

**DigiSem**  
Wir beschaffen und  
digitalisieren



---

b  
UNIVERSITÄT  
BERN

Universitätsbibliothek Bern

Dieses Dokument steht Ihnen online zur Verfügung  
dank DigiSem, einer Dienstleistung der  
Universitätsbibliothek Bern.

Kontakt: Gabriela Scherrer  
Koordinatorin digitale Semesterapparate  
E-Mail [digisem@ub.unibe.ch](mailto:digisem@ub.unibe.ch), Telefon 031 631 93 26

David A. Patterson

John L. Hennessy

# Rechnerorganisation und -entwurf

## Die Hardware/Software-Schnittstelle

3. Auflage herausgegeben von Arndt Bode,  
Wolfgang Karl und Theo Ungerer

Aus dem Amerikanischen übersetzt von Elke Jauch  
und Judith Muhr

A-4'52'313

Universitätsbibliothek Bern  
Zentralbibliothek

2007



Spektrum  
AKADEMISCHER VERLAG

T 1750 11

---

**Zuschriften und Kritik an:**

Elsevier GmbH, Spektrum Akademischer Verlag, Dr. Andreas Rüdinger, Slevogtstraße 3–5, 69126 Heidelberg

---

Titel der Originalausgabe: Computer Organization and Design, the hardware/software interface, 3rd edition

First published in the United States by Morgan Kaufmann Publishers, San Francisco, CA

Morgan Kaufmann Publishers is an Imprint of Elsevier. Copyright © 2005 Elsevier Inc. All rights reserved.

**Wichtiger Hinweis für den Benutzer**

Verlag, Autoren, Übersetzerinnen und Herausgeber haben alle Sorgfalt walten lassen, um vollständige und akkurate Informationen in diesem Buch und der beiliegenden CD-ROM zu publizieren. Der Verlag übernimmt weder Garantie noch die juristische Verantwortung oder irgendeine Haftung für die Nutzung dieser Informationen, für deren Wirtschaftlichkeit oder fehlerfreie Funktion für einen bestimmten Zweck. Ferner kann der Verlag für Schäden, die auf einer Fehlfunktion von Programmen oder ähnliches zurückzuführen sind, nicht haftbar gemacht werden. Auch nicht für die Verletzung von Patent- und anderen Rechten Dritter, die daraus resultieren. Eine telefonische oder schriftliche Beratung durch den Verlag über den Einsatz der Programme ist nicht möglich. Der Verlag übernimmt keine Gewähr dafür, dass die beschriebenen Verfahren, Programme usw. frei von Schutzrechten Dritter sind. Die Wiedergabe von Gebrauchsnamen, Handelsnamen, Warenbezeichnungen usw. in diesem Buch berechtigt auch ohne besondere Kennzeichnung nicht zu der Annahme, dass solche Namen im Sinne der Warenzeichen- und Markenschutz-Gesetzgebung als frei zu betrachten wären und daher von jedermann benutzt werden dürfen. Der Verlag hat sich bemüht, sämtliche Rechteinhaber von Abbildungen zu ermitteln. Sollte dem Verlag gegenüber dennoch der Nachweis der Rechtsinhaberschaft geführt werden, wird das branchenübliche Honorar gezahlt.

**Bibliografische Information Der Deutschen Bibliothek**

Die Deutsche Bibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über <http://dnb.ddb.de> abrufbar.

**Alle Rechte vorbehalten**

3. Auflage 2005

© Elsevier GmbH, München

Spektrum Akademischer Verlag ist ein Imprint der Elsevier GmbH.

05 06 07 08 09        5 4 3 2 1 0

Für Copyright in Bezug auf das verwendete Bildmaterial siehe Abbildungsnachweis.

Das Werk einschließlich aller seiner Teile ist urheberrechtlich geschützt. Jede Verwertung außerhalb der engen Grenzen des Urheberrechtsgesetzes ist ohne Zustimmung des Verlages unzulässig und strafbar. Das gilt insbesondere für Vervielfältigungen, Übersetzungen, Mikroverfilmungen und die Einspeicherung und Verarbeitung in elektronischen Systemen.

Planung und Lektorat: Dr. Andreas Rüdinger, Bianca Alton

Redaktion: Martin Radke

Herstellung: Ute Kreutzer

Umschlaggestaltung: SpieszDesign, Neu-Ulm

Titelfotografie: © zefa/stockbyte

Satz: Steingraeber Satztechnik, Dossenheim

Druck und Bindung: LegoPrint S.p.A.: Lavis

Printed in Italy

ISBN 3-8274-1595-0

## 2.1

# Einführung

### Befehlssatz (*instruction set*)

Der Wortschatz mit den Befehlen, die eine bestimmte Architektur versteht.

Um die Hardware eines Rechners zu steuern, müssen Sie die Sprache des Computers sprechen. Die Wörter der Sprache eines Rechners werden *Befehle* genannt und der Wortschatz wird als **Befehlssatz (*instruction set*)** bezeichnet. In diesem Kapitel werden Sie den Befehlssatz eines realen Computers kennen lernen, sowohl in der von Menschen geschriebenen Form als auch in der Form, wie er vom Computer gelesen wird. Die Befehle werden zunächst in Top-Down-Vorgehensweise eingeführt. Wir beginnen zunächst mit einer Notation, die an eine eingeschränkte Programmiersprache erinnert. Nach und nach wird die Darstellung schrittweise verfeinert, bis Sie die echte Sprache eines realen Computers vor sich haben. In Kapitel 3 werden wir in der Hierarchie weiter nach unten fortschreiten und die Darstellung von Integer- und Gleitkommazahlen und die Hardware näher betrachten, die mit diesen Zahlen arbeitet.

Möglicherweise stellen Sie sich die Sprachen der Computer so vielfältig wie die Sprachen der Menschen vor. Tatsächlich ist es jedoch so, dass die Sprachen der Rechner einander sehr ähnlich sind. In dieser Hinsicht lassen sie sich vielmehr mit Dialektien als mit eigenständigen Sprachen vergleichen. Wenn Sie also eine Sprache erlernt haben, ist es einfach, eine weitere zu erlernen. Diese Ähnlichkeiten ergeben sich aufgrund der Tatsache, dass alle Rechner mit Hardware-Techniken aufgebaut werden, denen ähnliche Prinzipien zugrunde liegen, und es wenige elementare Operationen gibt, die alle Rechner anbieten müssen. Darüber hinaus verfolgen Rechnerarchitekten ein gemeinsames Ziel: Eine Sprache zu finden, die das Konstruieren der Hardware und des Compilers erleichtert und dabei die Leistung maximiert und die Kosten minimiert. Dies ist ein althergebrachtes Ziel. Das folgende Zitat wurde niedergeschrieben, bevor es den ersten Computer zu kaufen gab, und es gilt heute noch genauso wie 1947:

*It is easy to see by formal-logical methods that there exist certain [instruction sets] that are in abstract adequate to control and cause the execution of any sequence of operations [...] The really decisive considerations from the present point of view, in selecting an [instruction set], are more of a practical nature: simplicity of the equipment demanded by the [instruction set], and the clarity of its application to the actually important problems together with the speed of its handling of those problems.*

Burks, Goldstine und von Neumann, 1947

Diese „simplicity of equipment“, womit die Einfachheit des Systemaufbaus bedingt durch den Befehlssatz gemeint ist, ist im Hinblick auf die Rechner im 21. Jahrhundert noch genauso wertvoll wie in den 50er-Jahren des 20. Jahrhunderts. Das Ziel dieses Kapitels ist es, einen Befehlssatz einzuführen, der diesem Rat folgt, wobei zum einen seine Repräsentation in der Hardware und zum anderen die Beziehung zwischen höheren Programmiersprachen und dieser eher primitiveren Sprache aufgezeigt wird. Die Beispiele sind in der Programmiersprache C geschrieben. In Abschnitt 2.14 auf CD ist dargestellt, wie sich diese mit einer objektorientierten Sprache wie Java ändern.



**Von-Neumann-Konzept**  
**(stored-program concept)** Die Idee, dass Befehle und Daten im Speicher als Zahlen gespeichert werden können. Sie führt zum Von-Neumann-Rechner.

Beim Erlernen der Befehle und ihrer Repräsentation werden Sie ebenso das Geheimnis der Rechnerorganisation entdecken: das **Von-Neumann-Konzept (stored-program concept)**. Darüber hinaus werden Sie Ihre „fremdsprachlichen“ Fähigkeiten üben und Programme in der Sprache des Rechners schreiben, die Sie dann mit dem auf der CD bereitgestellten Simulator ausführen können. Sie werden außerdem den Einfluss von Programmiersprachen und Compileroptimierungen auf die Leistung kennen

lernen. Das Kapitel schließt mit einem Blick auf die historische Entwicklung von Befehlssätzen und einer Übersicht über andere Sprachdialekte, die bei Rechnern zu finden sind.

Der ausgewählte Befehlssatz ist von MIPS abgeleitet, ein für die seit 1980 entworfenen Befehlssätze typisches Beispiel. Nahezu 100 Millionen dieser bekannten Mikroprozessoren wurden 2002 hergestellt und sind u. a. in Produkten von ATI Technologies, Broadcom, Cisco, NEC, Nintendo, Silicon Graphics, Sony, Texas Instruments und Toshiba zu finden.

Wir legen schrittweise den MIPS-Befehlssatz dar und zeigen dabei die Grundprinzipien der Rechnerorganisation auf. In diese schrittweise Untersuchung von oben nach unten werden die Komponenten mit den entsprechenden Erläuterungen so eingebunden, dass die Assemblersprache besser verständlich wird. Damit Sie den Überblick bewahren, finden Sie am Ende der einzelnen Abschnitte eine Abbildung, in der die bis dahin untersuchten Teile des MIPS-Befehlssatzes zusammenfassend dargestellt und die im jeweiligen Abschnitt beschriebenen Teile hervorgehoben sind.

## 2.2

# Operationen der Rechnerhardware

Jeder Rechner muss arithmetische Operationen ausführen können. In der Notation der MIPS-Assemblersprache wird mit

`add a, b, c`

ein Rechner angewiesen, die Variablen `b` und `c` zu addieren und das Ergebnis in `a` zu speichern.

Diese Notation legt genau fest, dass jeder arithmetische MIPS-Befehl nur eine Operation ausführt und immer genau drei Variablen enthalten muss. Nehmen wir beispielweise an, wir möchten die Summe der Variablen `b`, `c`, `d` und `e` in der Variablen `a` speichern. (In diesem Abschnitt nehmen wir es bewusst noch nicht ganz so genau damit, was eine „Variable“ ist. Das werden wir im nächsten Abschnitt genau erklären.)

Mit der folgenden Befehlsfolge werden die vier Variablen addiert:

```
add a, b, c    # a wird die Summe aus b und c zugewiesen
add a, a, d    # addiere d zu a hinzu
add a, a, e    # a wird schließlich die Summe aus b, c, d
               # und e zugewiesen
```

Es werden also drei Befehle benötigt, um vier Variablen zu addieren.

Der Text nach dem Rautensymbol (#) in den Zeilen oben, ist ein *Kommentar* für den menschlichen Leser und wird vom Rechner ignoriert. Im Gegensatz zu anderen Programmiersprachen kann bei dieser Sprache jede Zeile maximal einen Befehl enthalten. Ein weiterer Unterschied zu C besteht darin, dass Kommentare immer mit dem Zeilende abschließen.

Die natürliche Anzahl von Operanden für eine Operation wie die Addition ist drei: die beiden Zahlen, die addiert werden, und eine Zahl für den Ort, an dem das Ergebnis gespeichert wird. Die Tatsache, dass jeder Befehl aus genau drei Operanden, nicht mehr und nicht weniger, bestehen muss, entspricht der Philosophie, die Hardware einfacher zu halten: Die Hardware für eine variable Anzahl an Operanden ist komplexer als die Hardware für eine feste Anzahl an Operanden. Damit wird das erste der vier grundlegenden Prinzipien für den Hardwareentwurf deutlich:

*There must certainly be instructions for performing the fundamental arithmetic operations.*

Burks, Goldstine und von Neumann, 1947.

*Entwurfsprinzip 1:* Simplicity favors regularity (Einfachheit begünstigt Regelmäßigkeit).

Anhand der beiden folgenden Beispiele können wir Programme, die in einer höheren Programmiersprache geschrieben werden, mit Programmen vergleichen, die mit dieser elementarereren Notation geschrieben werden.

## BEISPIEL

### Übersetzen von zwei C-Anweisungen nach MIPS

Dieser Ausschnitt aus einem C-Programm enthält die fünf Variablen a, b, c, d und e. Da Java aus C hervorgegangen ist, stehen dieses und die nächsten Beispiele für beide höheren Programmiersprachen:

```
a = b + c;
d = a - e;
```

Die C-Befehle werden vom *Compiler* in Befehle in der MIPS-Assemblersprache übersetzt. Geben Sie den von einem Compiler generierten MIPS-Code an.

## ANTWORT

Ein MIPS-Befehl verarbeitet zwei Quelloperanden und legt das Ergebnis in einem Zieloperanden ab. Somit werden die beiden einfachen Anweisungen oben direkt in diese beiden Befehle in MIPS-Assemblersprache kompiliert:

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

## BEISPIEL

### Übersetzen einer komplexen C-Zuweisung nach MIPS

Eine etwas komplexere Anweisung enthält die fünf Variablen f, g, h, i und j:

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

Was wird ein C-Compiler möglicherweise generieren?

## ANTWORT

Der Compiler muss diese Anweisung in mehrere Assemblerbefehle aufteilen, da mit einem MIPS-Befehl nur eine Operation ausgeführt werden kann. Mit dem ersten MIPS-Befehl wird die Summe aus g und h berechnet. Das Ergebnis muss irgendwo abgelegt werden. Daher generiert der Compiler eine temporäre Variable t0:

```
add t0,g,h # der temporären Variable t0 wird g + h
# zugewiesen
```

Bevor die Subtraktion durchgeführt werden kann, muss zunächst die Summe aus i und j berechnet werden. Mit dem zweiten Befehl wird deshalb die Summe i und j in einer weiteren temporären Variablen abgelegt, die ebenfalls vom Compiler generiert und mit t1 bezeichnet wird:

```
add t1,i,j # der temporären Variable t1 wird i + j
# zugewiesen
```

Schließlich wird mit dem Subtraktionsbefehl die zweite Summe von der ersten subtrahiert. Die Differenz wird in der Variablen f gespeichert. Der kompilierte Code sieht somit wie folgt aus:

```
sub f,t0,t1 # f wird zu t0 - t1, also zu (g + h) - (i + j)
```

**Tab. 2.1 Die in Abschnitt 2.2 eingeführten Teile der MIPS-Architektur.** Die Behandlung der Operanden im Rechner wird im nächsten Abschnitt vorgestellt. Die in einem Abschnitt neu eingeführten MIPS-Assemblersprachkonstrukte werden in diesen Zusammenfassungen jeweils hervorgehoben. In dieser ersten Tabelle ist alles neu.

Kategorie	Befehl	Beispiel	Bedeutung	Kommentare
Arithmetischer Befehl	add	add a, b, c	a = b + c	Immer drei Operanden
	subtract	sub a, b, c	a = b - c	Immer drei Operanden

In Tabelle 2.1 sind die in diesem Abschnitt beschriebenen Teile der MIPS-Assemblersprache zusammengefasst. Bei diesen Befehlen handelt es sich um symbolische Darstellungen dessen, was der MIPS-Prozessor tatsächlich versteht. In den nächsten Abschnitten werden wir diese symbolische Darstellung in die reale Sprache des MIPS-Prozessors überführen und sie dabei mit jedem Schritt weiter konkretisieren.

Welche Programmiersprache benötigt für eine gegebene Funktion wahrscheinlich mehr Codezeilen? Geben Sie die nachfolgenden drei Sprachen in der entsprechenden Reihenfolge an.



1. Java
2. C
3. MIPS-Assemblersprache

**Vertiefung:** Ein wesentliches Merkmal von Java ist die Portierbarkeit, die durch die Verwendung eines Software-Interpreters erreicht wird. Der Befehlssatz dieses Interpreters wird als *Java-Bytecode* bezeichnet und unterscheidet sich erheblich vom MIPS-Befehlssatz. Um möglichst nahe an die Leistungsfähigkeit eines entsprechenden C-Programms heranzukommen, übersetzen heute Java-Systeme den Java-Bytecode direkt in die Zielsprache, in unserem Fall den MIPS-Befehlssatz. Da dieser Übersetzungs vorgang meist zu einem späteren Zeitpunkt als bei C-Programmen erfolgt, werden derartige Java-Compiler häufig als *JIT-Compiler (Just In Time)* bezeichnet. In Abschnitt 2.10 wird gezeigt, wie JIT-Compiler beim Startvorgang zu einem späteren Zeitpunkt als C-Compiler angestoßen werden, und in Abschnitt 2.13 wird beschrieben, wie sich bei Java-Programmen die Übersetzung im Vergleich zur Interpretation auf die Leistungsfähigkeit auswirkt. Bei den Java-Beispielen in diesem Kapitel wird der Schritt mit dem Java-Bytecode übersprungen und nur der von einem Compiler generierte MIPS-Code dargestellt.



## 2.3

## Operanden der Rechnerhardware

Im Unterschied zu Programmen in höheren Programmiersprachen gelten für die Operanden arithmetischer Befehle bestimmte Einschränkungen. Sie müssen an speziellen Stellen im Rechner bereitstehen, die jedoch nur in einer beschränkten Anzahl zur Verfügung stehen: den *Registers*. Register sind elementare Komponenten beim Hardwareentwurf und bilden die Grundbausteine für den Aufbau von Rechnern. Nach der

**Wort (word)** Die natürliche Zugriffseinheit in einem Rechner, meist eine 32 Bit umfassende Einheit; entspricht der Größe eines Registers in der MIPS-Architektur.



Fertigstellung des Rechners sind sie auch für den Programmierer sichtbar. Die Größe eines Registers bei der MIPS-Architektur beträgt 32 Bit. Die Zusammenfassung von 32 Bit zu einer Einheit geschieht sehr häufig, weshalb eine solche Einheit bei der MIPS-Architektur die Bezeichnung **Wort (word)** erhalten hat.

Ein wesentlicher Unterschied zwischen den Variablen einer Programmiersprache und Registern ist die begrenzte Anzahl an Registern, die üblicherweise bei aktuellen Rechnern 32 beträgt. Der MIPS-Prozessor verfügt über 32 Register. (Die Geschichte zur Anzahl der Register finden Sie in Abschnitt 2.19 auf CD.) Daher haben wir, der schrittweisen Entwicklung der symbolischen Darstellung der MIPS-Sprache folgend, in diesem Abschnitt die Einschränkung hinzuzufügen, dass für die drei Operanden eines arithmetischen MIPS-Befehls jeweils eines der 32 32-Bit-Register ausgewählt werden muss.

Der Grund für die Beschränkung auf 32 Register liegt im zweiten unserer vier grundlegenden Entwurfsprinzipien:

*Entwurfsprinzip 2: Smaller is faster (Kleiner ist schneller).*

Eine große Anzahl von Registern kann zu einer längeren Taktzykluszeit führen, da die elektronischen Signale für den weiteren Weg mehr Zeit benötigen.

Prinzipien wie „kleiner ist schneller“ gelten nicht absolut. 31 Register sind nicht zwangsläufig schneller als 32 Register. Den wahren Kern hinter solchen Beobachtungen muss der Rechnerarchitekt ernsthaft berücksichtigen. In diesem Fall muss er seinen Wunsch nach einem schnelleren Takt mit dem Verlangen von Programmen nach mehr Registern abwägen. Ein weiterer Grund dafür, dass nicht mehr als 32 Register verwendet werden, ist die hierfür erforderliche Anzahl von Bits im Befehlsformat, wie in Abschnitt 2.4 gezeigt wird.

In den Kapiteln 5 und 6 wird die zentrale Rolle der Register beim Hardwareentwurf gezeigt. In diesem Kapitel werden wir dagegen sehen, dass die effektive Nutzung von Registern von besonderer Bedeutung für die Leistungsfähigkeit von Programmen ist.

Obwohl wir in den Befehlen einfach die Registernummern schreiben könnten, ist die Konvention bei MIPS für die Bezeichnung von Registern das Dollarzeichen gefolgt von zwei Zeichen. In Abschnitt 2.7 werden die Gründe für diese Namensgebung erklärt. Im Moment verwenden wir \$s0, \$s1, ... für Register, die einer Variablen in C- und Java-Programmen entsprechen, und \$t0, \$t1, ... für temporäre Register, die zum Kompilieren des Programms in MIPS-Befehle benötigt werden.

