..., wn`
Store the *n* 32-bit quantities in successive memory words.

SPIM does not distinguish various parts of the data segment (`.data`, `.rdata`, and `.sdata`).

A.1.5 System Calls

SPIM provides a small set of operating-system-like services through the system call (`syscall`) instruction. To request a service, a program loads the system call code (see Table A.1) into register `$v0` and the arguments into registers `$a0`...`$a3` (or `$f12` for floating point values). System calls that return values put their result in register `$v0` (or `$f0` for floating point results). For example, to print "`the answer = 5`", use the commands:

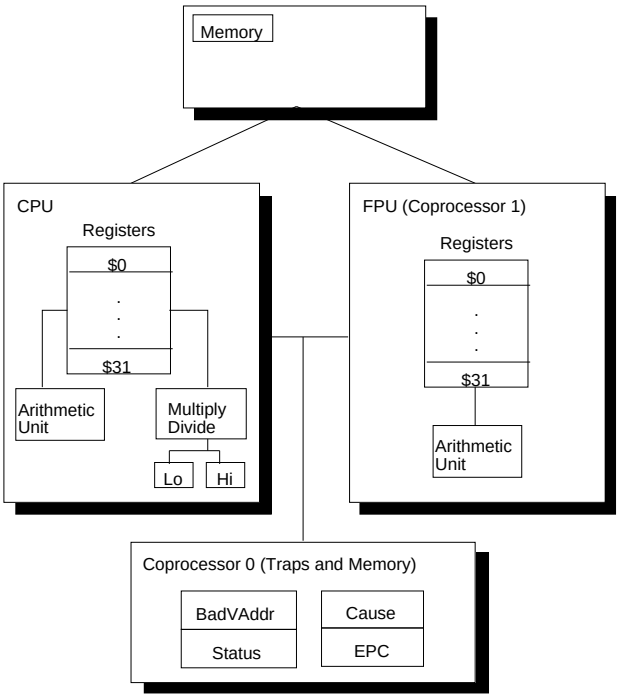


Abbildung A.2: MIPS R2000 CPU and FPU

```
        .data
str:    .asciiz "the answer = "
        .text
        li $v0, 4        # system call code for print_str
        la $a0, str      # address of string to print
        syscall          # print the string

        li $v0, 1        # system call code for print_int
        li $a0, 5        # integer to print
        syscall          # print it
```

`print_int` is passed an integer and prints it on the console. `print_float` prints a single floating point number. `print_double` prints a double precision number. `print_string` is passed a pointer to a null-terminated string, which it writes to the console.

`read_int`, `read_float`, and `read_double` read an entire line of input up to and including the newline. Characters following the number are ignored. `read_string` has the same semantics as the Unix library routine `fgets`. It reads up to $n-1$ characters into a buffer and terminates the string with a null byte. If there are fewer characters on the current line, it reads through the newline and again null-terminates the string. **Warning:** programs that use these syscalls to read from the terminal should not use memory-mapped IO (see Section A.5).

`sbrk` returns a pointer to a block of memory containing *n* additional bytes. `exit` stops a program from running.

A.2 Description of the MIPS R2000

A MIPS processor consists of an integer processing unit (the CPU) and a collection of coprocessors that perform ancillary tasks or operate on other types of data such as floating point numbers (see Figure A.2). SPIM simulates two coprocessors. Coprocessor 0 handles traps, exceptions, and the virtual memory system. SPIM simulates most of the first two and entirely omits details of the memory system. Coprocessor 1 is the floating point unit. SPIM simulates most aspects of this unit.

A.2.1 CPU Registers

Register Name	Number	Usage
zero	0	Constant 0
at	1	Reserved for assembler
v0	2	Expression evaluation and
v1	3	results of a function
a0	4	Argument 1
a1	5	Argument 2
a2	6	Argument 3
a3	7	Argument 4
t0	8	Temporary (not preserved across call)
t1	9	Temporary (not preserved across call)
t2	10	Temporary (not preserved across call)
t3	11	Temporary (not preserved across call)
t4	12	Temporary (not preserved across call)
t5	13	Temporary (not preserved across call)
t6	14	Temporary (not preserved across call)
t7	15	Temporary (not preserved across call)
s0	16	Saved temporary (preserved across call)
s1	17	Saved temporary (preserved across call)
s2	18	Saved temporary (preserved across call)
s3	19	Saved temporary (preserved across call)
s4	20	Saved temporary (preserved across call)
s5	21	Saved temporary (preserved across call)
s6	22	Saved temporary (preserved across call)
s7	23	Saved temporary (preserved across call)
t8	24	Temporary (not preserved across call)
t9	25	Temporary (not preserved across call)
k0	26	Reserved for OS kernel
k1	27	Reserved for OS kernel
gp	28	Pointer to global area
sp	29	Stack pointer
fp	30	Frame pointer
ra	31	Return address (used by function call)

Tabelle A.2: MIPS registers and the convention governing their use.

The MIPS (and SPIM) central processing unit contains 32 general purpose 32-bit registers that are numbered 0–31 (see table A.2). Register *n* is designated by `$n`. Register `$0` always contains the hardwired value 0. MIPS has established a set of conventions as to how registers should be used. These suggestions are guidelines, which are not enforced by the hardware. However a program that violates them will not work properly with other software. Table A.2 lists the registers and describes their intended use.

Registers `$at` (1), `$k0` (26), and `$k1` (27) are reserved for use by the assembler and operating system.

Registers `$a0`–`$a3` (4–7) are used to pass the first four arguments to routines (remaining arguments are passed on the stack). Registers `$v0` and `$v1` (2, 3) are used to return values from functions. Registers `$t0`–`$t9` (8–15, 24, 25) are caller-saved registers used for temporary quantities that do not need to be preserved across calls. Registers `$s0`–`$s7` (16–23) are callee-saved registers that hold long-lived values that should be preserved across calls.

Register `$sp` (29) is the stack pointer, which points to the last location in use on the stack.[5] Register `$fp` (30) is the frame pointer.[6] Register `$ra` (31) is written with the return address for a call by the `jal` instruction.

Register `$gp` (28) is a global pointer that points into the middle of a 64K block of memory in the heap that holds constants and global variables. The objects in this heap can be quickly accessed with a single load or store instruction.

In addition, coprocessor 0 contains registers that are useful to handle exceptions. SPIM does not implement all of these registers, since they are not of much use in a simulator or are part of the memory system, which is not implemented.

[5] In earlier version of SPIM, `$sp` was documented as pointing at the first free word on the stack (not the last word of the stack frame). Recent MIPS documents have made it clear that this was an error. Both conventions work equally well, but we choose to follow the real system.

[6] The MIPS compiler does not use a frame pointer, so this register is used as callee-saved register `$s8`.