## BEISPIEL

### Übersetzung einer Zuweisung in C mithilfe von Registern

Die Aufgabe des Compilers besteht darin, Programmvariablen Registern zuzuweisen. Nehmen wir beispielsweise die Zuweisung aus unserem obigen Beispiel:

$f = (g + h) - (i + j);$

Die Variablen f, g, h, i und j werden jeweils den Registern \$s0, \$s1, \$s2, \$s3 und \$s4 zugewiesen. Wie sieht der kompilierte MIPS-Code aus?

Das übersetzte Programm ist dem vorigen Beispiel sehr ähnlich. Es unterscheidet sich lediglich dadurch, dass die Variablen durch die oben erwähnten Registernamen und die temporären Variablen durch die beiden temporären Register \$t0 und \$t1 ersetzt werden:

```
add $t0,$s1,$s2 # Register $t0 wird g + h zugewiesen
add $t1,$s3,$s4 # Register $t1 wird i + j zugewiesen
sub $s0,$t0,$t1 # f wird $t0 - $t1 zugewiesen
                  # also (g + h) - (i + j)
```

## ANTWORT

## Speicheroperanden

Programmiersprachen verfügen über einfache Variablen, die wie in diesen Beispielen einzelne Datenelemente enthalten, sie verfügen jedoch auch über komplexere Datenstrukturen: Felder und Strukturen. Diese komplexen Datenstrukturen können wesentlich mehr Datenelemente enthalten, als es Register in einem Computer gibt. Wie kann ein Computer nun diese komplexen Strukturen darstellen und auf diese zugreifen?

Erinnern Sie sich an die fünf Komponenten eines Rechners, die in Kapitel 1 vorgestellt wurden und jeweils zu Beginn eines Kapitels dargestellt sind. Im Prozessor kann nur eine kleine Menge von Daten in den Registern gehalten werden. Im Hauptspeicher können dagegen Millionen von Datenelementen gespeichert werden. Daher werden Datenstrukturen (Felder und Strukturen) im Hauptspeicher abgelegt.

Wie weiter oben bereits erläutert, treten bei arithmetischen Operationen im MIPS-Befehlssatz nur Register auf, weshalb dieser auch Befehle zum Transport von Daten zwischen Hauptspeicher und Register enthalten muss. Diese Befehle werden als **Datentransfer-Befehle** (*data transfer instruction*) bezeichnet.

Für den Zugriff auf ein Wort im Hauptspeicher muss im Befehl die **Speicheradresse** (*address*) angegeben sein. Der Hauptspeicher ist ein großes, eindimensionales Feld, wobei die Adresse beginnend bei 0 als Index für das Feld dient. Beispiel: In Abbildung 2.1 lautet die Adresse des dritten Datenelements 2 und der Wert von Speicher[2] ist 10.

Der Datentransfer-Befehl, mit dem Daten vom Speicher in ein Register kopiert werden, wird als *Ladebefehl* bezeichnet. Das Format des Ladebefehls setzt sich aus dem Namen der Operation gefolgt von dem zu ladenden Register sowie einer Konstanten und einem Register für den Speicherzugriff zusammen. Die Summe der Konstanten des Befehls und der Inhalt des zweiten Registers bilden die Speicheradresse. Der eigentliche MIPS-Name für diesen Befehl lautet *lw*, was für *load word* (Wort laden) steht.

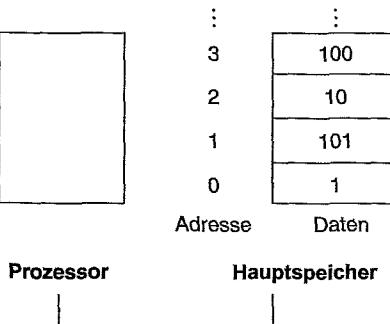


Abb. 2.1 Speicheradressen und Inhalt des Speichers an diesen Stellen. Hierbei handelt es sich um eine Vereinfachung der MIPS-Adressierung. In Abbildung 2.2 ist die tatsächliche MIPS-Adressierung für sequenzielle Wortadressen im Hauptspeicher dargestellt.

### Übersetzung einer Zuweisung mit einem Operanden im Speicher

Nehmen wir an, A sei ein Feld mit 100 Wörtern, und der Compiler weise wie zuvor die Variablen g und h den Registern \$s1 und \$s2 zu. Nehmen wir weiter an, die Startadresse oder *Basisadresse* des Felds befände sich in \$s3. Übersetzen Sie diese C-Zuweisung:

`g = h + A[8];`

### BEISPIEL

## ANTWORT

Diese Zuweisung enthält zwar nur eine Operation, aber einer der Operanden befindet sich im Hauptspeicher. Daher müssen wir zunächst A [8] in ein Register übertragen. Die Adresse dieses Feldelements ist die Summe der Basisadresse von Feld A, die im Register \$s3 steht, und dem Index für die Auswahl von Element 8. Damit die Daten im nächsten Befehl verwendet werden können, müssen sie in einem temporären Register gespeichert werden. Auf der Grundlage von Abbildung 2.1 lautet der erste übersetzte Befehl wie folgt:

```
lw $t0,8($s3) # temp. Reg. $t0 = A[8]
```

(Auf der nächsten Seite werden wir an diesem Befehl eine geringfügige Änderung vornehmen. Im Moment verwenden wir jedoch diese vereinfachte Version.) Der folgende Befehl kann auf dem Wert in \$t0, der gleich A [8] ist, eine Operation durchführen, da dieser sich in einem Register befindet. Der Befehl muss h (steht in \$s2) zu A [8] (steht in \$t0) addieren und das Ergebnis in das Register speichern, das der Variablen g zugeteilt ist (\$s1):

```
add $s1,$s2,$t0 # g = h + A[8]
```

Die Konstante in einem Datentransfer-Befehl wird als *konstante Abstandsgröße* oder *Offset* bezeichnet, und das Register, dessen Inhalt zur Adressbildung addiert wird, heißt Basisregister.

## Hardware-Software-Schnittstelle

**Ausrichtung an Wortgrenzen (alignment restriction)** Die Anforderung, dass Daten im Hauptspeicher an Wortgrenzen ausgerichtet sein müssen.



Der Compiler bindet nicht nur Variablen an Register, er ordnet darüber hinaus auch Datenstrukturen wie Felder und Strukturen Stellen im Hauptspeicher zu. So kann der Compiler dann die richtige Startadresse in die Datentransfer-Befehle einfügen.

Da in vielen Programmen sinnvollerweise *Byte* (8 Bit) verwendet werden, adressieren die meisten Architekturen Bytes. Daher entspricht die Adresse eines Wortes der Adresse eines der 4 Bytes im Wort. Die Adressen von aufeinander folgenden Wörtern unterscheiden sich somit um 4. In Abbildung 2.2 sind beispielsweise die tatsächlichen MIPS-Adressen für Abbildung 2.1 dargestellt. Die Byteadresse des dritten Wortes ist 8.

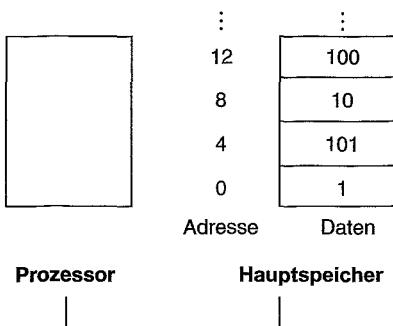
Beim MIPS-Befehlssatz müssen Wörter bei Adressen beginnen, die ein Vielfaches von 4 sind. Diese Forderung wird als **Ausrichtung an Wortgrenzen (alignment restriction)** bezeichnet, die von vielen Architekturen erfüllt wird. (In Kapitel 5 wird beschrieben, warum die Ausrichtung an Wortgrenzen eine schnellere Datenübertragung ermöglicht.)

Bezüglich der Adressierung eines Wortes im Speicher lassen sich Rechner in zwei Gruppen aufteilen. Eine Gruppe verwendet die Adresse des linken oder „big end“-Byte als Wortadresse, bei der anderen Gruppe gilt die Adresse des rechten oder „little end“-Byte als Wortadresse. Der MIPS-Befehlssatz gehört zur *Big-Endian*-Gruppe. (In Appendix A, Seite A-43 auf der CD sind die beiden Möglichkeiten zur Nummerierung der Byte in einem Wort dargestellt.)

Die Byteadressierung wirkt sich auch auf den Feldindex aus. Um die richtige Byteadresse im obigen Code zu erhalten, *muss der zum Basisregister \$s3 addierte Offset  $4 \times 8$  bzw. 32 betragen*, so dass die Ladeadresse nicht A [8/4], sondern A [8] auswählt. (Siehe hierzu den Fallstrick Seite 124 in Abschnitt 2.17.)

---

Das Befehlspendant zum Ladebefehl ist der *Speicherbefehl*, mit dem Daten aus einem Register in den Hauptspeicher kopiert werden. Der Speicherbefehl weist ein ähnliches Format wie der Ladebefehl auf: Auf den Namen der Operation folgt das zu



**Abb. 2.2 Tatsächliche MIPS-Speicheradressen und Speicherinhalte für diese Wörter.** Die geänderten Adressen sind zum Vergleich mit Abbildung 2.1 hervorgehoben. MIPS unterstützt die Byteadressierung. Weshalb Wortadressen Vielfache von 4 sind: Ein Wort besteht aus vier Byte.

speichernde Register, der Offset zur Auswahl des Feldelements und schließlich das Basisregister. Auch hier wird die MIPS-Adresse zum einen durch eine Konstante und zum anderen durch den Inhalt eines Registers spezifiziert. Der eigentliche MIPS-Name für diesen Befehl lautet `sw`, was für *store word* (Wort speichern) steht.

## Übersetzen mit Lade- und Speicherbefehlen

## **BEISPIEL**

Angenommen, die Variable h ist an das Register \$s2 gebunden und die Basisadresse von Feld A steht in \$s3. Wie lautet dann der MIPS-Assemblercode für die folgende Zuweisung in C?

A[12] = h + A[8];

Diese C-Anweisung enthält zwar nur eine Operation, aber nun befinden sich zwei Operanden im Hauptspeicher. Daher benötigen wir noch mehr MIPS-Befehle. Die ersten beiden Befehle sind dieselben wie im Beispiel weiter oben, außer dass hier nun der korrekte Offset für die Byteadressierung im `lw`-Befehl für den Zugriff auf `A[8]` verwendet wird, und mit dem `add`-Befehl wird das Ergebnis in `$t0` gespeichert:

## **ANTWORT**

```
lw $t0,32($s3)    # in temp. Reg. $t0 wird A[8] geladen  
add $t0,$s2,$t0   # temp. Reg. $t0 wird h + A[8]  
                  # zugewiesen
```

Mit dem letzten Befehl wird das Ergebnis in A[12] gespeichert, wobei 48 als Offset und Register \$s3 als Basisregister verwendet wird.

```
sw $t0,48($s3) # speichere h + A[8] in A[12]
```

## Konstante oder Direktoperanden

In Programmen werden in Operationen häufig Konstanten verwendet, z.B. beim Inkrementieren eines Index, damit dieser auf das nächste Element eines Felds zeigt. Mehr als die Hälfte der arithmetischen MIPS-Befehle verwenden beim Ausführen der SPEC2000-Benchmarks eine Konstante als Operand.

## Hardware-Software-Schnittstelle

Viele Programme enthalten mehr Variablen als es Register in einem Rechner gibt. Folglich versucht der Compiler, die am häufigsten verwendeten Variablen in Registern zu halten, und legt den Rest im Hauptspeicher ab, wobei er die Variablen mithilfe von Lade- und Speicherbefehlen zwischen den Registern und dem Hauptspeicher hin und her transportiert. Der Vorgang, weniger häufig verwendete Variablen (oder Variablen, die erst später benötigt werden) im Hauptspeicher abzulegen, wird als Registerauslagerung (*Spilling*) bezeichnet.

Das Hardware-Entwurfsprinzip der Beziehung zwischen Größe und Geschwindigkeit legt nahe, dass der Hauptspeicher langsamer sein muss als die Register, da die Register kleiner sind. Das trifft auch tatsächlich zu. Der Zugriff auf Daten in Registern ist schneller als der auf Daten im Hauptspeicher.

Zudem sind die Daten nützlicher, wenn sie sich in einem Register befinden. Ein arithmetischer MIPS-Befehl kann zwei Register lesen, die Operanden miteinander verknüpfen und das Ergebnis schreiben. Ein MIPS-Datentransfer-Befehl liest nur einen Operanden oder schreibt einen Operanden, ohne eine Operation darauf auszuführen.

Im Vergleich zum Hauptspeicher kann auf die Register schneller zugegriffen werden *und* mit ihnen erzielt man einen höheren Durchsatz. Eine seltene Kombination. Dadurch erfolgt der Zugriff auf Daten in Registern schneller und die Daten in Registern lassen sich einfacher verwenden. Um ein Höchstmaß an Leistungsfähigkeit zu erzielen, müssen Compiler Register effizient nutzen.

Wenn wir nur die bisher bekannten Befehle verwenden würden, müssten wir eine Konstante aus dem Hauptspeicher laden, um sie zu verwenden. (Die Konstanten müssten im Speicher abgelegt werden, wenn das Programm geladen wird.) Um beispielsweise die Konstante 4 zum Inhalt des Registers \$s3 zu addieren, könnten wir das Programm

```
lw $t0, AddrConstant4($s1) # $t0 = 4
add $s3,$s3,$t0 # $s3 = $s3 + $t0 ($t0 == 4)
```

hernehmen, wobei `AddrConstant4` die Speicheradresse der Konstante 4 ist.

Eine Alternative, die keinen Ladebefehl erfordert, besteht darin, Versionen der arithmetischen Befehle bereitzustellen, bei denen ein Operand eine Konstante ist. Dieser schnelle Add-Befehl mit einer Konstante als Operand wird als *add immediate* („addiere direkt“) oder `addi` bezeichnet. Um die Konstante 4 zum Inhalt des Registers \$s3 zu addieren, schreiben wir einfach

```
addi $s3,$s3,4 # $s3 = $s3 + 4
```

Immediate-Befehle veranschaulichen das dritte Prinzip für den Hardwareentwurf, das in Kapitel 4.5 näher untersucht wird:

*Entwurfsprinzip 3: Make the common case fast* (Optimiere den häufig vorkommenden Fall).

Konstanten werden häufig als Operanden verwendet. Und durch die Verwendung von Konstanten in arithmetischen Befehlen können diese viel schneller ausgeführt werden, als wenn Konstanten erst aus dem Hauptspeicher geladen werden müssten.

In Tabelle 2.2 sind die in diesem Abschnitt beschriebenen Teile der symbolischen Darstellung des MIPS-Befehlssatzes zusammengefasst. Mit den Befehlen „load word“ und „store word“ werden in der MIPS-Architektur Wörter zwischen Speicher und Register transportiert. Andere Architekturen verwenden Befehle zusammen mit Lade- und Speicheroperationen, um Daten zu übertragen. Eine Architektur mit solchen Alternativen ist die in Abschnitt 2.16 beschriebene Intel IA-32.

**Tab. 2.2 Die bis Abschnitt 2.3 bearbeitete MIPS-Architektur.** Die in Abschnitt 2.3 neu eingeführten Strukturen in MIPS-Assemblersprache sind farblich hervorgehoben.

### MIPS-Operanden

Name	Beispiel	Anmerkungen
32 Register	<code>\$s0, \$s1, ..., \$t0, \$t1, ...</code>	Speicherort für schnellen Zugriff auf Daten. Bei MIPS müssen die Daten für die arithmetischen Operationen in Registern stehen.
$2^{30}$ Speicherwörter	<code>Speicher[0], Speicher[4], ..., Speicher[4294967292]</code>	Zugriff bei MIPS nur durch Datentransport-Befehle. MIPS verwendet Byteadressen. Daher unterscheiden sich aufeinander folgende Wörter um 4. Im Hauptspeicher werden Datenstrukturen, Felder und ausgelagerte Register gespeichert.

### MIPS-Assemblersprache

Kategorie	Befehl	Beispiel	Bedeutung	Anmerkungen
Arithmetischer Befehl	add	<code>add \$s1, \$s2, \$s3</code>	$\$s1 = \$s2 + \$s3$	Drei Operanden; Daten in Registern
	subtract	<code>sub \$s1, \$s2, \$s3</code>	$\$s1 = \$s2 - \$s3$	Drei Operanden; Daten in Registern
	add immediate	<code>addi \$s1, \$s2, 100</code>	$\$s1 = \$s2 + 100$	Addieren von Konstanten
Daten transport	load word	<code>lw \$s1, 100(\$s2)</code>	$\$s1 = \text{Speicher}[\$s2 + 100]$	Daten vom Hauptspeicher in ein Register
	store word	<code>sw \$s1, 100(\$s2)</code>	$\text{Speicher}[\$s2 + 100] = \$s1$	Daten von einem Register in den Hauptspeicher

Wie schnell hat sich die Anzahl der Register im Laufe der Zeit erhöht, angesichts der Bedeutung der Register?



1. Sehr schnell: Die Anzahl der Register nimmt gemäß dem Gesetz von Moore zu, das eine Verdopplung der Anzahl der Transistoren auf einem Chip alle 18 Monate voraussagt.
2. Sehr langsam: Da Programme normalerweise in der Sprache des Computers verbreitet werden, ist für die Befehlssatzarchitektur eine gewisse Trägheit zu beobachten. Daher nimmt die Anzahl der Register lediglich mit der Verfügbarkeit neuer Befehlssätze zu.

**Vertiefung:** Die MIPS-Register in diesem Buch sind 32 Bit breit. Es gibt auch eine 64-Bit-Version des MIPS-Befehlssatzes mit 32 64-Bit-Registern. Um die beiden Versionen des MIPS-Befehlssatzes auseinander zu halten, werden sie offiziell als MIPS-32 und MIPS-64 bezeichnet. In diesem Kapitel verwenden wir eine Teilmenge von MIPS-32. In Appendix D auf der CD werden die Unterschiede zwischen MIPS-32 und MIPS-64 erläutert.



Die Adressierungsart bei MIPS mit Offset und Basisregister ermöglicht in exzellenter Weise, Strukturen und Felder nachzubilden. Ein Beispiel hierfür finden Sie in Abschnitt 2.13.

Ursprünglich wurde das Register in den Datentransfer-Befehlen zum Speichern eines Index für ein Feld eingeführt, wobei der Offset für die Anfangsadresse eines

Felds verwendet wird. Daher wird das Basisregister auch als *Indexregister* bezeichnet. Die Hauptspeicher von heute sind wesentlich größer und das Softwaremodell der Datenzuordnung ist komplexer. Daher wird die Basisadresse des Felds normalerweise in einem Register gespeichert, da sie, wie wir noch sehen werden, in das Feld des Offsets nicht mehr passt.

In Abschnitt 2.4 wird erläutert, dass der MIPS-Befehlssatz negative Konstanten unterstützt. Weshalb ein Subtract-immediate-Befehl nicht notwendig ist.

## 2.4

# Darstellung von Befehlen im Rechner

Nun ist es soweit, dass wir den Unterschied zwischen der Art und Weise, wie Menschen Rechnern Befehle erteilen, und der Art und Weise, wie Rechner die Befehle sehen, erklären können. Zuvor möchten wir jedoch kurz zusammenfassen, wie Zahlen im Rechner dargestellt werden.

Menschen denken in Zahlensystemen zur Basis 10. Zahlen können jedoch mit einer beliebigen anderen Basis dargestellt werden. Beispielsweise ist 123 zur Basis 10 gleich 1111011 zur Basis 2.

In der Rechnerhardware werden Zahlen als Folge elektrischer Signale mit hohem und niedrigen Potenzial behandelt und damit als Zahlen zur Basis 2 betrachtet. (So wie Zahlen zur Basis 10 als *Dezimalzahlen* bezeichnet werden, heißen Zahlen zur Basis 2 *Binärzahlen*.) Somit bildet eine Stelle einer Binärzahl das „Atom“ der Rechnertechnik, da alle Informationen aus **Binärziffern** (*binary digit*) oder **Bits** zusammengesetzt sind. Dieser elementare Baustein kann einen von zwei Werten annehmen, wobei mehrere Alternativen denkbar sind: hohes oder niedriges Potenzial, wahr oder falsch oder 1 oder 0.

Befehle werden im Rechner ebenfalls als Folge elektronischer Signale mit jeweils hohem und niedrigem Potenzial betrachtet und können somit auch als Zahlen interpretiert werden. So kann jeder Teil eines Befehls als eine Zahl betrachtet werden. Die einzelnen Zahlen aneinander gereiht ergeben den Befehl.

Da Register Teil nahezu aller Befehle sind, muss es eine Konvention geben, wie Registernamen Zahlen zugeordnet werden. In der MIPS-Assemblersprache werden die Register \$s0 bis \$s7 den Registern 16 bis 23 und die Register \$t0 bis \$t7 den Registern 8 bis 15 zugeordnet. Somit bedeutet \$s0 Register 16, \$s1 Register 17, \$s2 Register 18, ..., \$t0 Register 8, \$t1 Register 9 usw. Die Konvention für die restlichen der 32 Register wird in den folgenden Abschnitten beschrieben.

## BEISPIEL

### Übersetzung eines MIPS-Assemblerbefehls in einen Maschinenbefehl

Den nächsten Schritt bei der Erarbeitung der MIPS-Sprache zeigen wir anhand eines Beispiels. Wir zeigen den Befehl mit der symbolischen Darstellung

`add $t0,$s1,$s2`

in der tatsächlichen MIPS-Sprache zunächst als Kombination von Dezimalzahlen und anschließend als Folge von Binärzahlen.

Die Darstellung mit Dezimalzahlen sieht wie folgt aus:

0	17	18	8	0	32
---	----	----	---	---	----

## ANTWORT

Jedes dieser Segmente eines Befehls wird als *Feld* bezeichnet. Das erste und das letzte Feld (hier mit den Zahlen 0 und 32) teilen dem MIPS-Computer mit, dass mit diesem Befehl eine Addition durchzuführen ist. Das zweite Feld enthält die Nummer des Registers, das den ersten Quelloperanden der Addition enthält ( $17 = \$s1$ ), und das dritte Feld enthält den zweiten Quelloperanden für die Addition ( $18 = \$s2$ ). Das vierte Feld enthält die Nummer des Registers, in dem das Ergebnis gespeichert werden soll ( $8 = \$t0$ ). Das fünfte Feld wird in diesem Befehl nicht genutzt, daher ist es auf 0 gesetzt. Somit werden mit diesem Befehl die Inhalte der Register  $\$s1$  und  $\$s2$  addiert und das Ergebnis in das Register  $t0$  gespeichert.

Statt mit Dezimalzahlen in den einzelnen Feldern kann der Befehl auch mit Binärzahlen dargestellt werden:

000000	10001	10010	01000	00000	100000
6 Bit	5 Bit	5 Bit	5 Bit	5 Bit	6 Bit

Um diese Darstellung von der Assemblersprache zu unterscheiden, nennen wir diese numerische Version von Befehlen **Maschinensprache (machine language)** und eine Folge von Befehlen dieser Art als **Maschinencode**.

Diese Darstellungsform des Befehls wird als **Befehlsformat (instruction format)** bezeichnet. Wenn Sie die Anzahl der Bits zusammenzählen, erhalten Sie genau 32, exakt die Breite eines Datenworts. Entsprechend unserem Entwurfsprinzip bezüglich der Einfachheit und Regelmäßigkeit sind alle MIPS-Befehle 32 Bit lang.

Es könnte nun der Eindruck entstehen, dass Sie endlose, langweilige Folgen mit Binärzahlen lesen und schreiben müssten. Um dies zu vermeiden, nehmen wir eine höhere Basis als die 2, die sich aber leicht in die binäre Darstellung umrechnen lässt. Da praktisch alle Formate für Daten in einem Rechner ein Vielfaches von 4 sind, werden **Hexadezimalzahlen (hexadecimal)** (Basis 16) verwendet. Die Basis 16 ist eine Potenz von 2 weshalb man einfach jede Gruppe mit vier Binärziffern durch eine hexadezimale Ziffer ersetzen kann und umgekehrt. Tabelle 2.3 zeigt die Umrechnung von der hexadezimalen in die binäre Darstellung und umgekehrt.

**Tab. 2.3 Tabelle zur Umrechnung von Hexadezimal- in Binärzahlen und umgekehrt.** Ersetzen Sie einfach eine Hexadezimalziffer durch die entsprechenden vier Binärziffern und umgekehrt. Wenn die Länge der Binärzahl keinem Vielfachen von 4 entspricht, beginnen Sie rechts.

Hexadezimal	Binär	Hexadezimal	Binär	Hexadezimal	Binär	Hexadezimal	Binär
0 <sub>H</sub>	0000 <sub>B</sub>	4 <sub>H</sub>	0100 <sub>B</sub>	8 <sub>H</sub>	1000 <sub>B</sub>	c <sub>H</sub>	1100 <sub>B</sub>
1 <sub>H</sub>	0001 <sub>B</sub>	5 <sub>H</sub>	0101 <sub>B</sub>	9 <sub>H</sub>	1001 <sub>B</sub>	d <sub>H</sub>	1101 <sub>B</sub>
2 <sub>H</sub>	0010 <sub>B</sub>	6 <sub>H</sub>	0110 <sub>B</sub>	a <sub>H</sub>	1010 <sub>B</sub>	e <sub>H</sub>	1110 <sub>B</sub>
3 <sub>H</sub>	0011 <sub>B</sub>	7 <sub>H</sub>	0111 <sub>B</sub>	b <sub>H</sub>	1011 <sub>B</sub>	f <sub>H</sub>	1111 <sub>B</sub>

Da wir häufig mit unterschiedlichen Zahlenbasen zu tun haben, werden wir, um Verwechslungen zu vermeiden, Dezimalzahlen mit dem Index 10 (oder „D“), Binärzahlen mit dem Index 2 (oder „B“) und Hexadezimalzahlen mit dem Index 16 (oder „H“) versetzen. (Wenn kein Index angegeben wird, gilt Basis 10 als Standard.) Bei C und Java wird übrigens für Hexadezimalzahlen die Schreibweise `0xnnnn` verwendet.

### Umrechnung von Binärzahlen in Hexadezimalzahlen und umgekehrt

Rechnen Sie die folgenden Hexadezimal- und Binärzahlen in Zahlen der jeweils anderen Basis um:  $eca8\ 6420_H$

0001 0011 0101 0111 1001 1011 1101 1111<sub>B</sub>

**Maschinensprache (machine language)** Binäre Darstellung für die Kommunikation in einem Rechnersystem.

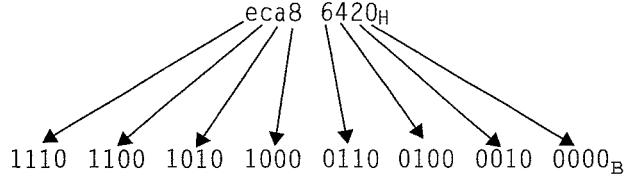
**Befehlsformat (instruction format)** Eine Darstellungsform für Befehle, zusammengesetzt aus Feldern mit Binärzahlen.

**Hexadezimalzahlen (hexadecimal)** Zahlen zur Basis 16.

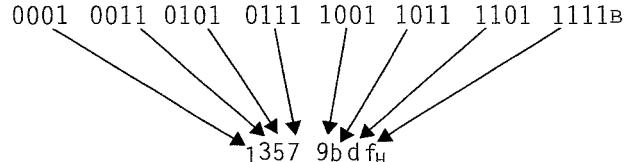
### BEISPIEL

**ANTWORT**

Mithilfe der Tabelle 2.3 gehen Sie schrittweise in der einen Richtung vor:



Und schließlich auch in der anderen Richtung:



## Die Felder im MIPS-Befehlsformat

Um die Diskussion zu vereinfachen, erhalten die Felder im MIPS-Befehlsformat Namen:

<b>op</b>	<b>rs</b>	<b>rt</b>	<b>rd</b>	<b>shamt</b>	<b>funct</b>
6 Bit	5 Bit	5 Bit	5 Bit	5 Bit	6 Bit

Die Namen der Felder im MIPS-Befehlsformat haben folgende Bedeutung:

- **op:** Basisoperation des Befehls, üblicherweise als **Opcode** (auch Operationscode) bezeichnet
- **rs:** Das Register des ersten Quelloperanden
- **rt:** Das Register des zweiten Quelloperanden
- **rd:** Das Zielregister in dem das Ergebnis der Operation gespeichert wird
- **shamt:** Abkürzung für *Shift Amount* („Anzahl der Stellen, um die verschoben wird“). (In Abschnitt 2.5 werden Schiebebefehle und dieser Begriff erläutert. Bis dahin wird dieser Ausdruck nicht verwendet, weshalb das Feld den Wert Null enthält.)
- **funct:** Dieses Feld wählt die spezielle Variante der Operation im op-Feld aus und wird auch als *Funktionscode (function)* bezeichnet.

Ein Problem entsteht, wenn ein Befehl längere Felder als die oben abgebildeten benötigt. Beispiel: Im Load-word-Befehl müssen zwei Register und eine Konstante angegeben werden. Wenn für die Adresse eines der 5-Bit-Felder im obigen Format verwendet würde, wäre die Konstante im Load-word-Befehl auf nur  $2^5$  oder 32 begrenzt. Die Konstante wird zum Auswählen von Elementen in Feldern oder Datenstrukturen verwendet und muss daher häufig wesentlich größer als 32 sein. Dieses 5-Bit-Feld ist somit einfach zu klein.

Wir haben also einen Konflikt zwischen dem Wunsch, für alle Befehle dieselbe Länge zu verwenden, und dem Wunsch, ein einheitliches Befehlsformat zu verwenden. Dies führt zum letzten Prinzip für den Hardwareentwurf:

*Entwurfsprinzip 4: Good design demands compromises (Ein guter Entwurf erfordert gute Kompromisse).*

Tab. 2.4 MIPS-Befehlscodierung. In der Tabelle steht „reg“ für eine Registernummer zwischen 0 und 31 und „address“ für eine 16-Bit-Adresse. „Entfällt“ bedeutet, dass es dieses Feld in dem jeweiligen Format nicht gibt. Die Befehle add und sub haben im op-Feld denselben Wert. Die Hardware entscheidet mithilfe des funct-Felds, welche Variante der Operation verwendet wird: add (32) oder subtract (34).

Befehl	Format	op	rs	rt	rd	shamt	funct	address
add	R	0	reg	reg	reg	0	$32_D$	entfällt
sub (subtract)	R	0	reg	reg	reg	0	$34_D$	entfällt
add immediate	I	$8_D$	reg	reg	entfällt	entfällt	entfällt	constant
lw (load word)	I	$35_D$	reg	reg	entfällt	entfällt	entfällt	address
sw (store word)	I	$43_D$	reg	reg	entfällt	entfällt	entfällt	address

Der von den MIPS-Entwicklern gewählte Kompromiss besteht darin, für alle Befehle dieselbe Länge und dafür für die verschiedenen Befehlsarten unterschiedliche Befehlsformate zu verwenden. So wird das obige Format beispielsweise als *R-Typ* (für Register) oder als *R-Format* bezeichnet. Ein weiterer Befehlsformattyp wird als *I-Typ* (für *immediate* = direkt) oder *I-Format* bezeichnet und für Immediate- und Datentransfer-Befehle verwendet. Für das I-Format gibt es folgende Felder:

op	rs	rt	constant oder address
6 Bit	5 Bit	5 Bit	16 Bit

Die 16-Bit-Adresse bedeutet, dass mit einem Load-word-Befehl ein beliebiges Wort in einem Bereich von  $\pm 2^{15}$  oder 65 536 Byte ( $\pm 2^{13}$  oder 16 384 Wörter) ab der Adresse im Basisregister rs geladen werden kann. Entsprechend ist der Add-immediate-Befehl auf Konstanten im Bereich von  $\pm 2^{15}$  beschränkt. (In Kapitel 3 wird erläutert, wie negative Zahlen dargestellt werden.) Wie wir sehen, wären mehr als 32 Register in diesem Format schwierig zu handhaben, da die Felder rs und rt jeweils ein weiteres Bit benötigten, wodurch es schwieriger wird, das Alles in einem Wort unterzubringen.

Betrachten wir noch einmal den Load-word-Befehl von Seite 45:

```
lw $t0, 32($s3) # lade temp. Reg. $t0 mit A[8]
```

Hier wird in das rs-Feld 19 (für  $\$s3$ ), in das rt-Feld 8 (für  $\$t0$ ) und in das Adressfeld 32 gesetzt. Die Bedeutung des rt-Felds hat sich bei diesem Befehl geändert: In einem Ladebefehl gibt das rt-Feld das Zielregister an, in dem das Ergebnis des Ladevorgangs gespeichert wird.

Mehrere Formate führen zwar zu einer komplizierteren Hardware, aber die Komplexität lässt sich reduzieren, wenn ähnliche Formate verwendet werden. So sind beispielsweise die ersten drei Felder der R- und I-Formate gleich groß und haben die gleichen Namen. Und das vierte Feld im I-Format ist gleich lang wie die letzten drei Felder im R-Format.

Falls Sie sich wundern: Die Formate unterscheiden sich durch die Werte im ersten Feld. Jedem Format ist eine Reihe von Werten im ersten Feld (op) zugewiesen, so dass die Hardware weiß, ob die zweite Hälfte des Befehls als drei Felder (R-Typ) oder als ein Feld (I-Typ) behandelt werden muss. In Tabelle 2.4 sind die in den einzelnen Feldern für die bis Abschnitt 2.3 beschriebenen MIPS-Befehle verwendeten Zahlen dargestellt.

**BEISPIEL****MIPS-Assemblersprache in Maschinensprache übersetzen**

Wir können nun an einem Beispiel den ganzen Weg von dem, was der Programmierer schreibt, hin zu dem, was der Rechner ausführt, aufzeigen. Wenn \$t1 auf die Basis des Felds A zeigt und \$s2 h entspricht, wird die Zuweisung

`A[300] = h + A[300];`

übersetzt in

```
lw $t0,1200($t1)    # temp. Reg. $t0
                      # wird zugewiesen A[300]
add $t0,$s2,$t0      # temp. Reg. $t0
                      # wird zugewiesen h + A[300]
sw $t0,1200($t1)    # speichere h + A[300] an die Stelle
                      # von A[300]
```

Wie lautet der Code in MIPS-Maschinensprache für diese drei Befehle?

**ANTWORT**

Der Einfachheit halber schreiben wir die Befehle in Maschinensprache zunächst als Dezimalzahlen. Aus Tabelle 2.4 können wir die drei Befehle in Maschinensprache ermitteln:

op	rs	rt	rd	address/shamt	funct
35	9	8		1200	
0	18	8	8	0	32
43	9	8		1200	

Der `lw`-Befehl wird durch die 35 im ersten Feld (op) (siehe Tabelle 2.4) angezeigt. Das Basisregister 9 (\$t1) wird im zweiten Feld (rs) und das Zielregister 8 (\$t0) im dritten Feld (rt) angegeben. Der Offset zum Auswählen von A[300] (1200 = 300 × 4) steht im letzten Feld (address).

Der nachfolgende `add`-Befehl wird durch die 0 im ersten Feld (op) und die 32 im letzten Feld (funct) spezifiziert. Die drei Registeroperanden (18, 8 und 8) stehen im zweiten, dritten und vierten Feld und entsprechen den Registern \$s2, \$t0 und \$t0.

Der `sw`-Befehl wird durch 43 im ersten Feld spezifiziert. Der Rest dieses letzten Befehls ist mit dem `lw`-Befehl identisch.

Die zur dezimalen Darstellung äquivalente binäre Form ist der folgenden Tabelle zu entnehmen (1200 zur Basis 10 entspricht 0000 0100 1011 0000 zur Basis 2):

100011	01001	01000	0000 0100 1011 0000	
000000	10010	01000	01000	00000 100000
101011	01001	01000	0000 0100 1011 0000	

Beachten Sie die Ähnlichkeit der Binärdarstellung des ersten und letzten Befehls. Die beiden Darstellungen unterscheiden sich lediglich durch das dritte Bit von links.

In Tabelle 2.5 sind die in diesem Abschnitt beschriebenen Teile der MIPS-Assemblersprache zusammenfassend dargestellt. Wie wir in Kapitel 5 und 6 noch sehen werden, wird der Hardwareentwurf durch die Ähnlichkeit der Binärdarstellungen von ähnlichen Befehlen vereinfacht. Diese Befehle sind ein weiteres Beispiel für die Regelmäßigkeit in der MIPS-Architektur.

**Tab. 2.5 Die bis Abschnitt 2.4 bearbeitete MIPS-Architektur.** In Abschnitt 2.4 neu eingeführte Strukturteile der MIPS-Befehlssprache sind farblich hervorgehoben. Die beiden bisher bekannten MIPS-Befehlsformate sind der R-Typ und der I-Typ. Die ersten 16 Bit sind identisch: beide enthalten ein op-Feld, das die Grundoperation angibt; ein rs-Feld, das einen der Quelloperanden angibt; und das rt-Feld, das den anderen Quelloperanden angibt, außer beim Load-word-Befehl, bei dem es das Zielregister angibt. Beim R-Format sind die letzten 16 Bit auf drei Felder aufgeteilt: das rd-Feld, das das Zielregister angibt, das shamt-Feld, das in Abschnitt 2.5 erläutert wird und das funct-Feld, das die spezifische Operation eines R-Formatbefehls angibt. Beim I-Format bilden die letzten 16 Bit ein address-Feld.

#### MIPS-Operanden

Name	Beispiel	Anmerkungen
32 Register	$\$s0, \$s1, \dots, \$s7$ $\$t0, \$t1, \dots, \$t7$	Speicherort für schnellen Zugriff auf Daten. Bei MIPS müssen die Daten für die arithmetischen Operationen in Registern stehen. Register $\$s0-\$s7$ werden 16–23 und $\$t0-\$t7$ 8–15 zugeordnet.
$2^{30}$ Speicherwörter	Speicher[0], Speicher[4], ..., Speicher[4294967292]	Zugriff in der MIPS-Architektur nur durch Datentransport-Befehle. Die MIPS-Architektur verwendet Byteadressen. Daher unterscheiden sich aufeinander folgende Wortadressen um 4. Im Hauptspeicher werden Datenstrukturen, Felder und ausgelagerte Register gespeichert.

#### MIPS-Assemblersprache

Kategorie	Befehl	Beispiel	Bedeutung	Anmerkungen
Arithmetischer Befehl	add	add \$s1,\$s2, \$s3	$\$s1 = \$s2 + \$s3$	Drei Operanden; Daten in Registern
	subtract	sub \$s1,\$s2, \$s3	$\$s1 = \$s2 - \$s3$	Drei Operanden; Daten in Registern
Daten-transfer	load word	lw \$s1, 100(\$s2)	$\$s1 = \text{Speicher}[\$s2 + 100]$	Daten vom Hauptspeicher in ein Register
	store word	sw \$s1, 100(\$s2)	$\text{Speicher}[\$s2 + 100] = \$s1$	Daten von einem Register in den Hauptspeicher

#### MIPS-Maschinensprache

Name	Format	Beispiel							Anmerkungen
add	R	0	18	19	17	0	32		add \$s1,\$s2, \$s3
sub	R	0	18	19	17	0	34		sub \$s1,\$s2, \$s3
addi	I	8	18	17	100				addi \$s1,\$s2,100
lw	I	35	18	17	100				lw \$s1,100(\$s2)
sw	I	43	18	17	100				sw \$s1,100(\$s2)
Feldgröße		6 Bit	5 Bit	5 Bit	5 Bit	5 Bit	6 Bit		Alle MIPS-Befehle 32 Bit lang
R-Format	R	op	rs	rt	rd	shamt	funct		Format für arithmetische Befehle
I-Format	I	op	rs	rt	address				Format für Datentransport

Warum gibt es im MIPS-Befehlssatz keinen Subtract-immediate-Befehl?

1. Negative Konstanten kommen in C und Java viel seltener vor. Sie sind also nicht der häufig vorkommende Fall („common case“) und bedürfen daher keiner speziellen Unterstützung.
2. Da im Immediate-Feld sowohl negative als auch positive Konstanten stehen können, entspricht ein Add-immediate-Befehl mit einer negativen Zahl einem Subtract-immediate-Befehl mit einer positiven Zahl, daher ist der Subtract-immediate-Befehl überflüssig.





Computer von heute beruhen auf zwei Grundprinzipien:

1. Befehle werden in Form von Zahlen dargestellt.
2. Programme werden wie Zahlen im Hauptspeicher gespeichert, um gelesen oder geschrieben werden zu können.

Diese Prinzipien führen zum *Von-Neumann-Konzept*. Diese Erfindung öffnete dem Geist der Datenverarbeitung die Flasche. In Abbildung 2.3 wird die Leistungsfähigkeit dieses Konzepts deutlich: Im Hauptspeicher kann der Quellcode für einen Editor, der entsprechende kompilierte Maschinencode, der Text, der vom kompilierten Programm verwendet wird, und sogar der Compiler, der den Maschinencode generiert, gespeichert werden.

Die Tatsache, dass Befehle in Form von Zahlen dargestellt werden können, hat zur Folge, dass Programme häufig als Dateien mit Binärzahlen ausgeliefert werden. Die kommerzielle Folge hiervon ist, dass Rechner fertige Programme übernehmen können, vorausgesetzt sie sind zu einem vorhandenen Befehlssatz kompatibel. Diese „Binärkompatibilität“ führt dazu, dass sich die Industrie auf wenige Befehlssatzarchitekturen konzentriert.