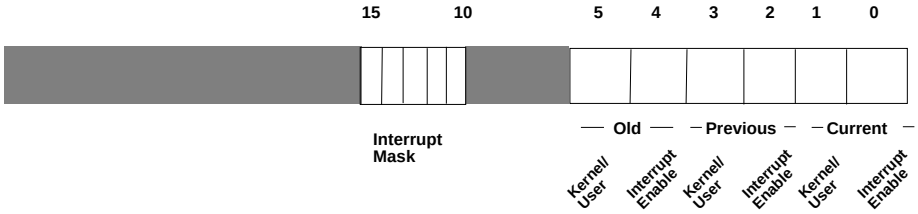


Abbildung A.3: The `Status` register.

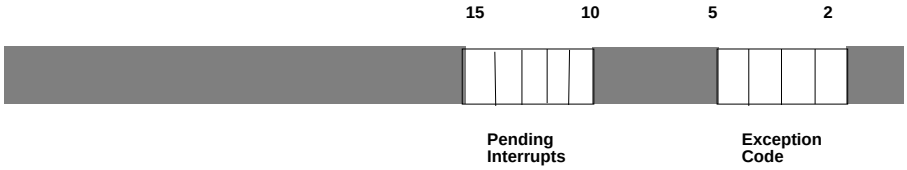


Abbildung A.4: The `Cause` register.

However, it does provide the following:

Register Name	Number	Usage
BadVAddr	8	Memory address at which address exception occurred
Status	12	Interrupt mask and enable bits
Cause	13	Exception type and pending interrupt bits
EPC	14	Address of instruction that caused exception

These registers are part of coprocessor 0's register set and are accessed by the `lwc0`, `mfc0`, `mtc0`, and `swc0` instructions.

Figure A.3 describes the bits in the `Status` register that are implemented by SPIM. The `interrupt mask` contains a bit for each of the five interrupt levels. If a bit is one, interrupts at that level are allowed. If the bit is zero, interrupts at that level are disabled. The low six bits of the `Status` register implement a three-level stack for the `kernel/user` and `interrupt enable` bits. The `kernel/user` bit is 0 if the program was running in the kernel when the interrupt occurred and 1 if it was in user mode. If the `interrupt enable` bit is 1, interrupts are allowed. If it is 0, they are disabled. At an interrupt, these six bits are shifted left by two bits, so the current bits become the previous bits and the previous bits become the old bits. The current bits are both set to 0 (i.e., kernel mode with interrupts disabled).

Figure A.4 describes the bits in the `Cause` registers. The five `pending interrupt` bits correspond to the five interrupt levels. A bit becomes 1 when an interrupt at its level has occurred but has not been serviced. The `exception code` register contains a code from the following table describing the cause of an exception.

Number	Name	Description
0	INT	External interrupt
4	ADDRL	Address error exception (load or instruction fetch)
5	ADDRS	Address error exception (store)
6	IBUS	Bus error on instruction fetch
7	DBUS	Bus error on data load or store
8	SYSCALL	Syscall exception
9	BKPT	Breakpoint exception
10	RI	Reserved instruction exception
12	OVF	Arithmetic overflow exception

A.2.2 Byte Order

Processors can number the bytes within a word to make the byte with the lowest number either the leftmost or rightmost one. The convention used by a machine is its *byte order*. MIPS processors can operate with either *big-endian* byte order:

Byte #			
0	1	2	3

or *little-endian* byte order:

Byte #			
3	2	1	0

SPIM operates with both byte orders. SPIM's byte order is determined by the byte order of the underlying hardware running the simulator. On a DECstation 3100, SPIM is little-endian, while on a HP Bobcat, Sun 4 or PC/RT, SPIM is big-endian.

A.2.3 Addressing Modes

MIPS is a load/store architecture, which means that only load and store instructions access memory. Computation instructions operate only on values in registers. The bare machine provides only one memory addressing mode: `c(rx)`, which uses the sum of the immediate (integer) `c` and the contents of register `rx` as the address. The virtual machine provides the following addressing modes for load and store instructions:

Format	Address Computation
(register)	contents of register
imm	immediate
imm (register)	immediate + contents of register
symbol	address of symbol
symbol $\pm$ imm	address of symbol $+$ or $-$ immediate
symbol $\pm$ imm (register)	address of symbol $+$ or $-$ immediate + contents of register

Most load and store instructions operate only on aligned data. A quantity is *aligned* if its memory address is a multiple of its size in bytes. Therefore, a halfword object must be stored at even addresses and a full word object must be stored at addresses that are a multiple of 4. However, MIPS provides some instructions for manipulating unaligned data.

A.2.4 Arithmetic and Logical Instructions

In all instructions below, `Src2` can either be a register or an immediate value (a 16 bit integer). The immediate forms of the instructions are only included for reference. The assembler will translate the more general form of an instruction (e.g., `add`) into the immediate form (e.g., `addi`) if the second argument is constant.

`abs Rdest, Rsrc` *Absolute Value* †
Put the absolute value of the integer from register `Rsrc` in register `Rdest`.

`add Rdest, Rsrc1, Src2` *Addition (with overflow)*
`addi Rdest, Rsrc1, Imm` *Addition Immediate (with overflow)*
`addu Rdest, Rsrc1, Src2` *Addition (without overflow)*
`addiu Rdest, Rsrc1, Imm` *Addition Immediate (without overflow)*
Put the sum of the integers from register `Rsrc1` and `Src2` (or `Imm`) into register `Rdest`.

`and Rdest, Rsrc1, Src2` *AND*
`andi Rdest, Rsrc1, Imm` *AND Immediate*
Put the logical AND of the integers from register `Rsrc1` and `Src2` (or `Imm`) into register `Rdest`.

`div Rsrc1, Rsrc2` *Divide (signed)*
`divu Rsrc1, Rsrc2` *Divide (unsigned)*
Divide the contents of the two registers. `divu` treats is operands as unsigned values. Leave the quotient in register

`lo` and the remainder in register `hi`. Note that if an operand is negative, the remainder is unspecified by the MIPS architecture and depends on the conventions of the machine on which SPIM is run.

`div Rdest, Rsrc1, Src2` *Divide (signed, with overflow)* †
`divu Rdest, Rsrc1, Src2` *Divide (unsigned, without overflow)* †
Put the quotient of the integers from register `Rsrc1` and `Src2` into register `Rdest`. `divu` treats is operands as unsigned values.

`mul Rdest, Rsrc1, Src2` *Multiply (without overflow)* †
`mulo Rdest, Rsrc1, Src2` *Multiply (with overflow)* †

`mulou Rdest, Rsrc1, Src2` *Unsigned Multiply (with overflow)* †
Put the product of the integers from register `Rsrc1` and `Src2` into register `Rdest`.

`mult Rsrc1, Rsrc2` *Multiply*
`multu Rsrc1, Rsrc2` *Unsigned Multiply*
Multiply the contents of the two registers. Leave the low-order word of the product in register `lo` and the high-word in register `hi`.