**Vertiefung:** Die Darstellung von Dezimalzahlen in Zahlen zur Basis 2 eröffnet eine einfache Möglichkeit, positive ganze Zahlen in Wörtern darzustellen. In Kapitel 3 wird erläutert, wie negative Zahlen dargestellt werden. Im Moment müssen Sie einfach glauben, dass mit einem 32-Bit-Wort ganze Zahlen im Bereich von  $-2^{31}$  bis  $+2^{31}-1$  oder von -2 147 483 648 bis +2 147 483 647 dargestellt werden können, und dass im constant-Feld mit 16 Bit tatsächlich Zahlen zwischen  $-2^{15}$  und  $+2^{15}-1$  oder von -32 768 bis 32 767 angegeben werden können. Diese ganzen Zahlen werden als *Zweierkomplementzahlen* bezeichnet. In Kapitel 3 ist dargestellt, wie wir addi \$t0,\$t0,-1 oder lw \$t0,-4(\$s0) als Code schreiben würden, wobei im I-Format im constant-Feld negative Zahlen erforderlich sind.

„Contrariwise,“ continued Tweedledee, „if it was so, it might be; and if it were so, it would be; but as it isn't, it ain't. That's logic.“

Lewis Carroll, *Alice's Adventures in Wonderland*, 1865.

## 2.5

## Logische Operationen

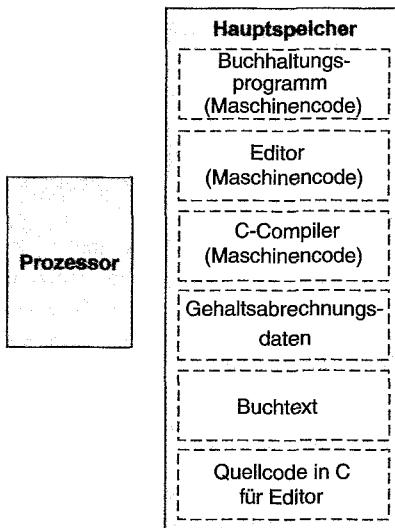
Obwohl bei den ersten Rechnern vornehmlich ganze Wörter betrachtet wurden, stellte sich rasch heraus, dass es sinnvoll ist, auf Bitfelder in einem Wort oder auch auf einzelne Bits zugreifen zu können. Die Überprüfung von Zeichen in einem Wort, die mit jeweils 8 Bit gespeichert sind, ist ein Beispiel für eine Operation dieser Art. Dies führte dazu, dass Operationen hinzugefügt wurden, mit denen unten anderem das Setzen und Zurücksetzen von Bits in einem Wort vereinfacht wurde. Diese Befehle werden als logische Operationen bezeichnet. In Tabelle 2.6 sind logische Operationen in C und Java dargestellt.

Die erste Klasse von Operationen dieser Art sind *Schiebeoperationen*. Sie schieben alle Bits in einem Wort nach links oder nach rechts, wobei die frei werdenden Bits mit einer Null aufgefüllt werden. Beispiel: Wenn Register \$s0 die Bitfolge

0000 0000 0000 0000 0000 0000 0000 1001<sub>B</sub> = 9<sub>D</sub>

enthält und der Befehl zum Schieben um 4 nach links ausgeführt wird, ergibt sich folgender neuer Wert:

0000 0000 0000 0000 0000 0000 1001 0000<sub>B</sub> = 144<sub>D</sub>



**Abb. 2.3 Das Von-Neumann-Konzept.** Mithilfe von gespeicherten Programmen kann ein Rechner, der ein Buchhaltungsprogramm ausführt, im nächsten Augenblick zu einem Rechner werden, der einem Autor hilft, ein Buch zu schreiben. Dieser Wechsel erfolgt durch Laden von Programmen und Daten in den Speicher und durch Anweisen des Rechners, an einer bestimmten Position im Speicher mit der Ausführung zu beginnen. Dadurch, dass Befehle wie Daten behandelt werden, wird sowohl die Speicherhardware als auch die Software von Rechnersystemen erheblich vereinfacht. So kann insbesondere die für Daten erforderliche Spechertechnologie auch für Programme verwendet werden, und Programme – wie z.B. Compiler – können Code, der in einer für Menschen einfacheren Form geschrieben ist, in einen Code übersetzen, der vom Rechner verstanden wird.

**Tab. 2.6 Logische C- und Java-Operatoren und die entsprechenden MIPS-Befehle.** MIPS verwendet NOR mit einem Operand gleich null, um NOT zu implementieren.

Logische Operationen	C-Operatoren	Java-Operatoren	MIPS-Befehle
Linksschieben	<code>&lt;&lt;</code>	<code>&lt;&lt;</code>	<code>sll</code>
Rechtsschieben	<code>&gt;&gt;</code>	<code>&gt;&gt;</code>	<code>srl</code>
Bitweise UND-Verknüpfung	<code>&amp;</code>	<code>&amp;</code>	<code>and, andi</code>
Bitweise ODER-Verknüpfung	<code> </code>	<code> </code>	<code>or, ori</code>
Bitweise NOT	<code>~</code>	<code>~</code>	<code>nor</code>

Die duale Operation zum Schieben nach links ist das Schieben nach rechts. Die beiden MIPS-Schiebebefehle heißen *Shift Left Logical* (logisches Linksschieben, `sll`) und *Shift Right Logical* (logisches Rechtsschieben, `srl`).

Mit dem folgenden Befehl wird die obige Operation ausgeführt und das Ergebnis in Register `$t2` gespeichert:

```
sll $t2,$s0,4 # Reg. $t2 = Reg. $s0 << 4 Bit
```

Das *shamt*-Feld im R-Format haben wir nicht gleich beim ersten Auftreten des Ausdrucks erläutert. Der Ausdruck *shamt* steht für *Shift Amount* (Anzahl der Stellen, um die verschoben wird) und wird in Schiebebefehlen verwendet. Die Version des obigen Befehls in Maschinensprache lautet somit wie folgt:

op	rs	rt	c	rd	shamt	funct
0	0	16	10	4	0	0

Der Code von `sll` lautet sowohl im op- als auch im funct-Feld 0, das rd-Feld enthält `$t2`, das rt-Feld enthält `$s0` und das shamt-Feld enthält 4. Das rs-Feld wird nicht verwendet und ist daher auf 0 gesetzt.

Das logische Schieben nach links bringt einen weiteren Vorteil mit sich. Wenn um  $i$  Bit nach links verschoben wird, ergibt dies dasselbe Ergebnis wie die Multiplikation mit  $2^i$ . (In Kapitel 3 wird erläutert, warum das so ist.) Beispiel: Mit dem obigen `sll`-Befehl wird um 4 Stellen verschoben, was dasselbe ergibt wie die Multiplikation mit  $2^4$  oder mit 16. Das erste Bitmuster oben stellt 9 dar und  $9 \times 16 = 144$  den Wert des zweiten Bitmusters.

Eine weitere sinnvolle Operation zum Isolieren von Feldern ist die *UND-Verknüpfung*. (Um eine Verwechslung mit der natürlichsprachlichen Konjunktion zu vermeiden, werden die Namen der Verknüpfungsoperationen in Großbuchstaben geschrieben.) Bei der UND-Verknüpfung handelt es sich um eine bitweise Operation, bei der das Ergebnis nur dann eine 1 ist, wenn an den entsprechenden Bitstellen der Operanden jeweils der Wert 1 steht. Beispiel: Wenn Register `$t2` nach wie vor die Bitfolge

0000 0000 0000 0000 0000 1101 0000 0000<sub>B</sub>

enthält und Register `$t1` die Bitfolge

0000 0000 0000 0000 0011 1100 0000 0000<sub>B</sub>

enthält, ergibt sich nach dem Ausführen des MIPS-Befehls

`and $t0,$t1,$t2 # Reg. $t0 = Reg. $t1 & Reg.$t2`

für den Wert von Register `$t0` die Bitfolge

0000 0000 0000 0000 0000 1100 0000 0000<sub>B</sub>

Mit der UND-Verknüpfung kann ein Bitmuster auf eine Menge von Bits angewendet werden, um an den Stellen jeweils eine Null zu erzwingen, an denen sich im Bitmuster eine Null befindet. Ein derartiges Bitmuster mit einer UND-Verknüpfung wird als „Maske“ bezeichnet, da die Maske einige Bits „verbirgt“.

Um einer Menge von Bitstellen mit einer Null einen Wert zuzuweisen, gibt es die duale Operation zur UND-Verknüpfung, die *ODER-Verknüpfung*. Hierbei handelt es sich um eine bitweise Operation, bei der das Ergebnis 1 ist, wenn *einer* der Operandenbits eine 1 aufweist. Mit dem obigen Beispiel kann die Wirkung der ODER-Verknüpfung verdeutlicht werden. Wenn die Register `$t1` und `$t2` aus dem vorhergehenden Beispiel unverändert bleiben, ergibt der MIPS-Befehl

`or $t0,$t1,$t2 # Reg. $t0 = Reg. $t1 | Reg. $t2`

den folgenden Wert in Register `$t0`:

0000 0000 0000 0000 0011 1101 0000 0000<sub>B</sub>

Bei der letzten logischen Operation handelt es sich um die Negation. **NICHT (NOT)** ergibt 1, wenn ein Operandenbit den Wert 0 hat und umgekehrt. Um das Format mit zwei Operanden beizubehalten, haben sich die Entwickler von MIPS für die Aufnahme des Befehls **NOR (NICHT ODER)** anstelle der Negation entschieden. Wenn ein Operand null ist, entspricht er einem NICHT. Beispiel: A NOR 0 = NICHT (A ODER 0) = NICHT (A).

**NICHT (NOT)** Eine logische bitweise Operation, bei der ein Operand die Bitwerte invertiert, d.h. er ersetzt jede 1 durch eine 0 und jede 0 durch eine 1.

**NOR** Eine logische bitweise Operation mit zwei Operanden, mit der die Negation des Ergebnisses einer ODER-Verknüpfung von zwei Operanden berechnet wird.

Tab. 2.7 Bisher bearbeitete MIPS-Architektur. Die seit Tabelle 2.5 beschriebenen Teile sind farblich hervorgehoben. Außerdem finden Sie im Buch eine herausnehmbare Karte mit den MIPS-Maschinenbefehlen.

### MIPS-Operanden

Name	Beispiel	Anmerkungen
32 Register	\$s0, \$s1, ..., \$s7 \$t0, \$t1, ..., \$t7	Speicherort für schnellen Zugriff auf Daten. Bei MIPS müssen die Daten für die arithmetischen Operationen in Registern stehen. Register \$s0~\$s7 werden 16–23 und \$t0~\$t7 8–15 zugeordnet.
$2^{30}$ Speicherwörter	Speicher[0], Speicher[4], ..., Speicher[4294967292]	Zugriff nur durch Datentransport-Befehle. Die MIPS-Architektur verwendet Byteadressen. Daher unterscheiden sich aufeinander folgende Wortadressen um 4. Im Hauptspeicher werden Datenstrukturen, Felder und ausgelagerte Register gespeichert.

### MIPS-Assemblersprache

Kategorie	Befehl	Beispiel	Bedeutung	Anmerkungen
Arithmetischer Befehl	add	add \$s1,\$s2,\$s3	$\$s1 = \$s2 + \$s3$	Drei Operanden; Überlauf feststellen
	subtract	sub \$s1,\$s2,\$s3	$\$s1 = \$s2 - \$s3$	Drei Operanden; Überlauf feststellen
	add immediate	addi \$s1,\$s2, 100	$\$s1 = \$s2 + 100$	+ Konstante; Überlauf feststellen
Logischer Befehl	and	and \$s1,\$s2,\$s3	$\$s1 = \$s2 \& \$s3$	Drei Registeroperanden; bitweise UND-Verknüpfung
	or	or \$s1,\$s2,\$s3	$\$s1 = \$s2 \mid \$s3$	Drei Registeroperanden; bitweise ODER-Verknüpfung
	nor	nor \$s1,\$s2,\$s3	$\$s1 = \sim (\$s2 \mid \$s3)$	Drei Registeroperanden; bitweise NOR-Verknüpfung
	and immediate	andi \$s1,\$s2,100	$\$s1 = \$s2 \& 100$	Bitweise UND-Verknüpfung von Registeroperanden mit Konstante
	or immediate	ori \$s1,\$s2,100	$\$s1 = \$s2 \mid 100$	Bitweise ODER-Verknüpfung von Registeroperanden mit Konstante
	shift left logical	sll \$s1,\$s2,10	$\$s1 = \$s2 \ll 10$	Linksschieben um Konstante
Daten-transfer	shift right logical	srl \$s1,\$s2,10	$\$s1 = \$s2 \gg 10$	Rechtsschieben um Konstante
	load word	lw \$s1, 100 (\$s2)	$\$s1 = \text{Speicher}[\$s2 + 100]$	Wort vom Hauptspeicher in ein Register
	store word	sw \$s1, 100(\$s2)	$\text{Speicher}[\$s2 + 100] = \$s1$	Wort von einem Register in den Hauptspeicher

*The utility of an automatic computer lies in the possibility of using a given sequence of instructions repeatedly, the number of times it is iterated being dependent upon the results of the computation. When the iteration is completed a different sequence of [instructions] is to be followed, so we must, in most cases, give two parallel trains of [instructions] preceded by an instruction as to which routine is to be followed. This choice can be made to depend upon the sign of a number (zero being reckoned as plus for machine purposes). Consequently, we introduce an [instruction] (the conditional transfer [instruction]) which will, depending on the sign of a given number, cause the proper one of two routines to be executed.*

Burks, Goldstine und von Neumann, 1947

**Bedingte Verzweigung (conditional branch)** Ein Befehl, bei dem zunächst zwei Werte verglichen werden, um in Abhängigkeit vom Ergebnis dieses Vergleichs den Kontrollfluss zu ändern.

## BEISPIEL

Wenn das Register \$t1 aus dem vorhergehenden Beispiel unverändert bleibt und Register \$t3 den Wert 0 aufweist, ergibt der MIPS-Befehl

```
nor $t0,$t1,$t3 # Reg. $t0 = ~ (Reg. $t1 | Reg. $t3)
```

den folgenden Wert in Register \$t0 :

```
1111 1111 1111 1111 1100 0011 1111 1111B
```

In Tabelle 2.6 weiter oben ist die Beziehung zwischen C- und Java-Operatoren und den MIPS-Befehlen dargestellt. Konstanten sind sowohl in logischen UND- und ODER-Operationen als auch in arithmetischen Operationen hilfreich. Daher gibt es im MIPS-Befehlssatz auch die Befehle *and immediate* (andi) und *or immediate* (ori).

Für NOR werden selten Konstanten verwendet, da dieser Operator zum Umkehren der Bits eines einzelnen Operanden verwendet wird. Daher gibt es in der Hardware keine Immediate-Version dieses Operators. In Tabelle 2.5, in der eine Übersicht über die bisher bekannten MIPS-Befehle dargestellt ist, sind die logischen Befehle farblich hervorgehoben.

## 2.6

# Befehle zum Treffen von Entscheidungen

Ein Computer unterscheidet sich von einem einfachen Taschenrechner dadurch, dass er Entscheidungen treffen kann. Abhängig von den Eingabedaten und den während der Berechnung erhaltenen Werten werden unterschiedliche Befehle ausgeführt. Entscheidungen werden in Programmiersprachen in der Regel mithilfe der *If*-Anweisung, gelegentlich zusammen mit *Go-to*-Anweisungen und Sprungmarken dargestellt. Die MIPS-Assemblersprache enthält zwei Entscheidungsbefehle ähnlich einer *If*-Anweisung mit *go to*. Der erste Befehl lautet

```
beq register1, register2, L1
```

Bei der Ausführung dieses Befehls wird zur Anweisung an der Marke L1 verzweigt, wenn der Wert in register1 gleich dem Wert in register2 ist. Die mnemonische Bezeichnung *beq* steht für *branch if equal* („verzweige, wenn gleich“). Der zweite Befehl lautet

```
bne register1, register2, L1
```

Dieser Befehl verzweigt zur Anweisung an der Marke L1, wenn der Wert in register1 *nicht gleich* dem Wert in register2 ist. Die mnemonische Bezeichnung *bne* steht für *branch if not equal* („verzweige, wenn nicht gleich“). Diese beiden Befehle werden als **bedingte Verzweigungen (conditional branch)** bezeichnet.

### Übersetzung einer If-then-else-Anweisung in eine bedingte Verzweigung

Im folgenden Codesegment sind f, g, h, i und j Variablen. Wenn die fünf Variablen f bis j den fünf Registern \$s0 bis \$s4 entsprechen, wie lautet dann der übersetzte MIPS-Code für diese *If*-Anweisung in C?

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

In Abbildung 2.4 ist in einem Flussdiagramm dargestellt, was der MIPS-Code bewirken soll. Mit dem ersten Ausdruck wird die Gleichheit überprüft, weshalb wohl `beq` der geeignete Befehl zu sein scheint. Im Allgemeinen wird der Code effizienter, wenn wir überprüfen, ob die gegenteilige Bedingung erfüllt ist, um den Code so zu verzweigen, dass der nachfolgende *Then*-Teil der *If*-Anweisung ausgeführt wird (die Marke `Else` wird unten definiert):

```
bne $s3,$s4,Else # verzweige zu Else, wenn i ≠ j
```

Die nächste Zuweisung führt eine Operation aus und, wenn alle Operanden bereits den Registern zugeteilt sind, wird dafür nur ein Befehl benötigt:

```
add $s0,$s1,$s2 # f = g + h (wird ausgelassen,  
# wenn i ≠ j)
```

Nach diesem Befehl muss das Ende der *If*-Anweisung erreicht werden. Mit diesem Beispiel lernen wir eine weitere Art der Verzweigung kennen, die als *unbedingte Verzweigung* bezeichnet wird. Diese Anweisung besagt, dass der Prozessor die Verzweigung immer ausführt. Um zwischen bedingten und unbedingten Verzweigungen zu unterscheiden, wird diese Art des Befehls in der MIPS-Assemblersprache als *Sprung* bezeichnet und mit `j` (jump, engl. für Sprung) abgekürzt. (Die Marke `Exit` wird unten definiert.)

```
j Exit # springe zu Exit
```

Die Zuweisung im *Else*-Teil der *If*-Anweisung kann wieder mit einem Befehl übersetzt werden. Darüber hinaus muss bei diesem Befehl die Marke `Else` stehen. Wir zeigen außerdem die Marke `Exit`, die nach diesem Befehl steht und damit das Ende des übersetzten Codes für eine *If-then-else*-Anweisung angezeigt:

```
Else: sub $s0,$s1,$s2 # f = g - h (wird ausgelassen,  
# wenn i = j)
```

```
Exit:
```

## ANTWORT

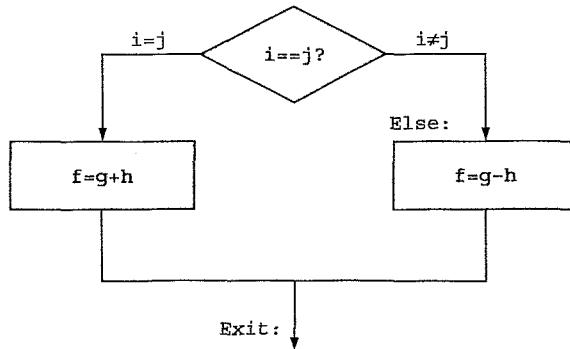


Abb. 2.4 Darstellung der Alternativen der obigen *If*-Anweisung. Das linke Kästchen entspricht dem *Then*-Teil der *If*-Anweisung und das rechte Kästchen dem *Else*-Teil.

Der Assembler nimmt dem Compiler und dem in Assemblersprache Programmierenden die Arbeit ab, Adressen für Verzweigungen berechnen zu müssen, ebenso wie der Assembler auch die Adressen der Daten für die Lade- und Speicherbefehle berechnet (siehe Abschnitt 2.10).

## Hardware-Software-Schnittstelle

Compiler erzeugen häufig Verzweigungen und Sprungmarken an Stellen, an denen diese in der Programmiersprache nicht vorkommen. Dass Sprungmarken und Verzweigungen nicht explizit geschrieben werden müssen, ist einer der Vorteile höherer Programmiersprachen und ein Grund dafür, warum sich Code auf dieser Ebene schneller schreiben lässt.

## Schleifen

Abfragen sind sowohl für die Wahl zwischen zwei Alternativen (bei *If*-Anweisungen) als auch für die Iteration einer Berechnung (bei Schleifen) wichtig. In beiden Fällen dienen dieselben Assemblerbefehle als Grundbausteine.