`neg Rdest, Rsrc` *Negate Value (with overflow)* †
`negu Rdest, Rsrc` *Negate Value (without overflow)* †
Put the negative of the integer from register `Rsrc` into register `Rdest`.

`nor Rdest, Rsrc1, Src2` *NOR*
Put the logical NOR of the integers from register `Rsrc1` and `Src2` into register `Rdest`.

`not Rdest, Rsrc` *NOT* †
Put the bitwise logical negation of the integer from register `Rsrc` into register `Rdest`.

`or Rdest, Rsrc1, Src2` *OR*
`ori Rdest, Rsrc1, Imm` *OR Immediate*
Put the logical OR of the integers from register `Rsrc1` and `Src2` (or `Imm`) into register `Rdest`.

`rem Rdest, Rsrc1, Src2` *Remainder* †
`remu Rdest, Rsrc1, Src2` *Unsigned Remainder* †
Put the remainder from dividing the integer in register `Rsrc1` by the integer in `Src2` into register `Rdest`. Note that if an operand is negative, the remainder is unspecified by the MIPS architecture and depends on the conventions of the machine on which SPIM is run.

`rol Rdest, Rsrc1, Src2` *Rotate Left* †
`ror Rdest, Rsrc1, Src2` *Rotate Right* †
Rotate the contents of register `Rsrc1` left (right) by the distance indicated by `Src2` and put the result in register `Rdest`.

`sll Rdest, Rsrc1, Src2` *Shift Left Logical*
`sllv Rdest, Rsrc1, Rsrc2` *Shift Left Logical Variable*
`sra Rdest, Rsrc1, Src2` *Shift Right Arithmetic*
`srav Rdest, Rsrc1, Rsrc2` *Shift Right Arithmetic Variable*
`srl Rdest, Rsrc1, Src2` *Shift Right Logical*
`srlv Rdest, Rsrc1, Rsrc2` *Shift Right Logical Variable*
Shift the contents of register `Rsrc1` left (right) by the distance indicated by `Src2` (`Rsrc2`) and put the result in register `Rdest`.

`sub Rdest, Rsrc1, Src2` *Subtract (with overflow)*
`subu Rdest, Rsrc1, Src2` *Subtract (without overflow)*
Put the difference of the integers from register `Rsrc1` and `Src2` into register `Rdest`.

`xor Rdest, Rsrc1, Src2` *XOR*
`xori Rdest, Rsrc1, Imm` *XOR Immediate*
Put the logical XOR of the integers from register `Rsrc1` and `Src2` (or `Imm`) into register `Rdest`.

A.2.5 Constant-Manipulating Instructions

`li Rdest, imm` *Load Immediate* †
Move the immediate `imm` into register `Rdest`.

`lui Rdest, imm` *Load Upper Immediate*
Load the lower halfword of the immediate `imm` into the upper halfword of register `Rdest`. The lower bits of the register are set to 0.

A.2.6 Comparison Instructions

In all instructions below, `Src2` can either be a register or an immediate value (a 16 bit integer).

`seq Rdest, Rsrc1, Src2` *Set Equal* †
Set register `Rdest` to 1 if register `Rsrc1` equals `Src2` and to be 0 otherwise.

`sge Rdest, Rsrc1, Src2` *Set Greater Than Equal* †
`sgeu Rdest, Rsrc1, Src2` *Set Greater Than Equal Unsigned* †
Set register `Rdest` to 1 if register `Rsrc1` is greater than or equal to `Src2` and to 0 otherwise.

`sgt Rdest, Rsrc1, Src2` *Set Greater Than* †
`sgtu Rdest, Rsrc1, Src2` *Set Greater Than Unsigned* †
Set register `Rdest` to 1 if register `Rsrc1` is greater than `Src2` and to 0 otherwise.

`sle Rdest, Rsrc1, Src2` *Set Less Than Equal* †
`sleu Rdest, Rsrc1, Src2` *Set Less Than Equal Unsigned* †
Set register `Rdest` to 1 if register `Rsrc1` is less than or equal to `Src2` and to 0 otherwise.

`slt Rdest, Rsrc1, Src2` *Set Less Than*
`slti Rdest, Rsrc1, Imm` *Set Less Than Immediate*
`sltu Rdest, Rsrc1, Src2` *Set Less Than Unsigned*
`sltiu Rdest, Rsrc1, Imm` *Set Less Than Unsigned Immediate*
Set register `Rdest` to 1 if register `Rsrc1` is less than `Src2` (or `Imm`) and to 0 otherwise.

`sne Rdest, Rsrc1, Src2` *Set Not Equal* †
Set register `Rdest` to 1 if register `Rsrc1` is not equal to `Src2` and to 0 otherwise.

A.2.7 Branch and Jump Instructions

In all instructions below, `Src2` can either be a register or an immediate value (integer). Branch instructions use a signed 16-bit offset field; hence they can jump $2^{15} - 1$ *instructions* (not bytes) forward or 2^{15} instructions backwards. The *jump* instruction contains a 26 bit address field.

`b label` *Branch instruction* †
Unconditionally branch to the instruction at the label.

`bczt label` *Branch Coprocessor z True*
`bczf label` *Branch Coprocessor z False*
Conditionally branch to the instruction at the label if coprocessor *z*'s condition flag is true (false).

`beq Rsrc1, Src2, label` *Branch on Equal*
Conditionally branch to the instruction at the label if the contents of register `Rsrc1` equals `Src2`.

`beqz Rsrc, label` *Branch on Equal Zero* †
Conditionally branch to the instruction at the label if the contents of `Rsrc` equals 0.

`bge Rsrc1, Src2, label` *Branch on Greater Than Equal* †
`bgeu Rsrc1, Src2, label` *Branch on GTE Unsigned* †
Conditionally branch to the instruction at the label if the contents of register `Rsrc1` are greater than or equal to `Src2`.

`bgez Rsrc, label` *Branch on Greater Than Equal Zero*
Conditionally branch to the instruction at the label if the contents of `Rsrc` are greater than or equal to 0.

`bgezal Rsrc, label` *Branch on Greater Than Equal Zero And Link*
Conditionally branch to the instruction at the label if the contents of `Rsrc` are greater than or equal to 0. Save the address of the next instruction in register 31.

`bgt Rsrc1, Src2, label` *Branch on Greater Than* †
`bgtu Rsrc1, Src2, label` *Branch on Greater Than Unsigned* †
Conditionally branch to the instruction at the label if the contents of register `Rsrc1` are greater than `Src2`.

`bgtz Rsrc, label` *Branch on Greater Than Zero*
Conditionally branch to the instruction at the label if the contents of `Rsrc` are greater than 0.

`ble Rsrc1, Src2, label` *Branch on Less Than Equal* †
`bleu Rsrc1, Src2, label` *Branch on LTE Unsigned* †
Conditionally branch to the instruction at the label if the contents of register `Rsrc1` are less than or equal to `Src2`.

`blez Rsrc, label` *Branch on Less Than Equal Zero*
Conditionally branch to the instruction at the label if the contents of `Rsrc` are less than or equal to 0.

`bgezal Rsrc, label` *Branch on Greater Than Equal Zero And Link*
`bltzal Rsrc, label` *Branch on Less Than And Link*
Conditionally branch to the instruction at the label if the contents of `Rsrc` are greater or equal to 0 or less than 0, respectively. Save the address of the next instruction in register 31.

`blt Rsrc1, Src2, label` *Branch on Less Than* †
`bltu Rsrc1, Src2, label` *Branch on Less Than Unsigned* †
Conditionally branch to the instruction at the label if the contents of register `Rsrc1` are less than `Src2`.

`bltz Rsrc, label` *Branch on Less Than Zero*
Conditionally branch to the instruction at the label if the contents of `Rsrc` are less than 0.

`bne Rsrc1, Src2, label` *Branch on Not Equal*
Conditionally branch to the instruction at the label if the contents of register `Rsrc1` are not equal to `Src2`.

`bnez Rsrc, label` *Branch on Not Equal Zero* †
Conditionally branch to the instruction at the label if the contents of `Rsrc` are not equal to 0.

`j label` *Jump*
Unconditionally jump to the instruction at the label.

`jal label` *Jump and Link*
`jalr Rsrc` *Jump and Link Register*
Unconditionally jump to the instruction at the label or whose address is in register `Rsrc`. Save the address of the next instruction in register 31.

`jr Rsrc` *Jump Register*
Unconditionally jump to the instruction whose address is in register `Rsrc`.

A.2.8 Load Instructions

`la Rdest, address` *Load Address* †
Load computed *address*, not the contents of the location, into register `Rdest`.

`lb Rdest, address` *Load Byte*
`lbu Rdest, address` *Load Unsigned Byte*
Load the byte at *address* into register `Rdest`. The byte is sign-extended by the `lb`, but not the `lbu`, instruction.

`ld Rdest, address` *Load Double-Word* †
Load the 64-bit quantity at *address* into registers `Rdest` and `Rdest + 1`.

`lh Rdest, address` *Load Halfword*
`lhu Rdest, address` *Load Unsigned Halfword*
Load the 16-bit quantity (halfword) at *address* into register `Rdest`. The halfword is sign-extended by the `lh`, but not the **lhu**, instruction

`lw Rdest, address` *Load Word*
Load the 32-bit quantity (word) at *address* into register `Rdest`.

`lwcz Rdest, address` *Load Word Coprocessor*
Load the word at *address* into register `Rdest` of coprocessor *z* (0–3).

`lwl Rdest, address` *Load Word Left*
`lwr Rdest, address` *Load Word Right*
Load the left (right) bytes from the word at the possibly-unaligned *address* into register `Rdest`.

`ulh Rdest, address` *Unaligned Load Halfword* †
`ulhu Rdest, address` *Unaligned Load Halfword Unsigned* †
Load the 16-bit quantity (halfword) at the possibly-unaligned *address* into register `Rdest`. The halfword is sign-extended by the `ulh`, but not the **ulhu**, instruction

`ulw Rdest, address` *Unaligned Load Word* †
Load the 32-bit quantity (word) at the possibly-unaligned *address* into register `Rdest`.

A.2.9 Store Instructions

`sb Rsrc, address` *Store Byte*
Store the low byte from register `Rsrc` at *address*.

`sd Rsrc, address` *Store Double-Word* †
Store the 64-bit quantity in registers `Rsrc` and `Rsrc + 1` at *address*.

`sh Rsrc, address` *Store Halfword*
Store the low halfword from register `Rsrc` at *address*.

`sw Rsrc, address` *Store Word*
Store the word from register `Rsrc` at *address*.

`swc`*z* `Rsrc, address` *Store Word Coprocessor*
Store the word from register `Rsrc` of coprocessor *z* at *address*.

`swl Rsrc, address` *Store Word Left*
`swr Rsrc, address` *Store Word Right*
Store the left (right) bytes from register `Rsrc` at the possibly-unaligned *address*.

`ush Rsrc, address` *Unaligned Store Halfword* †
Store the low halfword from register `Rsrc` at the possibly-unaligned *address*.

`usw Rsrc, address` *Unaligned Store Word* †
Store the word from register `Rsrc` at the possibly-unaligned *address*.

A.2.10 Data Movement Instructions

`move Rdest, Rsrc` *Move* †
Move the contents of `Rsrc` to `Rdest`.

The multiply and divide unit produces its result in two additional registers, hi and lo. These instructions move values to and from these registers. The multiply, divide, and remainder instructions described above are pseudoinstructions that make it appear as if this unit operates on the general registers and detect error conditions such as divide by zero or overflow.

`mfhi Rdest` *Move From hi*
`mflo Rdest` *Move From lo*
Move the contents of the hi (lo) register to register `Rdest`.

`mthi Rdest` *Move To hi*
`mtlo Rdest` *Move To lo*
Move the contents register `Rdest` to the hi (lo) register.

Coprocessors have their own register sets. These instructions move values between these registers and the CPU's registers.

`mfcz Rdest, CPsrc` *Move From Coprocessor z*
Move the contents of coprocessor *z*'s register `CPsrc` to CPU register `Rdest`.

`mfc1.d Rdest, FRsrc1` *Move Double From Coprocessor 1* †
Move the contents of floating point registers `FRsrc1` and `FRsrc1 + 1` to CPU registers `Rdest` and `Rdest + 1`.

`mtcz Rsrc, CPdest` *Move To Coprocessor z*
Move the contents of CPU register `Rsrc` to coprocessor *z*'s register `CPdest`.

A.2.11 Floating Point Instructions

The MIPS has a floating point coprocessor (numbered 1) that operates on single precision (32-bit) and double precision (64-bit) floating point numbers. This coprocessor has its own registers, which are numbered `$f0`–`$f31`. Because these registers are only 32-bits wide, two of them are required to hold doubles. To simplify matters, floating point operations only use even-numbered registers—including instructions that operate on single floats.

Values are moved in or out of these registers a word (32-bits) at a time by `lwc1`, `swc1`, `mtc1`, and `mfc1` instructions described above or by the `l.s`, `l.d`, `s.s`, and `s.d` pseudoinstructions described below. The flag set by floating point comparison operations is read by the CPU with its `bc1t` and `bc1f` instructions.

In all instructions below, `FRdest`, `FRsrc1`, `FRsrc2`, and `FRsrc` are floating point registers (e.g., `$f2`).

`abs.d FRdest, FRsrc` *Floating Point Absolute Value Double*
`abs.s FRdest, FRsrc` *Floating Point Absolute Value Single*
Compute the absolute value of the floating float double (single) in register `FRsrc` and put it in register `FRdest`.