## BEISPIEL

### Eine *While*-Schleife in C übersetzen

So sieht eine typische Schleife in C aus:

```
while (save[i] == k)
    i += 1;
```

Nehmen wir an, dass *i* und *k* den Registern \$s3 und \$s5 zugeteilt sind und dass die Basis des Felds *save* in \$s6 gespeichert ist. Wie lautet der in MIPS-Assemblersprache geschriebene Code für dieses C-Segment?

## ANTWORT

Im ersten Schritt muss das Element *i* des Felds *save*, also *save[i]* in ein temporäres Register geladen werden. Hierfür benötigen mir zunächst die Adresse der Speicherzelle von *save[i]*. Zur Bildung der Adresse muss der Index *i* wegen des Byteadressierungsproblems mit 4 multipliziert werden und kann dann auf die Basis des Felds *save* addiert werden. Für die Multiplikation mit 4 können wir die logische Schiebeoperation nach links verwenden, da ein Schieben um zwei Bit nach links das gleiche Ergebnis liefert, wie die Multiplikation mit 4. Wir müssen die Marke *Loop* setzen, so dass am Ende der Schleife zu diesem Befehl zurückgesprungen werden kann:

```
Loop: sll $t1,$s3,2 # temp. Reg. $t1 = 4 * i
```

Um die Adresse von *save[i]* zu erhalten, müssen wir \$t1 und die Basis von *save* in \$s6 addieren:

```
add $t1,$t1,$s6 # $t1 = Adresse von save[i]
```

Nun können wir mithilfe dieser Adresse *save[i]* in ein temporäres Register laden:

```
lw $t0,0($t1) # temp. Reg. $t0 = save[i]
```

Mit dem nächsten Befehl wird die Schleifenabbruchbedingung geprüft. Falls *save[i]* ≠ *k* wird die Schleife verlassen:

```
bne $t0,$s5, Exit # verzweige zu Exit,
                    # wenn save[i] ≠ k
```

Mit dem nächsten Befehl werden *i* und *i* addiert:

```
addi $s3,$s3,1 # i = i + 1
```

Am Ende der Schleife wird zurück an den Anfang der Schleife gesprungen. Danach brauchen wir nur noch die `Exit`-Marke einzufügen, und schon sind wir fertig:

```
j Loop # springe zu Loop
Exit:
```

Befehlsfolgen, die mit einem Sprung enden, spielen beim Übersetzungsvorgang eine zentrale Rolle, weshalb sie einen eigenen Namen erhalten haben: So wird eine Befehlsfolge ohne Sprünge, außer möglicherweise am Ende der Befehlsfolge, und ohne Sprungziel oder Sprungmarke, außer möglicherweise am Anfang der Befehlsfolge, als **Grundblock** oder **Basisblock** (*basic block*) bezeichnet. Einer der ersten Schritte beim Kompilieren besteht darin, das Programm in Grundblöcke zu zerlegen.

Am häufigsten wird wohl die Gleichheit oder Ungleichheit zweier Werte geprüft. Gelegentlich ist es jedoch hilfreich festzustellen, ob eine Variable kleiner als eine andere Variable ist. Beispielsweise ist es bei einer *For*-Schleife manchmal sinnvoll abzufragen, ob die Index-Variable kleiner als 0 ist. Vergleiche dieser Art werden in der MIPS-Assemblersprache mit einem Befehl durchgeführt, der die Inhalte zweier Register miteinander vergleicht und ein drittes Register auf 1 setzt, wenn der Wert im ersten Register kleiner als der im zweiten ist. Wenn das erste Register nicht kleiner als das zweite ist, wird das dritte Register auf 0 gesetzt. Dieser MIPS-Befehl lautet *set on less than* oder `slt`. Beispiel:

```
slt $t0, $s3, $s4
```

bedeutet, dass Register `$t0` auf 1 gesetzt wird, wenn der Wert in Register `$s3` kleiner als der Wert in Register `$s4` ist. Andernfalls wird Register `$t0` auf 0 gesetzt.

Konstanten als Operanden werden gerne für Vergleiche herangezogen. Da Register `$zero` immer den Wert 0 hat, können wir bereits Vergleiche mit 0 durchführen. Zum Vergleich mit anderen Werten gibt es eine Immediate-Version des Set-on-less-than-Befehls. Um zu prüfen, ob Register `$s2` kleiner als die Konstante 10 ist, können wir einfach Folgendes schreiben:

```
slti $t0,$s2,10 # $t0 = 1, wenn $s2 < 10
```

Von Neumanns Warnung hinsichtlich der Einfachheit des Systems berücksichtigend enthält die MIPS-Architektur keinen *Branch-on-less-than*-Befehl, da er zu kompliziert ist. Für einen solchen Befehl würde es entweder notwendig sein, die Taktzykluszeit zu verlängern oder es würden zusätzliche Taktzyklen pro Maschinenbefehl erforderlich sein. Zwei schnellere Befehle sind hier sinnvoller.

MIPS-Compiler erstellen mithilfe der Befehle `slt`, `slti`, `beq`, `bne` und dem Wert 0 (immer verfügbar durch Lesen des Registers `$zero`) alle relativen Bedingungen: ist gleich, ist nicht gleich, kleiner als oder gleich, größer als, größer als oder gleich. (Wie zu erwarten, ist Register `$zero` auf das Register 0 abgebildet.)

## Hardware-Software-Schnittstelle

**Grundblock oder Basisblock (*basic block*)** Eine Befehlsfolge ohne Sprünge (außer möglicherweise am Ende der Befehlsfolge) und ohne Sprungziel oder Sprungmarke (außer möglicherweise am Anfang der Befehlsfolge).

## Hardware-Software-Schnittstelle

## Case-/switch-Anweisung

Die meisten Programmiersprachen enthalten eine *Case*- oder *Switch*-Anweisung, mit deren Hilfe der Programmierer auf der Grundlage eines Wertes eine von mehreren Alternativen auswählen kann. Die *Switch*-Anweisung lässt sich am einfachsten über eine Folge von Bedingungsabfragen implementieren, wodurch die *Switch*-Anweisung zu einer Kette von *If-then-else*-Anweisungen wird.

Manchmal können die Alternativen effizienter in Form einer Tabelle mit Adressen von alternativen Befehlsfolgen kodiert werden. Diese Tabelle wird als *Sprungadresstabelle (jump address table)* bezeichnet, und das Programm muss dann nur noch die Tabelle indizieren und zur entsprechenden Befehlsfolge springen. So mit handelt es sich bei der Sprungadresstabelle einfach um ein Feld von Wörtern mit Adressen, die Marken im Programm entsprechen.

Für Situationen wie diese enthalten Prozessoren wie MIPS einen *Jump-register*-Befehl (*j r*), der einen unbedingten Sprung zu der in einem Register enthaltenen Adresse bewirkt. Das Programm lädt den entsprechenden Eintrag aus der Sprungadresstabelle in ein Register und springt dann mithilfe eines *Jump-register*-Befehls zur entsprechenden Adresse. Dieser Befehl wird in Abschnitt 2.7 beschrieben.

## Hardware- Software- Schnittstelle

Obwohl Programmiersprachen wie C und Java viele Anweisungen für Entscheidungen und Schleifen enthalten, ist die zugrundeliegende Anweisung, mit der diese Anweisung auf der nächst tieferen Ebene implementiert wird, eine bedingte Verzweigung.

In Tabelle 2.8 sind die in diesem Abschnitt beschriebenen Teile der MIPS-Assemblersprache und in Tabelle 2.9 die entsprechenden Befehle in MIPS-Maschinensprache zusammengefasst. Bei diesem Schritt in der Darlegung der MIPS-Sprache wurden Verzweigungen und Sprünge der symbolischen Darstellung hinzugefügt und der nützliche Wert 0 fest in einem Register abgelegt.



**Vertiefung:** Wenn Sie von verzögerten Sprüngen (siehe Kapitel 6) gehört haben, brauchen Sie sich keine Sorgen zu machen: Der MIPS-Assembler blendet diese für die in Assemblersprache Programmierenden aus.



In C gibt es viele Anweisungen für Abfragen und Schleifen, während MIPS nur wenige kennt. Welche der folgenden Aussagen erklären diese Unausgewogenheit bzw. welche Aussagen erklären den Unterschied nicht? Warum?

1. Je mehr Abfrageanweisungen, umso einfacher ist Code zu lesen und zu verstehen.
2. Je weniger Entscheidungsanweisungen, umso leichter hat es die darunter liegende Schicht, die für die Ausführung verantwortlich ist.
3. Je mehr Entscheidungsanweisungen, umso weniger Codezeilen sind erforderlich, wodurch sich Code schneller schreiben lässt.
4. Je mehr Entscheidungsanweisungen, umso weniger Codezeilen, und umso weniger Operationen müssen ausgeführt werden.

Warum gibt es bei C zwei Operatoren für UND (`&` und `&&`) und zwei Operatoren für ODER (`|` und `||`), bei MIPS jedoch nicht?

1. Mit den logischen Operationen UND und ODER werden die Operatoren `&` und `|` implementiert, während mit bedingten Verzweigungen die Operatoren `&&` und `||` implementiert werden.

2. Die obige Aussage gilt genau umgekehrt: `&&` und `||` entsprechen logischen Operationen, `&` und `|` entsprechen bedingten Verzweigungen.
  3. Die zweiten Operatoren sind redundant und bedeuten dasselbe wie die ersten Operatoren: `&&` und `||` wurden einfach aus der Programmiersprache B, der Vorgängersprache von C, übernommen.
- 

**Tab. 2.8 Bis Abschnitt 2.6 bearbeitete MIPS-Architektur.** Die in Abschnitt 2.6 neu eingeführten MIPS-Strukturen sind farblich hervorgehoben.

MIPS-Operanden		
Name	Beispiel	Anmerkungen
32 Register	<code>\$s0, \$s1, ..., \$s7</code> <code>\$t0, \$t1, ..., \$t7,</code> <code>\$zero</code>	Speicherort für schnellen Zugriff auf Daten. Bei MIPS müssen die Daten für die arithmetischen Operationen in Registern stehen. Register <code>\$s0–\$s7</code> werden 16–23 und <code>\$t0–\$t7</code> 8–15 zugeordnet. MIPS-Register <code>\$zero</code> ist immer 0.
$2^{30}$ Speicherwörter	<code>Speicher[0],</code> <code>Speicher[4], ... ,</code> <code>Speicher[4294967292]</code>	Zugriff nur durch Datentransport-Befehle. Die MIPS-Architektur verwendet Byteadressen. Da-her unterscheiden sich aufeinander folgende Wortadressen um 4. Im Hauptspeicher werden Datenstrukturen, Felder und ausgelagerte Register gespeichert.

MIPS-Assemblersprache				
Kategorie	Befehl	Beispiel	Bedeutung	Anmerkungen
Arithmetischer Befehl	add	<code>add \$s1, \$s2, \$s3</code>	$\$s1 = \$s2 + \$s3$	Drei Operanden; Daten in Regis-tern
	subtract	<code>sub \$s1, \$s2, \$s3</code>	$\$s1 = \$s2 - \$s3$	Drei Operanden; Daten in Regis-tern
Datentransfer	load word	<code>lw \$s1, 100(\$s2)</code>	$\$s1 = \text{Speicher}[\$s2 + 100]$	Daten vom Hauptspeicher in ein Register
	store word	<code>sw \$s1, 100(\$s2)</code>	$\text{Speicher}[\$s2 + 100] = \$s1$	Daten von einem Register in den Hauptspeicher
Logischer Befehl	and	<code>and \$s1, \$s2, \$s3</code>	$\$s1 = \$s2 \& \$s3$	Drei Registeroperanden; bitweise UND-Verknüpfung
	or	<code>or \$s1, \$s2, \$s3</code>	$\$s1 = \$s2   \$s3$	Drei Registeroperanden; bitweise ODER-Verknüpfung
	nor	<code>nor \$s1, \$s2, \$s3</code>	$\$s1 = \sim (\$s2   \$s3)$	Drei Registeroperanden; bitweise NOR-Verknüpfung
	and immediate	<code>andi \$s1, \$s2, 100</code>	$\$s1 = \$s2 \& 100$	Bitweise UND-Verknüpfung von Registeroperanden mit Konstante
	or immediate	<code>ori \$s1, \$s2, 100</code>	$\$s1 = \$s2   100$	Bitweise ODER-Verknüpfung von Registeroperanden mit Konstante
	shift left logical	<code>sll \$s1, \$s2, 10</code>	$\$s1 = \$s2 << 10$	Linksschieben um Konstante
	shift right logical	<code>srl \$s1, \$s2, 10</code>	$\$s1 = \$s2 >> 10$	Rechtsschieben um Konstante
Verzweigung	branch on equal	<code>beq \$s1, \$s2, L</code>	wenn ( $\$s1 == \$s2$ ), verzweige zu L	Überprüfen auf Gleichheit und Verzweigung
	branch on not equal	<code>bne \$s1, \$s2, L</code>	wenn ( $\$s1 != \$s2$ ), verzweige zu L	Überprüfung auf Ungleichheit und Verzweigung
	set on less than	<code>slt \$s1, \$s2, \$s3</code>	wenn ( $\$s2 < \$s3$ ), setze $\$s1 = 1$ , ansonsten $\$s1 = 0$	Vergleich: kleiner als; verwendet mit <code>beq, bne</code>
	set on less than immediate	<code>slti \$s1, \$s2, 100</code>	wenn ( $\$s2 < 100$ ), setze $\$s1 = 1$ , ansonsten $\$s1 = 0$	Vergleich kleiner als mit Konstan- te verwendet mit <code>beq, bne</code>
Unbedingter Sprung	jump	<code>j L</code>	springe zu L	Sprung zu Zieladresse

**Tab. 2.9 Die bis Abschnitt 2.6 bearbeiteten MIPS-Befehle.** In Abschnitt 2.6 eingeführte MIPS-Strukturen sind farblich hervorgehoben. Das für Sprungbefehle verwendete J-Format wird in Abschnitt 2.9 erläutert. In Abschnitt 2.9 wird außerdem auf die entsprechenden Werte in Adressfeldern von Sprungbefehlen eingegangen.

MIPS-Maschinensprache								
Name	Format	Beispiel					Anmerkungen	
add	R	0	18	19	17	0	32	add \$s1, \$s2, \$s3
sub	R	0	18	19	17	0	34	sub \$s1, \$s2, \$s3
lw	I	35	18	17	100			lw \$s1, 100(\$s2)
sw	I	43	18	17	100			sw \$s1, 100(\$s2)
and	R	0	18	19	17	0	36	and \$s1, \$s2, \$s3
or	R	0	18	19	17	0	37	or \$s1, \$s2, \$s3
nor	R	0	18	19	17	0	39	nor \$s1, \$s2, \$s3
andi	I	12	18	17	100			andi \$s1, \$s2, 100
ori	I	13	18	17	100			ori \$s1, \$s2, 100
sll	R	0	0	18	17	10	0	sll \$s1, \$s2, 10
srl	R	0	0	18	17	10	2	srl \$s1, \$s2, 10
beq	I	4	17	18	25			beq \$s1, \$s2, 100
bne	I	5	17	18	25			bne \$s1, \$s2, 100
slt	R	0	18	19	17	0	42	slt \$s1, \$s2, \$s3
j	J	2	2500					j 10000 (siehe Abschnitt 2.9)
Feldgröße		6 Bit	5 Bit	5 Bit	5 Bit	5 Bit	6 Bit	Alle MIPS-Befehle 32 Bit
R-Format	R	op	rs	rt	rd	shamt	funct	Format für arithmetische Befehle
I-Format	I	op	rs	rt	address			Datentransfer-Format, Sprungformat

## 2.7

# Unterstützung von Prozeduren durch die Rechnerhardware

**Prozedur (procedure)** Eine gespeicherte Subroutine, die eine bestimmte Aufgabe auf den ihr übergebenen Parametern ausführt.

Eine **Prozedur (procedure)** oder Funktion stellt für C- oder Java-Programmierer ein Hilfsmittel zur Strukturierung ihrer Programme dar, wodurch diese einfacher zu verstehen sind. Prozeduren unterstützen die Wiederverwendung von Code und erlauben es dem Programmierer, zu einem Zeitpunkt sich jeweils auf nur einen Teil der Aufgabe zu konzentrieren. Parameter dienen als Bindeglieder zwischen der Prozedur und dem übrigen Programm, mit denen Werte (an die aufrufende Prozedur) übergeben und Ergebnisse (an das aufrufende Programm) zurückgegeben werden können. Am Ende dieses Abschnitts werden die entsprechenden Methoden für Java beschrieben, wobei für Java vom Rechner all das bereitgestellt werden muss, was auch C benötigt.

Sie können sich eine Prozedur wie einen Spion vorstellen, der mit einem geheimen Plan loszieht, Ressourcen bereit gestellt bekommt, die Aufgabe ausführt, seine Spuren verwischt und dann mit dem gewünschten Ergebnis an den Ausgangspunkt zurückkehrt. Nach Abschluss des Auftrags soll nichts mehr darauf hinweisen. Außerdem weiß ein Spion nur das, was er unbedingt wissen muss, so dass er keinerlei Rückschlüsse auf seinen Auftraggeber ziehen kann.

In ähnlicher Weise muss das Programm beim Ausführen einer Prozedur die folgenden sechs Schritte beachten:

1. Die Parameter sind an einer Stelle abzulegen, worauf die Prozedur zugreifen kann.
2. Die Programmsteuerung ist an die Prozedur zu übergeben.
3. Die für die Prozedur benötigten Speicherressourcen müssen bereitgestellt werden.
4. Die Prozedur führt die gewünschte Aufgabe aus.
5. Das Ergebnis ist an einer Stelle abzulegen, auf die das aufrufende Programm zugreifen kann.
6. Die Ablaufsteuerung muss an die Stelle zurückkehren, an der die Prozedur aufgerufen wurde, da eine Prozedur an unterschiedlichen Punkten in einem Programm aufgerufen werden kann.

Wie bereits erwähnt, bieten Register in einem Rechner die schnellste Möglichkeit zum Zugriff auf Daten und sollten daher so oft wie möglich verwendet werden. Die MIPS-Software befolgt beim Reservieren der 32 Register für Prozeduraufrufe die folgende Konvention:

- \$a0-\$a3: vier Argumentregister für die Übergabe der Parameter
- \$v0-\$v1: zwei Register für Rückgabewerte
- \$ra: ein Register für die Rücksprungadresse, um zum Ausgangspunkt zurückzukehren

Zusätzlich zur Bindung dieser Register enthält die MIPS-Assemblersprache auch einen speziellen Befehl für die Prozeduraufrufe: Dieser Befehl springt zu einer Adresse und speichert dabei die Adresse des nachfolgenden Befehls im Register \$ra. Der **Jump-and-Link-Befehl (jump-and-link instruction)** (auch Unterprogrammaufruf genannt) wird wie folgt geschrieben:

```
jal ProcedureAddress
```

Der *Link*-Teil des Namens bedeutet, dass eine Adresse festgehalten wird bzw. ein Verweis auf die Stelle des Aufrufs gebildet wird, so dass die Prozedur an die richtige Adresse zurückkehren kann. Dieser in Register \$ra gespeicherte Verweis (bzw. „Link“) wird als **Rücksprungadresse (return address)** bezeichnet. Die Rücksprungadresse wird benötigt, da eine Prozedur von verschiedenen Stellen des Programms aufgerufen werden kann.

Nach dem Von-Neumann-Prinzip ist ein Register für die Adresse des gerade auszuführenden Befehls notwendig. Aus historischen Gründen wird dieses Register in der MIPS-Architektur als **Befehlszähler (program counter)** oder auch **Befehlszeiger (program counter)** (abgekürzt: PC) bezeichnet, obwohl **Befehlsadressregister** eine treffendere Bezeichnung wäre. Der jal-Befehl sichert den Befehlszählerwert + 4 in Register \$ra, das damit auf den nachfolgenden Befehl zeigt, zu dem der Rücksprung aus der Prozedur erfolgen soll.

Prozessoren wie MIPS unterstützen einen Rücksprung aus einer Prozedur oder ähnliche Fälle mit einem **Jump-register-Befehl (jr)**. Dieser Befehl führt einen unbedingten Sprung zu der in einem Register angegebenen Adresse aus:

```
jr $ra
```

Insbesondere springt dieser **Jump-register-Befehl** zu der in Register \$ra gespeicherten Adresse. Die **aufrufende Prozedur (caller)** speichert die Parameterwerte in \$a0-\$a3 und springt mithilfe des Befehls jal X zur Prozedur X (auch als **aufgerufene Prozedur (callee)** bezeichnet).