`add.d FRdest, FRsrc1, FRsrc2` *Floating Point Addition Double*
`add.s FRdest, FRsrc1, FRsrc2` *Floating Point Addition Single*
Compute the sum of the floating float doubles (singles) in registers `FRsrc1` and `FRsrc2` and put it in register `FRdest`.

`c.eq.d FRsrc1, FRsrc2` *Compare Equal Double*
`c.eq.s FRsrc1, FRsrc2` *Compare Equal Single*
Compare the floating point double in register `FRsrc1` against the one in `FRsrc2` and set the floating point condition flag true if they are equal.

`c.le.d FRsrc1, FRsrc2` *Compare Less Than Equal Double*
`c.le.s FRsrc1, FRsrc2` *Compare Less Than Equal Single*
Compare the floating point double in register `FRsrc1` against the one in `FRsrc2` and set the floating point condition flag true if the first is less than or equal to the second.

`c.lt.d FRsrc1, FRsrc2` *Compare Less Than Double*
`c.lt.s FRsrc1, FRsrc2` *Compare Less Than Single*
Compare the floating point double in register `FRsrc1` against the one in `FRsrc2` and set the condition flag true if the first is less than the second.

`cvt.d.s FRdest, FRsrc` *Convert Single to Double*
`cvt.d.w FRdest, FRsrc` *Convert Integer to Double*
Convert the single precision floating point number or integer in register `FRsrc` to a double precision number and put it in register `FRdest`.

`cvt.s.d FRdest, FRsrc` *Convert Double to Single*
`cvt.s.w FRdest, FRsrc` *Convert Integer to Single*

Convert the double precision floating point number or integer in register `FRsrc` to a single precision number and put it in register `FRdest`.

`cvt.w.d FRdest, FRsrc` *Convert Double to Integer*
`cvt.w.s FRdest, FRsrc` *Convert Single to Integer*
Convert the double or single precision floating point number in register `FRsrc` to an integer and put it in register `FRdest`.

`div.d FRdest, FRsrc1, FRsrc2` *Floating Point Divide Double*
`div.s FRdest, FRsrc1, FRsrc2` *Floating Point Divide Single*
Compute the quotient of the floating float doubles (singles) in registers `FRsrc1` and `FRsrc2` and put it in register `FRdest`.

`l.d FRdest, address` *Load Floating Point Double* †
`l.s FRdest, address` *Load Floating Point Single* †
Load the floating float double (single) at `address` into register `FRdest`.

`mov.d FRdest, FRsrc` *Move Floating Point Double*
`mov.s FRdest, FRsrc` *Move Floating Point Single*
Move the floating float double (single) from register `FRsrc` to register `FRdest`.

`mul.d FRdest, FRsrc1, FRsrc2` *Floating Point Multiply Double*
`mul.s FRdest, FRsrc1, FRsrc2` *Floating Point Multiply Single*
Compute the product of the floating float doubles (singles) in registers `FRsrc1` and `FRsrc2` and put it in register `FRdest`.

`neg.d FRdest, FRsrc` *Negate Double*
`neg.s FRdest, FRsrc` *Negate Single*
Negate the floating point double (single) in register `FRsrc` and put it in register `FRdest`.

`s.d FRdest, address` *Store Floating Point Double* †
`s.s FRdest, address` *Store Floating Point Single* †
Store the floating float double (single) in register `FRdest` at `address`.

`sub.d FRdest, FRsrc1, FRsrc2` *Floating Point Subtract Double*
`sub.s FRdest, FRsrc1, FRsrc2` *Floating Point Subtract Single*
Compute the difference of the floating float doubles (singles) in registers `FRsrc1` and `FRsrc2` and put it in register `FRdest`.

A.2.12 Exception and Trap Instructions

`rfe` *Return From Exception*
Restore the Status register.

`syscall` *System Call*
Register `$v0` contains the number of the system call (see Table A.1) provided by SPIM.

`break n` *Break*
Cause exception *n*. Exception 1 is reserved for the debugger.

`nop` *No operation*
Do nothing.

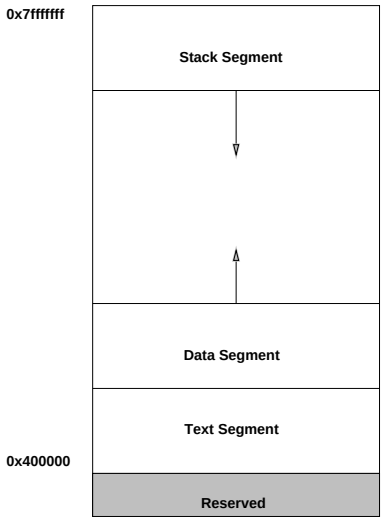


Abbildung A.5: Layout of memory.

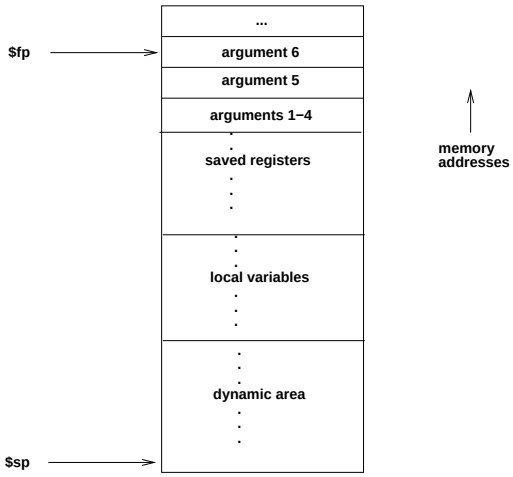


Abbildung A.6: Layout of a stack frame. The frame pointer points just below the last argument passed on the stack. The stack pointer points to the last word in the frame.

A.3 Memory Usage

The organization of memory in MIPS systems is conventional. A program's address space is composed of three parts (see Figure A.5).

At the bottom of the user address space (0x400000) is the text segment, which holds the instructions for a program.

Above the text segment is the data segment (starting at 0x10000000), which is divided into two parts. The static data portion contains objects whose size and address are known to the compiler and linker. Immediately above these objects is dynamic data. As a program allocates space dynamically (i.e., by `malloc`), the `sbrk` system call moves the top of the data segment up.

The program stack resides at the top of the address space (0x7fffffff). It grows down, towards the data segment.

A.4 Calling Convention

The calling convention described in this section is the one used by *gcc*, not the native MIPS compiler, which uses a more complex convention that is slightly faster.

Figure A.6 shows a diagram of a stack frame. A frame consists of the memory between the frame pointer (`$fp`), which points to the word immediately after the last argument passed on the stack, and the stack pointer (`$sp`), which points to the last word in the frame. As typical of Unix systems, the stack grows down from higher memory addresses, so the frame pointer is above stack pointer.

The following steps are necessary to effect a call:

1. Pass the arguments. By convention, the first four arguments are passed in registers `$a0`–`$a3` (though simpler compilers may choose to ignore this convention and pass all arguments via the stack). The remaining arguments are pushed on the stack.
2. Save the caller-saved registers. This includes registers `$t0`–`$t9`, if they contain live values at the call site.
3. Execute a `jal` instruction.

Within the called routine, the following steps are necessary:

1. Establish the stack frame by subtracting the frame size from the stack pointer.
2. Save the callee-saved registers in the frame. Register `$fp` is always saved. Register `$ra` needs to be saved if the routine itself makes calls. Any of the registers `$s0`–`$s7` that are used by the callee need to be saved.
3. Establish the frame pointer by adding the stack frame size - 4 to the address in `$sp`.

Finally, to return from a call, a function places the returned value into `$v0` and executes the following steps:

1. Restore any callee-saved registers that were saved upon entry (including the frame pointer `$fp`).
2. Pop the stack frame by adding the frame size to `$sp`.
3. Return by jumping to the address in register `$ra`.

A.5 Input and Output

In addition to simulating the basic operation of the CPU and operating system, SPIM also simulates a memory-mapped terminal connected to the machine. When a program is “running,” SPIM connects its own terminal (or a separate console window in `xspim`) to the processor. The program can read characters that you type while the processor is running. Similarly, if SPIM executes instructions to write characters to the terminal, the characters will appear on SPIM’s terminal or console window. One exception to this rule is control-C: it is not passed to the processor, but instead causes SPIM to stop simulating and return to command mode. When the processor stops executing (for example, because you typed control-C or because the machine hit a breakpoint), the terminal is reconnected to SPIM so you can type SPIM commands. To use memory-mapped IO, `spim` or `xspim` must be started with the `-mapped_io` flag.

The terminal device consists of two independent units: a *receiver* and a *transmitter*. The receiver unit reads characters from the keyboard as they are typed. The transmitter unit writes characters to the terminal’s display. The two units are completely independent. This means, for example, that characters typed at the keyboard are not automatically “echoed” on the display. Instead, the processor must get an input character from the receiver and re-transmit it to echo it.

The processor accesses the terminal using four memory-mapped device registers, as shown in Figure A.7. “Memory-mapped” means that each register appears as a special memory location. The Receiver Control Register is at location 0xffff0000; only two of its bits are actually used. Bit 0 is called “ready”: if it is one it means that a character has arrived from the keyboard but has not yet been read from the receiver data register. The ready bit is read-only: attempts to write it are ignored. The ready bit changes automatically from zero to one when a character is typed at the keyboard, and it changes automatically from one to zero when the character is read from the receiver data register.

Bit one of the Receiver Control Register is “interrupt enable”. This bit may be both read and written by the processor. The interrupt enable is initially zero. If it is set to one by the processor, an interrupt is requested by the terminal on

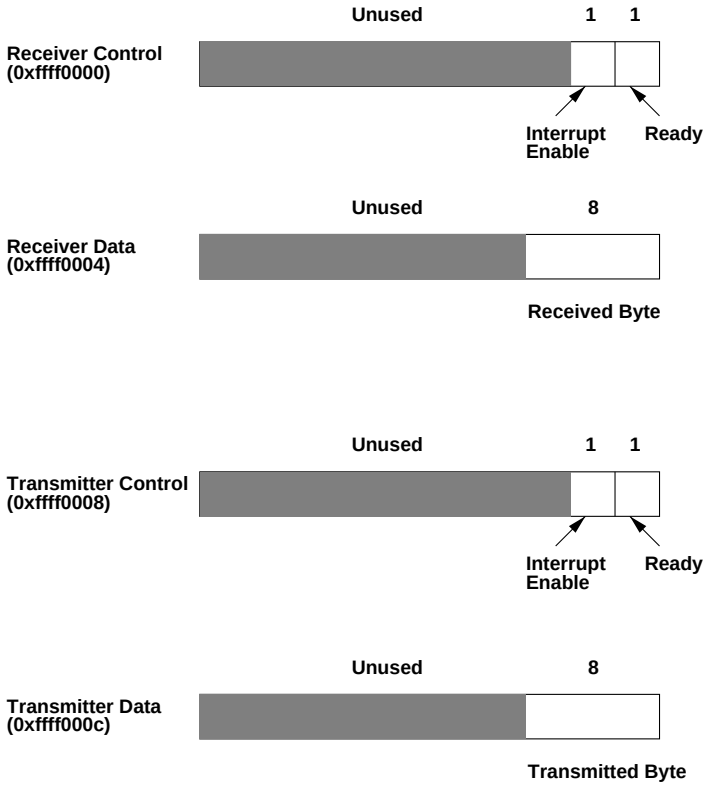


Abbildung A.7: The terminal is controlled by four device registers, each of which appears as a special memory location at the given address. Only a few bits of the registers are actually used: the others always read as zeroes and are ignored on writes.

level zero whenever the ready bit is one. For the interrupt actually to be received by the processor, interrupts must be enabled in the status register of the system coprocessor (see Section A.2).

Other bits of the Receiver Control Register are unused: they always read as zeroes and are ignored in writes.

The second terminal device register is the Receiver Data Register (at address 0xffff0004). The low-order eight bits of this register contain the last character typed on the keyboard, and all the other bits contain zeroes. This register is read-only and only changes value when a new character is typed on the keyboard. Reading the Receiver Data Register causes the ready bit in the Receiver Control Register to be reset to zero.

The third terminal device register is the Transmitter Control Register (at address 0xffff0008). Only the low-order two bits of this register are used, and they behave much like the corresponding bits of the Receiver Control Register. Bit 0 is called “ready” and is read-only. If it is one it means the transmitter is ready to accept a new character for output. If it is zero it means the transmitter is still busy outputting the previous character given to it. Bit one is “interrupt enable”; it is readable and writable. If it is set to one, then an interrupt will be requested on level one whenever the ready bit is one.

The final device register is the Transmitter Data Register (at address 0xffff000c). When it is written, the low-order eight bits are taken as an ASCII character to output to the display. When the Transmitter Data Register is written, the ready bit in the Transmitter Control Register will be reset to zero. The bit will stay zero until enough time has elapsed to transmit the character to the terminal; then the ready bit will be set back to one again. The Transmitter Data Register should only be written when the ready bit of the Transmitter Control Register is one; if the transmitter isn’t ready then writes to the Transmitter Data Register are ignored (the write appears to succeed but the character will not be output).

In real computers it takes time to send characters over the serial lines that connect terminals to computers. These time lags are simulated by SPIM. For example, after the transmitter starts transmitting a character, the transmitter’s ready bit will become zero for a while. SPIM measures this time in instructions executed, not in real clock time. This means that the transmitter will not become ready again until the processor has executed a certain number of instructions. If you stop the machine and look at the ready bit using SPIM, it will not change. However, if you let the machine run then the bit will eventually change back to one.