Die aufgerufene Prozedur führt die Berechnungen durch, speichert das Ergebnis in den Registern \$v0-\$v1 und übergibt anschließend mithilfe des Befehls jr \$ra die Steuerung wieder an die aufrufende Prozedur.

**Jump-and-Link-Befehl (jump-and-link instruction)** Ein Befehl, der zu einer Adresse springt und dabei die Adresse des nachfolgenden Befehls in einem Register (\$ra bei MIPS) speichert.

**Rücksprungadresse (return address)** Ein Verweis auf die Stelle des Prozederaufrufs, mit dessen Hilfe eine Prozedur nach ihrer Beendigung wieder zur richtigen Adresse zurückkehren kann. Bei der MIPS-Architektur wird sie im Register \$ra gespeichert.

**Befehlszähler (program counter)** Das Register, das die Adresse des Befehls im Programm enthält, der gerade ausgeführt wird.

**Aufrufende Prozedur (caller)** Das Programm, das eine Prozedur auuft und die erforderlichen Parameter bereitstellt.

**Aufgerufene Prozedur (callee)** Eine Prozedur, die eine Reihe gespeicherter Befehle auf den Parametern ausführt, die von der aufrufenden Prozedur bereitgestellt werden und die anschließend die Steuerung wieder an die aufrufende Prozedur übergibt.

## Verwendung weiterer Register

Nehmen wir an, ein Compiler benötige für eine Prozedur mehr als die für die Argumente und die Rückgabewerte vorgesehenen Register. Da wir unsere Spuren nach Erledigung des Auftrags verwischen müssen, muss jedes Register, das die aufrufende Prozedur benötigt, wieder mit den Werten belegt werden, die vor dem Aufruf einer Prozedur in den Registern enthalten waren. Dies ist ein Beispiel für eine Situation, in der Register in den Hauptspeicher ausgelagert werden müssen (siehe Abschnitt „Hardware-Software-Schnittstelle“ auf Seite 46).

**Keller (stack)** Eine als LIFO-Warteschlange (*Last In First Out*) organisierte Datenstruktur zum Auslagern von Registern.

**Kellerzeiger (stack pointer)** Ein Wert, der die in einem Keller zuletzt reservierte Adresse angibt und anzeigt, von welcher Position an auszulagernde Register gespeichert werden müssen oder wo alte Registerwerte gefunden werden können.

Die ideale Datenstruktur zum Auslagern von Registern ist ein **Keller (stack)**, der als LIFO-Warteschlange (*Last In First Out*) organisiert ist. Ein Keller benötigt einen Zeiger auf die zuletzt reservierte Adresse im Keller, um anzusehen, von welcher Position an die nächste Prozedur auszulagernde Register speichern soll beziehungsweise wo alte Registerwerte gefunden werden können. Der **Kellerzeiger (stack pointer)** wird jeweils um ein Wort für jedes gerettete oder wiederhergestellte Register verändert. Keller werden so häufig verwendet, dass es einen eigenen Ausdruck für die Übertragung von Daten auf den und von dem Keller gibt: Das Ablegen von Daten auf den Keller wird als *Push*-Operation bezeichnet und das Entfernen von Daten vom Keller als *Pop*-Operation.

Die MIPS-Software reserviert ein weiteres Register für den Keller: den Kellerzeiger (\$sp), der zum Retten der von der aufgerufenen Prozedur benötigten Register verwendet wird. Aus historischen Gründen „wächst“ der Keller von höherwertigen Adressen hin zu niederwertigen Adressen. Diese Konvention bedeutet, dass nach dem Erniedrigen des Kellerzeigers Werte auf dem Keller abgelegt werden können. Die Erhöhung des Kellerzeigers verkleinert den Keller, wodurch Werte aus dem Keller entfernt werden.

## BEISPIEL

### Übersetzung einer C-Prozedur, die keine andere Prozedur aufruft

Das Beispiel auf Seite 40 lässt sich als C-Prozedur folgendermaßen darstellen:

```
int leaf_example (int g, int h, int i, int j)
{
    int f;

    f = (g + h) - (i + j);
    return f;
}
```

Wie lautet der übersetzte MIPS-Assemblercode?

Die Parametervariablen g, h, i und j sind den Argumentregistern \$a0, \$a1, \$a2 und \$a3 zugeordnet, und f entspricht Register \$s0. Das kompilierte Programm beginnt mit der Marke der Prozedur:

```
leaf_example:
```

Der nächste Schritt besteht darin, die von der Prozedur verwendeten Register zu retten. Die C-Zuweisung im Prozedurkörper ist mit dem Beispiel auf Seite 40 identisch, bei dem zwei temporäre Register verwendet werden. Somit müssen drei Register gespeichert werden: \$s0, \$t0 und \$t1. Wir legen die alten Werte auf dem Keller ab, indem wir Platz für drei Wörter im Keller schaffen und diese dann speichern:

```
addi $sp,$sp,-12      # schaffe auf dem Keller Platz
# für 3 Register
```

## ANTWORT

```

sw $t1, 8($sp)      # speichere Reg. $t1
                      # für die spätere Verwendung
sw $t0, 4($sp)      # speichere Reg. $t0
                      # für die spätere Verwendung
sw $s0, 0($sp)      # speichere Reg. $s0
                      # für die spätere Verwendung

```

In Abbildung 2.5 ist der Keller vor, während und nach dem Prozedurauftruf dargestellt. Die nächsten drei Anweisungen entsprechen dem Prozedurkörper nach dem Beispiel auf Seite 40:

```

add $t0,$a0,$a1 # Reg. $t0 enthält g + h
add $t1,$a2,$a3 # Reg. $t1 enthält i + j
sub $s0,$t0,$t1 # f = $t0 - $t1, d.h.
                  # gleich (g + h) - (i + j)

```

Um den Wert von f zurückzugeben, kopieren wir ihn in ein Register für Rückgabewerte:

```
add $v0,$s0,$zero # Rückgabe von f ($v0 = $s0 + 0)
```

Vor dem Rücksprung stellen wir die drei alten Werte der geretteten Register wieder her, indem sie vom Keller gelesen und in die Register geladen werden. Der Kellerzeiger wird auf den Wert von vor dem Prozedurauftruf gesetzt:

```

lw $s0, 0($sp) # stelle Reg. $s0 für aufrufende
                  # Prozedur wieder her
lw $t0, 4($sp) # stelle Reg. $t0 für aufrufende
                  # Prozedur wieder her
lw $t1, 8($sp) # stelle Reg. $t1 für aufrufende
                  # Prozedur wieder her
addi $sp,$sp,12 # entferne die Werte vom Stapel

```

Die Prozedur endet mit einem *Jump-register*-Befehl mit der Rücksprungadresse im Register:

```
jr $ra # springe zurück zur aufrufenden Prozedur
```

Im obigen Beispiel wurden temporäre Register verwendet und angenommen, dass deren alte Werte gerettet und wiederhergestellt werden müssen. Damit ein Register, dessen Wert im weiteren Programmverlauf nicht mehr verwendet wird, was bei einem temporären Register durchaus vorkommen kann, nicht gerettet und wiederhergestellt werden muss, teilt die MIPS-Software 18 der Register in zwei Gruppen:

- \$t0-\$t9: 10 temporäre Register, die von der aufgerufenen Prozedur bei einem Prozedurauftruf *nicht* gerettet werden müssen.
- \$s0-\$s7: 8 zu sichernde Register (saved registers), die bei einem Prozedurauftruf gerettet werden müssen (die aufgerufene Prozedur rettet nur die von ihr verwendeten Register und stellt diese wieder her).

Durch diese einfache Konvention reduziert sich der Aufwand für das Auslagern der Register, da nicht unbedingt alle gerettet werden müssen. Im obigen Beispiel geht die aufrufende Prozedur nicht davon aus, dass die Register \$t0 und \$t1 über den Prozedurauftruf hinweg beibehalten werden, weshalb zwei Speicher- und zwei Ladebefehle

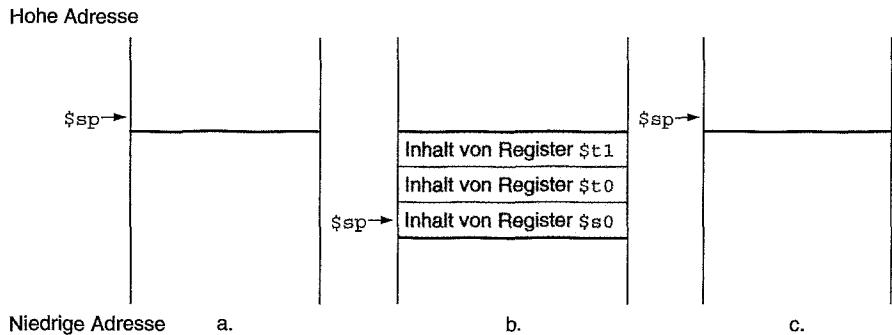


Abb. 2.5 Die Werte des Kellerzeigers und des Kellers (a) vor, (b) während und (c) nach dem Prozederaufruf. Der Kellerzeiger zeigt immer auf das „oberste“ Element des Kellers bzw. in dieser Abbildung auf das letzte Wort im Keller.

im Code weggelassen werden können. Das Register \$s0 muss jedoch gerettet und wiederhergestellt werden, da die aufgerufene Prozedur annehmen muss, dass die aufrufende Prozedur den darin enthaltenen Wert weiter benötigt.

## Geschachtelte Prozeduren

Prozeduren, die keine anderen Prozeduren aufrufen, werden als *Blattprozeduren* bezeichnet. Das Leben wäre einfach, wenn alle Prozeduren Blattprozeduren wären. Das ist jedoch nicht der Fall. So, wie ein Spion im Rahmen eines Auftrags andere Spione engagiert, die ihrerseits wieder andere Spione einsetzen können, so rufen Prozeduren andere Prozeduren auf. Zudem rufen rekursive Prozeduren „Klone“ von sich selbst auf. Wir müssen schon Acht geben, wenn wir in Prozeduren Register verwenden. Noch mehr Sorgfalt müssen wir jedoch walten lassen, wenn Prozeduren aufgerufen werden, die keine Blattprozeduren sind.

Nehmen wir beispielsweise an, das Hauptprogramm ruft Prozedur A mit dem Argument 3 auf, indem es den Wert 3 im Register \$a0 ablegt und danach den Befehl ja1 A ausführt. Nehmen wir weiter an, dass Prozedur A mit dem Befehl ja1 B Prozedur B mit dem Argument 7 aufruft, das ebenfalls in Register \$a0 übergeben wird. Da A die Aufgabe noch nicht erledigt hat, kommt es hinsichtlich der Verwendung von Register \$a0 zu einem Konflikt. Entsprechend kommt es hinsichtlich der Rücksprungadresse in Register \$ra zu einem Konflikt, da sich dort nun die Rücksprungadresse für B befindet. Wenn wir keine Maßnahmen zum Beheben des Problems ergreifen, führt dieser Konflikt dazu, dass Prozedur A nicht mehr zum aufrufenden Programm zurückspringen kann.

Eine Lösungsmöglichkeit besteht darin, alle weiteren Register, die beibehalten werden müssen mit den zu sichernden Registern auf dem Keller abzulegen. Die aufrufende Prozedur rettet alle Argumentregister (\$a0-\$a3) oder temporäre Register (\$t0-\$t9), die nach dem Prozederaufruf benötigt werden, auf dem Keller. Ebenso speichert die aufgerufene Prozedur das Rücksprungadressregister \$ra sowie alle zu sichern den Register, die von der aufgerufenen Prozedur verwendet werden (\$s0-\$s7), auf dem Keller. Der Kellerzeiger \$sp wird entsprechend der Anzahl der auf dem Keller abgelegten Register eingestellt. Beim Rücksprung werden die Register aus dem Hauptspeicher wiederhergestellt und der Kellerzeiger wird zurückgesetzt.

## Übersetzung einer rekursiven C-Prozedur und Darstellung der Verknüpfung geschachtelter Prozeduren

## BEISPIEL

Das Beispiel zeigt eine rekursive Prozedur zur Berechnung der Fakultät:

```
int fact (int n)
{
    if (n < 1) return (1);
    else return (n * fact(n-1));
}
```

Wie lautet der MIPS-Assemblercode?

Die Parametervariable `n` entspricht dem Argumentregister `$a0`. Das übersetzte Programm beginnt mit der Marke der Prozedur und rettet zwei Register auf den Keller: die Rücksprungadresse und das Register `$a0`:

```
fact:
    addi $sp,$sp,-8    # schaffe auf dem Keller Platz
                        # für 2 Registerwerte
    sw $ra, 4($sp)    # sichere die Rücksprungadresse
    sw $a0, 0($sp)    # sichere das Argument n
```

Wenn `fact` zum ersten Mal aufgerufen wird, rettet `sw` eine Adresse in dem Programm, das `fact` aufgerufen hat. Die nächsten beiden Befehle überprüfen, ob `n` kleiner als 1 ist, und springen zu L1, wenn  $n \geq 1$ .

```
    slti $t0,$a0,1    # prüfe, ob n < 1
    beq $t0,$zero,L1 # wenn n >= 1, verzweige nach L1
```

Wenn `n` kleiner als 1 ist, gibt `fact` den Wert 1 zurück, indem 1 in einem Register für Rückgabewerte gespeichert wird. In unserem Beispiel wird die 1 zur 0 addiert und das Ergebnis in Register `$v0` gespeichert. Anschließend werden die beiden gespeicherten Werte durch das Versetzen des Kellerzeigers aus dem Keller entfernt, und die Prozedur springt an die Rücksprungadresse:

```
    addi $v0,$zero,1    # gebe 1 zurück
    addi $sp,$sp,8      # entferne zwei Werte vom Keller
    jr   $ra             # springe zurück zur aufrufenden
                        # Prozedur
```

Vor dem Versetzen des Kellerzeigers und damit vor dem Entfernen der zwei Elemente aus dem Keller, müssten diese wieder in die Register `$a0` und `$ra` geladen werden. Da `$a0` und `$ra` nicht verändert werden, wenn `n` kleiner als 1 ist, können wir diese Befehle weglassen.

Wenn `n` nicht kleiner als 1 ist, wird das Argument `n` dekrementiert. Anschließend wird `fact` noch einmal mit dem dekrementierten Wert aufgerufen:

```
L1: addi $a0,$a0,-1 # n >= 1: dekrementiere n
    jal fact          # rufe fact mit (n - 1) auf
```

Der nächste Befehl folgt auf den Rücksprung aus `fact`. Nun werden die alte Rücksprungadresse und das alte Argument zusammen mit dem Kellerzeiger wiederhergestellt:

```
lw   $a0, 0($sp) # zurück von fact: stelle n wieder her
lw   $ra, 4($sp) # stelle Rücksprungadresse wieder her
addi $sp,$sp,8   # aktualisiere den Kellerzeiger
```

## ANTWORT

Als Nächstes wird im Rückgabewertregister \$v0 das Produkt aus dem alten Argument in \$a0 und dem aktuellen Wert im Rückgabewertregister gespeichert. Wir nehmen an, es gibt einen Multiplikationsbefehl, auch wenn wir diesen erst in Kapitel 3 kennen lernen werden:

```
mul $v0,$a0,$v0 # gib n * fact (n - 1) zurück
```

Und abschließend springt `fact` wieder an die Rücksprungadresse:

```
jr $ra # kehre zum Aufrufer zurück
```

## Hardware-Software-Schnittstelle

**Globales Zeigerregister (*global pointer*)** Reserviertes Register, das auf statische Daten zeigt.

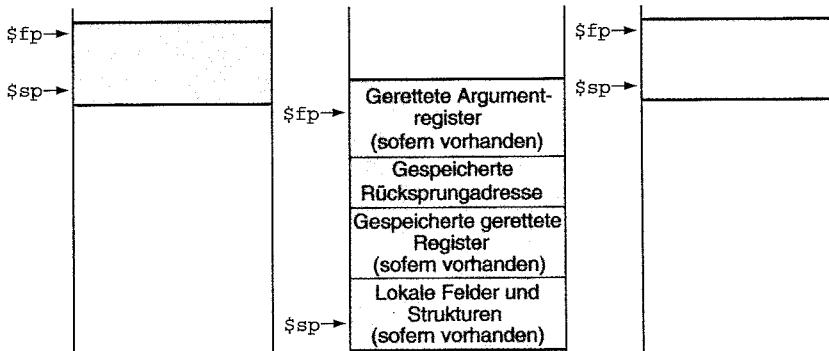
Eine C-Variable belegt eine Stelle im Speicher, wobei die Interpretation sowohl vom Typ als auch von der *Speicherklasse* abhängt. Typen werden in Kapitel 3 ausführlich beschrieben. Beispiele hierfür sind jedoch `int` und `char`. Bei C gibt es zwei Speicherklassen: *automatic* und *static*. Variablen der Speicherklasse *automatic* sind für eine Prozedur lokale Variablen, die nach Beenden der Prozedur nicht weiter verwendet werden. Variablen der Speicherklasse *static* (statische Variablen) existieren über das Ende und den Anfang von Prozeduren hinweg. Außerhalb der Prozeduren deklarierte C-Variablen sind ebenso statische Variablen wie alle Variablen, die mit dem Schlüsselwort `static` deklariert werden. Alle anderen Variablen gehören zur Speicherklasse *automatic*. Um den Zugriff auf statische Daten zu vereinfachen, reserviert die MIPS-Software ein weiteres Register, das so genannte **globale Zeigerregister (*global pointer*)** oder \$gp.

In Tabelle 2.10 ist zusammenfassend dargestellt, was über einen Prozeduraufruf hinweg gesichert wird. Beachten Sie, dass mit mehreren Methoden dafür gesorgt wird, den Keller zu sichern. Die im Keller gespeicherten Werte werden dadurch geschützt, dass nur auf den darüberliegenden Bereich zugegriffen wird, d.h. auf einen Bereich mit kleineren Adressen als der beim Aufruf aktuelle \$sp. Der Kellerzeiger bleibt erhalten, indem derselbe Wert hinzugefügt wird, der beim Eintritt in die Prozedur abgezogen wurde. Die weiteren Register bleiben erhalten, indem sie (wenn sie genutzt werden) auf den Keller gerettet und von dort aus wieder hergestellt werden. Diese Aktionen garantieren, dass die aufrufende Prozedur beim Laden ihrer Register vom Keller auch wieder dieselben Daten zurückbekommt, die zuvor von ihr auf den Keller gerettet wurden, da die aufgerufene Prozedur zusichert, zum einen den zum Zeitpunkt des Aufrufs aktuellen \$sp wieder herzustellen und zum anderen den Kellerbereich der aufrufenden Prozedur nicht zu verändern.

Tab. 2.10 Was über einen Prozeduraufruf hinweg beibehalten wird und was nicht. Wenn sich die Software auf das Rahmenzeigerregister oder auf das globale Zeigerregister bezieht, die beide im nachfolgenden Abschnitt beschrieben werden, werden diese ebenfalls beibehalten.

Beibehalten	Nicht beibehalten
Gerettete Register: \$s0-\$s7	Temporäre Register: \$t0-\$t9
Kellerzeigerregister: \$sp	Argumentregister: \$a0-\$a3
Rücksprungadressregister: \$ra	Rückgabewertregister: \$v0-\$v1
Keller über dem Kellerzeiger	Keller unter dem Kellerzeiger

Hohe Adresse



a. b. c.

**Abb. 2.6 Darstellung der Kellerzuordnung (a) vor, (b) während und (c) nach dem Prozederaufruf.** Der Rahmenzeiger (\$fp) zeigt auf das erste Wort im Rahmen, häufig ein gerettetes Argumentregister, und der Kellerzeiger (\$sp) zeigt auf das oberste Element des Kellers. Der Keller wird so angepasst, dass für alle geretteten Register und alle speicherresidenten lokalen Variablen genügend Platz vorhanden ist. Da sich der Kellerzeiger während der Programmausführung ändern kann, ist es für Programmierer einfacher, Variablen mithilfe des festen Rahmenzeigers zu referenzieren, auch wenn dies mithilfe des Kellerzeigers und etwas Adressberechnung durchgeführt werden könnte. Falls für eine Prozedur keine lokalen Variablen auf dem Keller abgelegt werden, kann der Compiler Zeit sparen, wenn er den Rahmenzeiger nicht einstellt und wiederherstellt. Bei Verwendung eines Rahmenzeigers wird dieser beim Prozederaufruf mit der Adresse in \$sp initialisiert und \$sp wird mithilfe von \$fp wiederhergestellt.

## Zuordnen von Speicherplatz für neue Daten im Keller

Schließlich ist noch zu beachten, dass der Keller auch zum Speichern von Variablen verwendet wird, die für die Prozedur lokal sind und nicht in Register passen. Beispiele hierfür sind lokale Felder und Strukturen. Das Segment im Keller, das die geretteten Register und lokalen Variablen einer Prozedur enthält, wird als **Prozederaufrufrahmen** (*procedure call frame*) bezeichnet. In Abbildung 2.6 ist der Zustand des Kellers vor, während und nach dem Prozederaufruf dargestellt.