Anhang B

SPIM-Traphandler

```
# SPIM S20 MIPS simulator.
# The default trap handler for spim.
#
# Copyright (C) 1990-2000 James Larus, larus@cs.wisc.edu.
# ALL RIGHTS RESERVED.
#
# SPIM is distributed under the following conditions:
#
# You may make copies of SPIM for your own use and modify those copies.
#
# All copies of SPIM must retain my name and copyright notice.
#
# You may not sell SPIM or distributed SPIM in conjunction with a commerical
# product or service without the expressed written consent of James Larus.
#
# THIS SOFTWARE IS PROVIDED ‘‘AS IS’’ AND WITHOUT ANY EXPRESS OR
# IMPLIED WARRANTIES, INCLUDING, WITHOUT LIMITATION, THE IMPLIED
# WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
# PURPOSE.
#

# $Header: $


# Define the exception handling code.  This must go first!

.kdata
__m1_: .asciiz "  Exception "
__m2_: .asciiz " occurred and ignored\n"
__e0_: .asciiz "  [Interrupt] "
__e1_: .asciiz ""
__e2_: .asciiz ""
__e3_: .asciiz ""
__e4_: .asciiz "  [Unaligned address in inst/data fetch] "
__e5_: .asciiz "  [Unaligned address in store] "
__e6_: .asciiz "  [Bad address in text read] "
__e7_: .asciiz "  [Bad address in data/stack read] "
__e8_: .asciiz "  [Error in syscall] "
__e9_: .asciiz "  [Breakpoint] "
__e10_: .asciiz "  [Reserved instruction] "
__e11_: .asciiz ""
__e12_: .asciiz "  [Arithmetic overflow] "
__e13_: .asciiz "  [Inexact floating point result] "
__e14_: .asciiz "  [Invalid floating point result] "
__e15_: .asciiz "  [Divide by 0] "
__e16_: .asciiz "  [Floating point overflow] "
```

```
__e17_: .asciiz "  [Floating point underflow] "
__excp: .word __e0_,__e1_,__e2_,__e3_,__e4_,__e5_,__e6_,__e7_,__e8_,__e9_
.word __e10_,__e11_,__e12_,__e13_,__e14_,__e15_,__e16_,__e17_
s1: .word 0
s2: .word 0

.ktext 0x80000080
.set noat
# Because we are running in the kernel, we can use $k0/$k1 without
# saving their old values.
move $k1 $at # Save $at
.set at
sw $v0 s1 # Not re-entrent and we can't trust $sp
sw $a0 s2
mfc0 $k0 $13 # Cause
        sgt $v0 $k0 0x44 # ignore interrupt exceptions
        bgtz $v0 ret
        addu $0 $0 0
li $v0 4 # syscall 4 (print_str)
la $a0 __m1_
syscall
li $v0 1 # syscall 1 (print_int)
        srl $a0 $k0 2 # shift Cause reg
syscall
li $v0 4 # syscall 4 (print_str)
lw $a0 __excp($k0)
syscall
bne $k0 0x18 ok_pc # Bad PC requires special checks
mfc0 $a0, $14 # EPC
and $a0, $a0, 0x3 # Is EPC word-aligned?
beq $a0, 0, ok_pc
li $v0 10 # Exit on really bad PC (out of text)
syscall

ok_pc:
li $v0 4 # syscall 4 (print_str)
la $a0 __m2_
syscall
mtc0 $0, $13 # Clear Cause register
ret: lw $v0 s1
lw $a0 s2
mfc0 $k0 $14 # EPC
.set noat
move $at $k1 # Restore $at
.set at
rfe # Return from exception handler
addiu $k0 $k0 4 # Return to next instruction
jr $k0


# Standard startup code.  Invoke the routine main with no arguments.

.text
.globl __start
__start:
lw $a0, 0($sp) # argc
addiu $a1, $sp, 4 # argv
addiu $a2, $a1, 4 # envp
sll $v0, $a0, 2
addu $a2, $a2, $v0
jal main
li $v0 10
syscall # syscall 10 (exit)
```

Literaturverzeichnis

[Bae80] BAER, J.L.: *Computer Systems Architecture*. Pitmen, 1980

[Beh99] BEHR, B.: Am laufenden Band. In: *c't, Nov. 1999* (1999)

[BGN46] BURKS, A.W. ; GOLDSTINE, H.H. ; NEUMANN, J. von: Preliminary Discussion of the Logical Design of an Electronic Computing Element. In: *aus: A.H. Taub (ed.): Collected Works of John von Neumann, vol. 5, S. 34-79, The Macmillan Company, New York, 1963; Original: Report to the U.S. Army Ordnance Department* (1946)

[BH80] BODE, A. ; HÄNDLER, W.: *Rechnerarchitektur*. Springer, 1980

[BH83] BODE, A. ; HÄNDLER, W.: *Rechnerarchitektur II*. Springer, 1983

[Bäh02a] BÄHRING, H.: *Mikrorechner-Technik - 1. Mikroprozessoren und digitale Signalprozessoren*. Springer, 2002

[Bäh02b] BÄHRING, H.: *Mikrorechner-Technik - 2. Busse, Speicher, Peripherie und Mikrocontroller*. Springer, 2002

[Bha] BHATT, Ajay V.: Creating a Third Generation I/O Interconnect. In: *http://www.intel.com/technology/pciexpress/devnet/*

[Blu04] BLU-RAY DISC ASSOCIATION: Blue-ray Disc Format (White Paper). In: *http://www.blu-raydisc.com/assets/downloadablefile/general_bluraydiscformat-12834.pdf* (2004)

[BM08] BECKER, Bernd ; MOLITOR, Paul: *Technische Informatik: Eine einführende Darstellung*. Oldenbourg Wissenschaftsverlag, 2008

[BO03] BRYANT, R.E. ; O'HALLARON, D.: *Computer Systems - A Programmer's Perspective*. Pearson, 2003

[Böt06] BÖTTCHER, Axel: *Rechneraufbau und Rechnerarchitektur*. Springer, 2006

[Bur01] BURKHARDT, J.: *Pervasive Computing*. Addison-Wesley, 2001

[BW90] BODE, A. ; WILKE, P.: *RISC-Architekturen: Systeme, optimierende Compiler, Anwendungen, Leistungsmessung, Parallelismus*. Bibliographisches Institut, Mannheim, 1990

[CHPI+00] COMPAQ ; HEWLETT-PACKARD ; INTEL ; LUCENT ; MICROSOFT ; NEC ; PHILIPS: Universal Serial Bus Specification. In: *//www.usb.org* (2000)

[CLG+94] CHEN, P. M. ; LEE, E. K. ; GIBSON, G. A. ; KATZ, R. H. ; PATTERSON, D. A.: RAID: High-Performance, Reliable Secondary Storage. In: *acm computing surveys* 26 (1994), S. 145–186

[Coh81] COHEN: On holy wars and a plea for peace. In: *IEEE Computer* (1981)

[Coy92] COY, W.: *Aufbau und Arbeitsweise von Rechenanlagen*. Vieweg, 1992

[CW96] CAMPOSANO, R. ; WOLF, W.: Message from the editors-in-chief. In: *Design Automation for Embedded Systems* (1996)

[Fis81] FISHER, J.A.: Trace Scheduling: A Technique for Global Microcode Compaction. In: *IEEE Trans. on Computers, Vol.C–30* (1981), S. 478–790

[Fis93] FISCHER, W.: Datenkommunikation mittels ATM – Architekturen, Protokolle, Betriebsmittelverwaltung. In: **it + ti** *(Informationstechnik und Technische Informatik)* (1993), S. 3–11

[Ges06] GESELLSCHAFT FÜR INFORMATIK: Positionspapier "Was ist Infomatik?". In: *http://www.gi.de/fileadmin/redaktion/Download/was-ist-informatik-lang.pdf* (2006)

[Gil92] GILOI, W.: *Rechnerarchitektur*. Springer, 1992

[Han01] HANSMANN, U.: *Pervasive Computing*. Springer Verlag, 2001

[Hay98] HAYES, J.P.: *Computer Architecture and Organization*. McGraw-Hill, 1998

[Häb11] HÄBERLEIN, Tobias: *Technische Informatik: Ein Tutorium der Maschinenprogrammierung und Rechnertechnik*. Vieweg+Teubner, 2011

[Her02] HERRMANN, Paul: *Rechnerarchitektur*. Vieweg+Teubner, 2002

[Hof77] HOFFMANN, R.: *Rechenwerke und Mikroprogrammierung*. Oldenbourg, 1977

[HP95] HENNESSY, J. L. ; PATTERSON, D. A.: *Computer Architecture – A Quantitative Approach*. Morgan Kaufmann Publishers Inc., 1995

[HP96] HENNESSY, J. L. ; PATTERSON, D. A.: *Computer Organization – The Hardware/Software Interface*. Morgan Kaufmann Publishers Inc., 1996

[HP02] HENNESSY, J. L. ; PATTERSON, D. A.: *Computer Architecture – A Quantitative Approach*. Morgan Kaufmann Publishers Inc., 3. Aufl., 2002

[HP07] HENNESSY, J. L. ; PATTERSON, D. A.: *Computer Architecture – A Quantitative Approach, 4. Aufl.* Morgan Kaufmann Publishers Inc., 2007

[HP08] HENNESSY, J. L. ; PATTERSON, D. A.: *Computer Organization – The Hardware/Software Interface, 4. Aufl.* Morgan Kaufmann Publishers Inc., 2008

[HT93] HILLIS, W.D. ; TUCKER, L. W.: The CM-5 Connection Machine: A Scalable Supercomputer. In: *Communications of the ACM* (1993), S. 31–49

[ITR11] ITRS ORGANIZATION: International Technology Roadmap for Semiconductors (ITRS). In: *http: //public.itrs.net* (2011)

[Jes75] JESSEN, E.: *Architektur digitaler Rechenanlagen*. Springer, 1975

[JS92] JUNGMANN, D. ; STANGE, H.: *Einführung in die Rechnerarchitektur*. Hanser, 1992

[Kai89a] KAIN, R. Y.: *Computer Architecture Vol. I*. Prentice-Hall, 1989

[Kai89b] KAIN, R. Y.: *Computer Architecture Vol. II*. Prentice-Hall, 1989

[Kog91] KOGGE, Peter M.: *The architecture of symbolic computing*. McGraw-Hill, 1991

[Koz04] KOZIEROK, C. M.: Redundant Arrays of Inexpensive Disks (RAID). In: *http://www.pcguide.com/ref/hdd/perf/raid/levels/multLevel01-c.html* (2004)

[Kuc78] KUCK, D.J.: *The Structure of Computers and Computations, Vol. I*. Wiley, 1978

[LM03] LIEBIG, H. ; MENGE, M.: *Rechnerorganisation: Die Prinzipien*. Springer, 2003

[MA03] MARZANO, S. ; AARTS, E.: *The New Everyday*. 010 Publishers, 2003

[Mal78] MALLETT, P.W.: Methods of compacting microprograms (Dissertation). In: *University of Southwestern Louisiana, Lafayette* (1978)

[Mal04] MALZ, Helmut: *Rechnerarchitektur*. Vieweg+Teubner, 2004

[Mar07] MARWEDEL, P.: *Eingebettete Systeme*. Springer, 2007

[Mar10] MARWEDEL, P.: *Embedded System Design - Embedded Systems Foundations of CYber-Physical Systems*. Springer, 2010

[Mär03] MÄRTIN, Christian: *Einführung in die Rechnerarchitektur: Prozessoren und Systeme*. Hanser-Verlag, 2003

[MSS91] MÜLLER-SCHLOER ; SCHMITTER: *RISC-Workstation-Architekturen*. Springer, 1991

[OV06] OBERSCHELP, W. ; VOSSEN, G.: *Rechneraufbau und Rechnerstrukturen*. Oldenbourg Wissenschaftsverlag, 2006

[SBN82] SIEWIOREK, D. P. ; BELL, C. ; NEWELL, A.: *Computer Structures: principles and examples*. McGraw-Hill, 1982

[Sch73] SCHECHER, H.: *Funktioneller Aufbau digitaler Rechenanlagen*. Springer, 1973

[Sch93] SCHNURRER, G.: Moderne PC-Bussysteme. In: *c't, Okt.* (1993)

[Sho75] SHORE, John E.: On the external storage fragmentation produced by first-fit and best-fit allocation strategies. In: *Communications of the ACM* 18 (1975), S. 433 – 440

[Spa76] SPANIOL, O.: *Arithmetik in Rechenanlagen*. Teubner, 1976

[SS05a] SCHIFFMANN, W. ; SCHMITZ, R.: *Technische Informatik 1: Grundlagen der digitalen Elektronik*. Springer, 2005

[SS05b] SCHIFFMANN, W. ; SCHMITZ, R.: *Technische Informatik 2*. Springer, 2005

[Sta07] STALLINGS, W.: *Data and Computer Communications*. MacMillan, 2007

[Sta08] STALLINGS, W.: *Operating Systems*. Prentice Hall, 2008

[Sti00] STILLER, A.: Prozessorgeflüster. In: *c't* 22 (2000), S. 52

[Sto80] STONE, H.S.: *Introduction to Computer Architecture*. UMI Research Press, 1980

[Tan05] TANENBAUM, A.S.: *Structured Computer Organization*. Prentice Hall, 2005

[Tan09] TANENBAUM, A.: *Moderne Betriebssysteme*. Prentice Hall, 2009

[Tra96] TRANSTEC, Fa.: Info-Teil. In: *Gesamtkatalog Frühjahr* (1996)

[Ung97] UNGERER, T.: *Parallelrechner und parallele Programmierung*. Spektrum Akademischer Verlag, 1997

Index