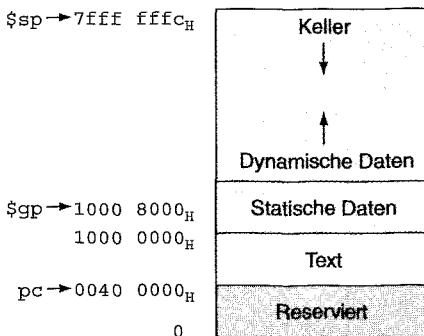
Manche MIPS-Software verwendet einen **Rahmenzeiger** (*frame pointer*) (\$fp), der auf das erste Wort in einem Prozedurrahmen zeigt. Weil sich ein Kellerzeiger im Laufe der Prozedur verändern kann, ist es unter Umständen schwierig den Offset zum Kellerzeiger für eine lokale Variable im Hauptspeicher zu bestimmen. Im Gegensatz dazu stellt ein Rahmenzeiger ein festes Basisregister für lokale Speicherreferenzen innerhalb einer Prozedur dar. Ein Prozederaufrufrahmen erscheint im Keller unabhängig davon, ob ein Rahmenzeiger verwendet wird oder nicht. Wir haben \$fp nicht verwendet, da wir \$sp in keiner Prozedur ändern: In unseren Beispielen wird der Keller nur am Anfang und am Ende der Prozedur geändert.

**Prozederaufrufrahmen**  
(*procedure call frame*) Das Segment im Keller, das die geretteten Register und lokalen Variablen einer Prozedur enthält.

**Rahmenzeiger** (*frame pointer*) Ein Wert, der die Position der geretteten Register und lokalen Variablen einer Prozedur anzeigt.

## Zuordnen von Speicherplatz für neue Daten auf der Halde

Neben den für Prozeduren lokalen Variablen vom Typ `automatic` benötigen C-Programmierer Speicherplatz für statische Variablen und für dynamische Datenstrukturen. In Abbildung 2.7 ist die MIPS-Konvention für die Speicherbelegung dargestellt. Der Keller beginnt am oberen Speicherende und wächst nach unten. Der erste Teil



**Abb. 2.7 Die MIPS-Speicheraufteilung für Programme und Daten.** Diese Adressen sind gemäß einer Softwarekonvention festgelegt worden und sind nicht Teil der MIPS-Architektur. Im oberen Speicherbereich wird der Kellerzeiger mit 7fff ffffc<sub>H</sub> initialisiert und wächst nach unten in Richtung Datensegment. Am unteren Ende beginnt der Programmcode („Text“) bei 0040 0000<sub>H</sub>. Die statischen Daten beginnen bei 1000 0000<sub>H</sub>. Der Bereich für die dynamischen Daten, der als Halde bezeichnet wird und in C mit malloc und in Java mit new reserviert wird, folgt als Nächstes und wächst nach oben in Richtung Keller. Der globale Zeiger \$gp wird auf eine Adresse gesetzt, mit der ein einfacher Zugriff auf die Daten ermöglicht wird. Er wird mit 1000 8000<sub>H</sub> initialisiert, so dass mit positiven und negativen 16-Bit-Offsets zum \$gp auf den Bereich zwischen 1000 0000<sub>H</sub> und 1000 ffff<sub>H</sub> zugegriffen werden kann (siehe die Zweierkomplementadressierung in Kapitel 3).

**Textsegment** Das Segment einer Unix-Objektdatei, der den Maschinencode für Routinen in der Quelldatei enthält.

am unteren Speicherende ist reserviert. Diesem Teil folgt der Bereich mit dem MIPS-Maschinencode, der als **Textsegment** bezeichnet wird. Über dem Code befindet sich das *statische Datensegment*, in dem Konstanten und andere statische Variablen abgelegt werden. Felder haben eine feste Länge und werden im statischen Datensegment abgelegt. Datenstrukturen, wie beispielsweise verkettete Listen, verändern dagegen ihre Länge im Laufe ihrer Lebensdauer. Das für Datenstrukturen dieser Art reservierte Segment wird als *Halde (heap)* bezeichnet, und kommt im Speicher nach dem statischen Datensegment. Aufgrund dieser Zuordnung wachsen der Keller und die Halde einander entgegen, wodurch der Speicher effizient genutzt werden kann, da die beiden Segmente zu- und abnehmen.

In C wird der Speicherplatz auf der Halde mit speziellen Funktionen reserviert und freigegeben. Mit `malloc()` wird Speicherplatz auf der Halde reserviert und ein Zeiger auf den Speicherplatz zurückgegeben. Mit `free()` wird Speicherplatz auf der Halde, auf den der Zeiger zeigt, freigegeben. Die Speicherbelegung wird in C von den Programmen verwaltet, was die Ursache für viele allgemeine und schwerwiegende Fehler ist. Wenn vergessen wird, Speicherplatz freizugeben, führt dies zu einem *Speicherleck (memory leak)*. Irgendwann ist so viel Speicher belegt, dass das Betriebssystem abstürzt. Wenn Speicherplatz zu früh freigegeben wird, führt das zu *hängenden Zeigern (dangling pointers)* mit Verweisen, die vom Programm so nie beabsichtigt waren.

Tabelle 2.11 fasst die Konventionen für die Registerbelegungen für die MIPS-Assemblersprache zusammen. In den Tabellen 2.12 und 2.13 sind die bisher beschriebenen Teile der MIPS-Assemblerbefehle sowie die entsprechenden MIPS-Maschinenbefehle aufgeführt.



**Vertiefung:** Was, wenn mehr als vier Parameter zu übergeben sind? Gemäß der MIPS-Konvention werden zusätzliche Parameter im Keller direkt über dem Rahmenzeiger abgelegt. Die Prozedur erwartet, dass sich die ersten vier Parameter in den Registern \$a0 bis \$a3 und alle anderen Parameter im Hauptspeicher befinden und über den Rahmenzeiger adressierbar sind.

Wie in der Bildunterschrift von Abbildung 2.6 bereits erwähnt, ist der Rahmenzeiger deshalb so praktisch, weil sich die Offsets der Referenzen auf die Variablen im Kellerspeicher während der Prozedur nicht verändern. Ein Rahmenzeiger ist jedoch nicht unbedingt notwendig. Der GNU MIPS C-Compiler verwendet einen Rahmenzeiger, der C-Compiler von MIPS/Silicon Graphics jedoch nicht. Dieser nutzt Register 30 als ein weiteres gerettetes Register (\$s8).

`jal` speichert die Adresse des dem `jal` folgenden Befehls in Register `$ra`, so dass ein Prozedurrücksprung einfach mit `jr $ra` erfolgt.

Tab. 2.11 MIPS-Registerkonventionen. Register 1, `$at`, ist für den Assembler reserviert (siehe Abschnitt 2.10), und die Register 26–27, `$k0–$k1`, sind für das Betriebssystem reserviert.

#### MIPS-Assemblersprache

Name	Registernummer	Nutzung	Bei Aufruf beibehalten?
<code>\$zero</code>	0	der konstante Wert 0	–
<code>\$v0–\$v1</code>	2–3	Werte für Ergebnisse und für die Auswertung von Ausdrücken	nein
<code>\$a0–\$a3</code>	4–7	Argumente	nein
<code>\$t0–\$t7</code>	8–15	temporäre Variablen	nein
<code>\$s0–\$s7</code>	16–23	gespeicherte Variablen	ja
<code>\$t8–\$t9</code>	24–25	weitere temporäre Variablen	nein
<code>\$gp</code>	28	globaler Zeiger	ja
<code>\$sp</code>	29	Kellerzeiger	ja
<code>\$fp</code>	30	Rahmenzeiger	ja
<code>\$ra</code>	31	Rücksprungadresse	ja

Welche der folgenden Aussagen zu C und Java sind allgemein richtig?

1. Prozeduraufrufe in C sind schneller als Methodenaufrufe in Java.
2. C-Programmierer verwalten ihre Daten explizit, während dies bei Java automatisch erfolgt.
3. C führt zu mehr Zeigerfehlern und Speicherlecks als Java.
4. C übergibt Parameter in Registern, Java im Keller.



## 2.8

## Kommunikation mit Menschen

Computer wurden ursprünglich als schnelle Rechenmaschinen erfunden. Mit der kommerziellen Verbreitung wurden sie auch für die Verarbeitung von Text eingesetzt. Die meisten modernen Computer verwenden zur Darstellung von Zeichen 8-Bit-Bytes, wobei der ASCII-Code (American Standard Code for Information Interchange) der allgemein anerkannte Standard für die Zeichendarstellung ist. In Tabelle 2.14 ist eine Übersicht über den ASCII-Code dargestellt.

Mit einer Reihe von Befehlen kann ein Byte aus einem Wort extrahiert werden, womit *Load-word*- und *Store-word*-Befehle ausreichen, um sowohl Bytes als auch Wörter

!(&|=>  
(wow open tab at bar is great)  
Vierte Zeile des  
Tastaturgedichts „Hatless  
Atlas“, 1991 (dabei werden den  
ASCII-Zeichen Namen  
zugewiesen: „!“ steht für  
„wow“ „.“ steht für „open“ „|“  
steht für „bar“ usw.)

Tab. 2.12 Bis Abschnitt 2.7 bearbeitete MIPS-Architektur. Die in Abschnitt 2.7 neu eingeführten MIPS-Strukturen sind farblich hervorgehoben. Das für Sprung- und Jump-and-Link-Befehle verwendete J-Format wird in Abschnitt 2.9 erläutert.

MIPS-Operanden		
Name	Beispiel	Anmerkungen
32 Register	<code>\$s0-\$s7, \$t0-\$t9, \$zero, \$a0-\$a3, \$v0-\$v1, \$gp, \$fp, \$sp, \$ra</code>	Speicherort für schnellen Zugriff auf Daten. Bei MIPS müssen die Daten für die arithmetischen Operationen in Registern stehen. Das MIPS-Register <code>\$zero</code> ist immer 0. <code>\$gp</code> (28) ist das globale Zeigerregister, <code>\$sp</code> (29) ist der Kellerzeiger, <code>\$fp</code> (30) ist der Rahmenzeiger und <code>\$ra</code> (31) ist die Rücksprungadresse.
$2^{30}$ Speicherwörter	<code>Speicher[0], Speicher[4], ..., Speicher[4294967292]</code>	Zugriff nur durch Datentransport-Befehle. Die MIPS-Architektur verwendet Byteadressen. Daher unterscheiden sich aufeinander folgende Wortadressen um 4. Im Hauptspeicher werden Datenstrukturen, Felder und ausgelagerte Register gespeichert, wie die, die bei einem Prozederaufruf gespeichert werden.

#### MIPS-Assemblersprache

Kategorie	Befehl	Beispiel	Bedeutung	Anmerkungen
Arithmetischer Befehl	add	<code>add \$s1, \$s2, \$s3</code>	$\$s1 = \$s2 + \$s3$	Drei Registeroperanden
	subtract	<code>sub \$s1, \$s2, \$s3</code>	$\$s1 = \$s2 - \$s3$	Drei Registeroperanden
Daten-transfer	load word	<code>lw \$s1,100(\$s2)</code>	$\$s1 = \text{Speicher}[\$s2 + 100]$	Daten vom Hauptspeicher in ein Register
	store word	<code>sw \$s1,100(\$s2)</code>	$\text{Speicher}[\$s2 + 100] = \$s1$	Daten von einem Register in den Hauptspeicher
Logischer Befehl	and	<code>and \$s1, \$s2, \$s3</code>	$\$s1 = \$s2 \& \$s3$	Drei Registeroperanden; bitweise UND-Verknüpfung
	or	<code>or \$s1, \$s2, \$s3</code>	$\$s1 = \$s2   \$s3$	Drei Registeroperanden; bitweise ODER-Verknüpfung
	nor	<code>nor \$s1, \$s2, \$s3</code>	$\$s1 = \sim (\$s2   \$s3)$	Drei Registeroperanden; bitweise NOR-Verknüpfung
	and immediate	<code>andi \$s1, \$s2, 100</code>	$\$s1 = \$s2 \& 100$	Bitweise UND-Verknüpfung von Registeroperanden mit Konstante
	or immediate	<code>ori \$s1, \$s2, 100</code>	$\$s1 = \$s2   100$	Bitweise ODER-Verknüpfung von Registeroperanden mit Konstante
	shift left logical	<code>sll \$s1, \$s2, 10</code>	$\$s1 = \$s2 << 10$	Linksverschieben um Konstante
	shift right logical	<code>srl \$s1, \$s2, 10</code>	$\$s1 = \$s2 >> 10$	Rechtsverschieben um Konstante
Verzweigung	branch on equal	<code>beq \$s1, \$s2,L</code>	wenn ( $\$s1 == \$s2$ ), verzweige zu L	Überprüfen auf Gleichheit und Verzweigung
	branch on not equal	<code>bne \$s1, \$s2,L</code>	wenn ( $\$s1 != \$s2$ ), verzweige zu L	Überprüfen auf Ungleichheit und Verzweigung
	set on less than	<code>slt \$s1, \$s2, \$s3</code>	wenn ( $\$s2 < \$s3$ ), setze $\$s1 = 1$ , ansonsten $\$s1 = 0$	Vergleich: kleiner als; verwendet mit <code>beq, bne</code>
	set on less than immediate	<code>slti \$s1, \$s2, 100</code>	wenn ( $\$s2 < 100$ ), setze $\$s1 = 1$ , ansonsten $\$s1 = 0$	Immediate-Version des Vergleichs: kleiner als; verwendet mit <code>beq, bne</code>
Unbedingter Sprung	jump	<code>j L</code>	springe zu L	Sprung zu Zieladresse
	jump register	<code>jr \$ra</code>	springe zu <code>\$ra</code>	Für Prozedurrücksprung
	jump and link	<code>jal L</code>	$\$ra = PC + 4$ , springe zu L	Für Prozederaufruf

**Tab. 2.13 Die bis Abschnitt 2.7 bearbeitete MIPS-Maschinensprache.** In Abschnitt 2.7 eingeführte MIPS-Strukturen sind farblich hervorgehoben. Das für Sprung- und Jump-and-Link-Befehle verwendete J-Format wird in Abschnitt 2.9 erläutert. In diesem Abschnitt wird außerdem erläutert, warum das Speichern des Werts 25 im Adressfeld der Maschinenbefehle `beq` und `bne` dasselbe wie das Speichern des Werts 100 in Assemblersprache ist.

### MIPS-Maschinensprache

Name	Format	Beispiel						Anmerkungen
add	R	0	18	19	17	0	32	add \$s1, \$s2, \$s3
sub	R	0	18	19	17	0	34	sub \$s1, \$s2, \$s3
lw	I	35	18	17	100			lw \$s1, 100(\$s2)
sw	I	43	18	17	100			sw \$s1, 100(\$s2)
and	R	0	18	19	17	0	36	and \$s1, \$s2, \$s3
or	R	0	18	19	17	0	37	or \$s1, \$s2, \$s3
nor	R	0	18	19	17	0	39	nor \$s1, \$s2, \$s3
andi	I	12	18	17	100			andi \$s1, \$s2, 100
ori	I	13	18	17	100			ori \$s1, \$s2, 100
sll	R	0	0	18	17	10	0	sll \$s1, \$s2, 10
srl	R	0	0	18	17	10	2	srl \$s1, \$s2, 10
beq	I	4	17	18	25			beq \$s1, \$s2, 100
bne	I	5	17	18	25			bne \$s1, \$s2, 100
slt	R	0	18	19	17	0	42	slt \$s1, \$s2, \$s3
j	J	2	2500					j 10000 (siehe Abschnitt 2.9)
jr	R	0	31	0	0	0	8	jr \$ra
jal	J	3	2500					jal 10000 (siehe Abschnitt 2.9)
Feldgröße		6 Bit	5 Bit	5 Bit	5 Bit	5 Bit	6 Bit	Alle MIPS-Befehle 32 Bit lang
R-Format	R	op	rs	rt	rd	shamt	funct	Format für arithmetische Befehle
I-Format	I	op	rs	rt	address			Datentransport-Format, Sprungformat

zu übertragen. Wegen der Bedeutung der Verarbeitung von Text in vielen Programmen, stellt MIPS jedoch auch Befehle zum Transport von Bytes bereit. Der *Load-byte*-Befehl (`lb`) lädt ein Byte aus dem Hauptspeicher und legt es in den rechtsbündigen 8 Bit eines Registers ab. Der *Store-byte*-Befehl (`sb`) nimmt ein Byte aus den rechtsbündigen 8 Bit eines Registers und schreibt es in den Hauptspeicher. Somit lässt sich ein Byte mit der folgenden Befehlsfolge kopieren:

```
lb $t0,0($sp) # lese Byte aus Speicher
sb $t0,0($gp) # schreibe Byte in Speicher
```

Zeichen werden normalerweise in Zeichenfolgen zusammengefasst, die eine variable Anzahl Zeichen enthalten. Es gibt drei Möglichkeiten, eine Zeichenfolge darzustellen: (1) Die erste Position der Zeichenfolge ist für die Längenangabe der Zeichenfolge reserviert. (2) Die Länge der Zeichenfolge steht (wie in einer Struktur) in einer begleitenden Variablen. (3) Die letzte Position einer Zeichenfolge wird durch ein Zeichen angezeigt, das das Ende einer Zeichenfolge kennzeichnet. In C ist die dritte Möglichkeit realisiert. Eine Zeichenfolge wird mit einem Byte mit dem Wert 0 (in ASCII als Null bezeichnet) abgeschlossen. Die Zeichenfolge „Cal“ wird somit in C durch die folgenden vier Bytes (in Dezimalschreibweise) dargestellt: 67, 97, 108, 0.

**Tab. 2.14 ASCII-Darstellung von Zeichen.** Groß- und Kleinbuchstaben unterscheiden sich exakt um den Wert 32. Damit lassen sich Groß- und Kleinbuchstaben schneller überprüfen oder ändern. Zu den hier nicht aufgeführten Werten zählen Steuerzeichen. So stellt der Wert 8 beispielsweise die Rücktaste dar, der Wert 9 das Tabulatorzeichen und der Wert 13 das Zeichen für Zeilenumbruch. Ein weiterer nützlicher Wert ist der Wert 0 für Null, mit dem die Programmiersprache C das Ende einer Zeichenfolge kennzeichnet.

ASCII-Wert	Zeichen	ASCII-Wert	Zeichen	ASCII-Wert	Zeichen	ASCII-Wert	Zeichen	ASCII-Wert	Zeichen	ASCII-Wert	Zeichen
32	Leerzeichen	48	0	64	@	80	P	096	'	112	p
33	!	49	1	65	A	81	Q	097	a	113	q
34	"	50	2	66	B	82	R	098	b	114	r
35	#	51	3	67	C	83	S	099	c	115	s
36	\$	52	4	68	D	84	T	100	d	116	t
37	%	53	5	69	E	85	U	101	e	117	u
38	&	54	6	70	F	86	V	102	f	118	v
39	'	55	7	71	G	87	W	103	g	119	w
40	(	56	8	72	H	88	X	104	h	120	x
41	)	57	9	73	I	89	Y	105	i	121	y
42	*	58	:	74	J	90	Z	106	j	122	z
43	+	59	;	75	K	91	[	107	k	123	{
44	,	60	<	76	L	92	\	108	l	124	
45	-	61	=	77	M	93	]	109	m	125	}
46	.	62	>	78	N	94	^	110	n	126	~
47	/	63	?	79	O	95	_	111	o	127	DEL

## BEISPIEL

### Übersetzung einer Prozedur zum Kopieren einer Zeichenfolge, zeigt den Umgang mit C-Zeichenfolgen

Die Prozedur `strcpy` kopiert die Zeichenfolge `y` in die Zeichenfolge `x` und verwendet dabei die Nullterminierung von C:

```
void strcpy (char x[], char y[])
{
    int i;

    i = 0;

    while ((x[i] = y[i]) != '\0') /* kopiere & prüfe Byte */
        i += 1;
}
```

Wie lautet der kompilierte MIPS-Assemblercode?

## ANTWORT

Unten ist das grundlegende Codesegment in der MIPS-Assemblersprache dargestellt. Wir nehmen an, die Basisadressen für die Felder `x` und `y` befinden sich in `$a0` und `$a1`, während sich `i` in `$s0` befindet. `strcpy` stellt den Kellerzeiger ein und speichert das zu sichernde Register `$s0` auf dem Keller:

```
strcpy:
    addi $sp,$sp,-4  # verringere Kellerzeiger
                    # für ein Wort
    sw   $s0, 0($sp) # sichere $s0
```

Um *i* mit 0 zu initialisieren, setzt der nächste Befehl das Register *\$s0* durch die Addition von 0 und 0 und der Speicherung der Summe in *\$s0* auf 0:

```
add $s0,$zero,$zero # i = 0 + 0
```

Das ist der Beginn der Schleife. Die Adresse *y[i]* wird zunächst durch die Addition von *i* und *y[]* gebildet:

```
L1: add $t1,$s0,$a1 # Adresse von y[i] nach $t1
```

In diesem Fall muss *i* nicht mit 4 multipliziert werden, da das Feld *y* aus *Bytes* und nicht wie in vorangegangenen Beispielen aus Wörtern besteht.

Um das Zeichen in *y[i]* zu laden, verwenden wir den *Load-byte*-Befehl, der das Zeichen nach *\$t2* liest:

```
lb $t2, 0($t1) # $t2 = y[i]
```

In ähnlicher Weise wird die Adresse von *x[i]* berechnet und in *\$t3* geladen. Das Zeichen in *\$t2* wird anschließend an dieser Adresse gespeichert.

```
add $t3,$s0,$a0 # Adresse von x[i] in $t3
sb $t2, 0($t3) # x[i] = y[i]
```

Falls das nächste Zeichen 0 ist, also wenn es das letzte Zeichen der Zeichenfolge ist, wird die Schleife verlassen:

```
beq $t2,$zero,L2 # wenn y[i] == 0, verzweige zu L2
```

Wenn das nächste Zeichen nicht 0 ist, inkrementieren wir *i* und verzweigen an den Anfang der Schleife:

```
addi $s0, $s0,1 # i = i + 1
j L1             # springe zu L1
```

Wenn nicht an den Schleifenanfang gesprungen wird, wurde das letzte Zeichen der Zeichenfolge bearbeitet. Wir stellen *\$s0* und den Kellerzeiger wieder her und springen dann aus der Prozedur zurück.

```
L2: lw $s0, 0($sp) # stelle den alten Wert von $s0
                # wieder her
    addi $sp,$sp,4 # stelle den alten Wert von $sp
                # wieder her
    jr $ra          # springe zurück
```

Beim Kopieren von Zeichenfolgen in C werden in der Regel Zeiger anstelle von Feldern verwendet, um die Operationen mit *i* wie im obigen Code zu vermeiden. Zur Erläuterung sei auf Abschnitt 2.15 verwiesen, in dem Felder und Zeiger einander gegenübergestellt werden.

Da die obige Prozedur *strcpy* eine Blattprozedur ist, könnte der Compiler *i* in einem temporären Register speichern und so vermeiden, dass *\$s0* gerettet und wiederhergestellt werden muss. Daher sollte man die *\$t*-Register nicht ausschließlich für temporäre Variablen vorsehen, sondern als Register betrachten, die die aufgerufene

**Tab. 2.15 Beispiele für Zeichensätze in Unicode.** Unicode Version 4.0 besteht aus mehr als 160 „Blöcken“. So werden die Zusammenstellungen von Symbolen genannt. Jeder Block ist ein Vielfaches von 16. So beginnt Griechisch beispielsweise bei 0370<sub>H</sub> und Kyrrilisch bei 0400<sub>H</sub>. In den ersten drei Spalten sind 48 Blöcke mit Schriftzeichen menschlicher Sprachen aufgeführt. Ihre Reihenfolge entspricht in etwa der numerischen Folge in Unicode. Die letzte Spalte enthält 16 Blöcke, die für mehrere Sprachen gelten und in keiner speziellen Folge aufgeführt sind. Die 16-Bit-Codierung UTF-16 wird standardmäßig verwendet. Die Codierung in variabler Länge (UTF-8) enthält die ASCII-Zeichen als 8 Bit und verwendet 16–32 Bit für andere Zeichen. UTF-32 verwendet 32 Bit pro Zeichen. Weitere Informationen finden Sie unter [www.unicode.org](http://www.unicode.org).

Latein	Malayalam	Tagbanwa	Allgemeine Satzzeichen
Griechisch	Sinhala	Khmer	Zeichen zur Abstandsbestimmung
Kyrrilisch	Thai	Mongolisch	Währungssymbole
Armenisch	Laotisch	Limbu	Kombinierte diakritische Sonderzeichen
Hebräisch	Tibetisch	Tai Le	Kombinierte Zeichen für Symbole
Arabisch	Myanmar (Birmanisch)	Kangxi Radikale	Hoch- und tiefgestellt
Syrisch	Georgisch	Hiragana	Nummernderivate
Thaana	Hangul Jamo (Koreanisch)	Katakana	Mathematische Zeichen
Devanagari	Äthiopisch	Bopomofo	Mathematische alphanumerische Zeichen
Bengali	Cherokee	Kanbun	Blindsight
Gurmukhi	Unified Canadian Aboriginal Syllabic (Urbevölkerung Kanada)	Shavian	OCR (optische Zeichenerkennung)
Gujarati	Ogham	Osmanyia	Byzantinische Musiksymbole
Oriya	Runic	Zypriotische Silbentabelle	Musiksymbole
Tamil	Tagalog	Tai Xuan Jing-Symbole	Pfeile
Telugu	Hanunoo	Yijing-Hexagrammsymbole	Blockgrafiken
Kannada	Buhid	Ägäische Zahlen	Geometrische Formen

Prozedur wann immer möglich einsetzen soll. Wenn ein Compiler auf eine Blattprozedur stößt, nutzt er alle temporären Register, bevor er Register verwendet, die er sichern muss.

## Zeichen und Zeichenfolgen in Java

Bei *Unicode* handelt es sich um eine universelle Codierung der Zeichensätze der meisten natürlichen Sprachen. Tabelle 2.15 enthält eine Liste der Unicode-Alphabete. Es gibt in Unicode etwa so viele *Zeichensätze* wie es *Symbole* in ASCII gibt. Um möglichst umfassend zu sein, verwendet Java Unicode für die Codierung von Zeichen. Dabei wird ein Zeichen standardmäßig mit 16 Bit dargestellt.

Beim MIPS-Befehlssatz gibt es spezielle Befehle zum Laden und Speichern dieser 16-Bit-Größen, die als *Halbwörter* bezeichnet werden. Der *Load-half*-Befehl (lh) lädt ein Halbwort aus dem Hauptspeicher und legt es in den rechtsbündigen 16 Bit eines Registers ab. Der *Store-half*-Befehl (sh) nimmt ein Halbwort aus den rechtsbündigen

16 Bit eines Registers und schreibt es in den Hauptspeicher. Somit lässt sich ein Halbwort mit der folgenden Befehlsfolge kopieren:

```
lh $t0,0($sp) # lese Halbwort (16 Bit) aus Speicher  
sh $t0,0($gp) # schreibe Halbwort (16 Bit) in Speicher
```

Für Zeichenfolgen stellt Java eine Standardklasse mit spezieller Unterstützung und vordefinierten Methoden für Konkatenation, Vergleich und Konvertierung zur Verfügung. Im Gegensatz zu C wird in Java ein Wort mitgeführt, das die Länge einer Zeichenfolge ähnlich wie bei Java-Feldern angibt.

**Vertiefung:** MIPS-Software versucht, den Keller an Wortadressen auszurichten, so dass im Programm immer mit `lw` und `sw` (die ausgerichtet sein müssen) auf den Keller zugegriffen werden kann. Diese Konvention bedeutet, dass eine auf dem Keller abgelegte `char`-Variable 4 Byte belegt, auch wenn sie eigentlich weniger Speicherplatz benötigt. In C werden bei einer Variablen einer Zeichenfolge oder eines Felds mit Bytes 4 Byte pro Wort gepackt. In Java werden bei einer Variablen einer Zeichenfolge oder bei einem Feld mit Elementen vom Typ `short` 2 Halbwörter pro Wort zusammengefasst.



Welche der folgenden Aussagen über Zeichen und Zeichenfolgen in C und Java treffen zu?



1. Eine Zeichenfolge in C belegt etwa halb so viel Speicherplatz wie dieselbe Zeichenfolge in Java.
2. Zeichenfolge ist nur eine saloppe Bezeichnung für eindimensionale `char`-Felder in C und Java.
3. Bei Zeichenfolgen in C und Java wird das Ende einer Zeichenfolge mit Null (0) gekennzeichnet.
4. Operationen auf Zeichenfolgen, wie `length`, können in C schneller durchgeführt werden als in Java.

## 2.9

# Umgang mit 32-Bit-Direktoperanden und 32-Bit-Adressen

Obwohl ein festes 32-Bit-Format für alle Befehle die Hardware vereinfacht, gibt es Fälle, in denen es praktisch wäre, die Möglichkeit für 32-Bit-Konstanten oder 32-Bit-Adressen zu haben. Dieser Abschnitt beginnt mit der allgemeinen Lösung für lange Konstanten und zeigt Optimierungsmöglichkeiten für Befehlsadressen in Verzweigungen und Sprüngen auf.

## 32-Bit-Direktoperanden

In der Regel sind Konstanten kurz und passen in das 16-Bit-Feld. Gelegentlich sind sie jedoch etwas länger. Der MIPS-Befehlssatz enthält den Befehl `lui` (*load upper immediate, lade höherwertige Hälfte des Direktoperanden*), mit dem die höherwertigen 16 Bit einer Konstante in ein Register geladen werden, so dass in einem nachfolgenden Befehl die niedrigerwertigen 16 Bit der Konstante spezifiziert werden können. Abbildung 2.8 zeigt die Funktionsweise des Befehls `lui`.

Der Maschinencode von lui \$t0,255 # \$t0 ist Register 8:

00111	00000	0100	0000 0000 1111 1111
-------	-------	------	---------------------

Inhalt des Registers \$t0 nach der Ausführung des Befehls lui \$t0,255:

0000 0000 1111 1111	0000 0000 0000 0000
---------------------	---------------------

Abb. 2.8 Die Wirkungsweise des Befehls lui. Der Befehl lui überträgt den Wert im 16-Bit-Direktoperandenfeld in die linksbündigen 16 Bit des Registers und füllt die unteren 16 Bit mit Nullen.

## Hardware-Software-Schnittstelle

Es ist die Aufgabe des Compilers oder des Assemblers, lange Konstanten aufzuteilen und anschließend in einem Register wieder zusammenzusetzen. Wie Sie vielleicht erwarten, kann die Größenbeschränkung des Direktoperandenfelds für Speicheradressen bei Lade- und Speichervorgängen sowie für Konstanten in Immediate-Befehlen ein Problem darstellen. Wenn diese Aufgabe der Assembler übernimmt, wie dies bei der MIPS-Software der Fall ist, muss der Assembler über ein temporäres Register verfügen, in dem lange Werte gebildet werden können. Dies ist der Grund dafür, warum es das für den Assembler reservierte Register \$at gibt.

Auf diese Weise wird die symbolische Darstellung der MIPS-Maschinensprache nicht mehr durch die Hardware beschränkt, sondern hängt davon ab, was der Entwickler eines Assemblers mit aufnehmen möchte (siehe Abschnitt 2.10). Wir orientieren uns bei der Erklärung der Architektur eines Rechners stark an der Hardware und weisen darauf hin, wenn wir die Konstrukte der erweiterten Sprache des Assemblers verwenden, die nicht vom Prozessor direkt unterstützt werden.

## BEISPIEL

### Laden einer 32-Bit-Konstante

Wie lautet der MIPS-Assemblercode zum Laden der folgenden 32-Bit-Konstante in Register \$s0?

0000 0000 0011 1101 0000 1001 0000 0000

## ANTWORT

Zunächst laden wir mit dem Befehl lui die oberen 16 Bit, die in Dezimalschreibweise dem Wert 61 entsprechen:

lui \$s0, 61 # 61 dezimal = 0000 0000 0011 1101 binär

Der Wert von Register \$s0 lautet danach

0000 0000 0011 1101 0000 0000 0000 0000

Im nächsten Schritt addieren wir die unteren 16 Bit, die in Dezimalschreibweise dem Wert 2304 entsprechen:

ori \$s0, \$s0, 2304 # 2304 dezimal  
# = 0000 1001 0000 0000 binär

Schließlich befindet sich in Register \$s0 der gewünschte Wert:

0000 0000 0011 1101 0000 1001 0000 0000

**Vertiefung:** Beim Zusammensetzen von 32-Bit-Konstanten muss mit Vorsicht vorgegangen werden. Der Befehl addi kopiert das höchstwertige Bit des 16-Bit-immediate-Felds des Befehls in die oberen 16 Bit eines Wortes. Mit den *logischen Operationen oder Immediate-Befehlen* aus Abschnitt 2.5 werden dagegen Nullen in die oberen 16 Bit geladen und daher vom Assembler zusammen mit dem Befehl lui zum Bilden von 32-Bit-Konstanten verwendet.



## Adressbildung bei Verzweigungen und Sprüngen

Die MIPS-Sprungbefehle verwenden die einfachste Adressierungsart. Für sie gibt es ein weiteres MIPS-Befehlsformat, das so genannte *J-Typ-Format*. Es setzt sich aus dem 6 Bit breiten Operationsfeld und dem Adressfeld zusammen, das die restlichen Bits umfasst. Somit könnte

```
j 10000 # springe an Stelle 10000
```

in folgendes Format assembledt werden. (Es ist etwas komplizierter, wie auf der folgenden Seite zusehen sein wird.)

2	10000
6 Bit	26 Bit

wobei der Wert für den Opcode des Sprungbefehls 2 beträgt und die Sprungadresse 10000 ist.

Im Gegensatz zum Sprungbefehl, müssen beim bedingten Verzweigungsbefehl neben der Sprungadresse zwei Operanden angegeben werden. Die Verzweigung

```
bne $s0,$s1,Exit # verzweige nach Exit, wenn $s0 ≠ $s1
```

wird in folgendes Format assembledt, in dem nur noch 16 Bit für die Sprungadresse zur Verfügung stehen:

5	16	17	Exit
6 Bit	5 Bit	5 Bit	16 Bit

Wenn die Programmadressen in dieses 16-Bit-Feld passen müssten, würde dies bedeuten, dass kein Programm größer als  $2^{16}$  Bytes sein dürfte, was viel zu wenig ist, um heute eine realistische Option zu sein. Eine Alternative wäre die Festlegung eines Registers, das immer zur Sprungadresse addiert wird. Der Verzweigungsbefehl würde dann die Folgeadresse mit

$$\text{Befehlszähler} = \text{Register} + \text{Sprungadresse}$$

berechnen. Auf diese Weise kann das Programm eine Größe von bis zu  $2^{32}$  Bytes annehmen und dennoch bedingte Sprünge verwenden, womit das Größenproblem bei Sprungadressen gelöst ist. Es stellt sich die Frage, welches Register verwendet werden könnte.

Die Lösung finden wir, wenn wir betrachten, wie Verzweigungen bzw. bedingte Sprünge verwendet werden. Bedingte Sprünge finden man in Schleifen und in *If*-Anweisungen, d.h. bedingte Sprünge verweisen auf nahe gelegene Befehle. Beispielsweise verzweigt etwa die Hälfte aller bedingten Sprünge in SPEC2000-Benchmarks an Stellen, die nicht weiter als 16 Befehle entfernt sind. Da der Befehlszähler die Adresse des aktuellen Befehls enthält, können wir in einen Bereich von  $\pm 2^{15}$  Wörtern vom aktuellen Befehl aus verzweigen, wenn wir den Befehlszähler als Register verwenden, das zur Adresse addiert wird. Fast alle Schleifen und *If*-Anweisungen sind wesentlich kleiner als  $2^{16}$  Wörter, so dass der Befehlszähler hierfür die richtige Wahl darstellt.

Diese Art der Adressierung bei Sprüngen wird als **befehlszählerrelative Adressierung (PC-relative addressing)** bezeichnet. Wie in Kapitel 5 zu sehen sein wird, ist es von Vorteil, wenn der Befehlszähler frühzeitig inkrementiert wird, um auf den

**Befehlszählerrelative Adressierung (PC-relative addressing)** Eine Adressierungsart, bei der die Adresse durch die Summe von Befehlszähler und einer konstanten Abstandsgroße im Befehl gebildet wird.

nächsten Befehl zu zeigen. Die Adresse bei MIPS ist damit relativ zur Adresse des nachfolgenden Befehls (Befehlszähler + 4) und nicht zum aktuellen Befehl (Befehlszähler).

Wie die meisten modernen Prozessoren verwendet MIPS die befehlszählerrelative Adressierung für alle Verzweigungen bzw. bedingten Sprünge, da das Sprungziel bei diesen Befehlen sehr wahrscheinlich nahe bei der Verzweigung ist. Dagegen rufen *Jump-and-Link*-Befehle Prozeduren auf, bei denen dies nicht der Fall ist. Daher werden für diese in der Regel andere Adressierungsarten verwendet. Die MIPS-Architektur stellt lange Adressen für Prozeduraufrufe mithilfe des J-Formats sowohl für Sprung- als auch für *Jump-and-Link*-Befehle bereit.

Da alle MIPS-Befehle 4 Byte lang sind, wird bei MIPS der Sprungbereich für eine Verzweigung vergrößert, in dem sich die befehlszählerrelative Adressierung auf die Anzahl der Wörter bis zum nächsten Befehl anstelle der Anzahl der Bytes bezieht. Mit einer konstanten Abstandsgröße im 16-Bit-Feld kann man viermal so weit verzweigen, wenn das Feld nicht als eine relative Byteadresse, sondern als relative Wortadresse interpretiert wird. Entsprechend ist auch das 26-Bit-Feld in Sprungbefehlen eine Wortadresse, d.h. es stellt eine 28-Bit-Byteadresse dar.



**Vertiefung:** Da der Befehlszähler 32 Bit umfasst, müssen 4 Bit von anderer Stelle bereitgestellt werden. Der MIPS-Sprungbefehl ersetzt nur die unteren 28 Bit des Befehlszählers und belässt die oberen 4 Bit des Befehlszählers unverändert. Der Lader und der Binder (Abschnitt 2.10) müssen darauf achten, dass kein Programm über die Adressgrenze von 256 MB (64 Millionen Befehle) hinweg geladen wird. Andernfalls muss der Sprungbefehl durch einen *Jump-register*-Befehl ersetzt werden, wobei andere Befehle zum Laden der vollständigen 32-Bit-Adresse in ein Register vorangestellt werden müssen.

## BEISPIEL

### Sprung-Offset in Maschinensprache

Die *While*-Schleife auf Seite 60 wurde in den folgenden MIPS-Assemblercode kompiliert:

```

Loop: sll $t1,$s3,2 # temp. Reg. $t1 = 4 * i
      add $t1,$t1,$s6 # $t1 = Adresse von save[i]
      lw   $t0,0($t1)  # temp. Reg. $t0 = save[i]
      bne $t0,$s5, Exit # verzweige nach Exit,
                         # wenn save[i] ≠ k
      addi $s3,$s3,1   # i = i + 1
      j   Loop          # springe zu Loop
Exit:

```

Angenommen, die Schleife beginnt an Stelle 80000 im Hauptspeicher, wie sieht der MIPS-Maschinencode für diese Schleife aus?

## ANTWORT

Die assemblierten Befehle und deren Adressen würden wie folgt aussehen:

	ell	x	\$s3	\$s1	2	
80000	0	0	19	9	4	0
80004	0	9	22	9	0	32
80008	35	9	8		0	
80012	5	8	21		2	
80016	8	19	19		1	
80020	2				20000	
80024	...					

MIPS-Befehle stehen an Byteadressen, so dass sich aufeinander folgende Wörter um 4, also um die Anzahl der Bytes in einem Wort unterscheiden. Der Befehl `bne` in der vierten Zeile addiert 2 Wörter bzw. 8 Byte zur Adresse des *nachfolgenden* Befehls (80016) und spezifiziert das Sprungziel relativ zum nachfolgenden Befehl (8 + 80016) und nicht relativ zum Verzweigungsbefehl (12 + 80012) oder mithilfe der vollständigen Zieladresse (80024). Der Sprungbefehl in der letzten Zeile verwendet die vollständige Adresse ( $20000 \times 4 = 80000$ ), die der Marke Loop entspricht.

## Hardware-Software-Schnittstelle

Die meisten bedingten Sprünge verzweigen innerhalb eines beschränkten Adressbereichs. Es gibt jedoch Situationen, in denen weiter verzweigt werden muss, als dies in den 16 Bit des bedingten Sprungbefehls dargestellt werden kann. Der Assembler löst dieses Problem auf ähnliche Weise wie das Problem mit den langen Adressen bzw. Konstanten: Er fügt einen unbedingten Sprung mit dem Sprungziel nach der Verzweigung ein und invertiert die Bedingung, so dass die Verzweigung entscheidet, ob der Sprung genommen wird.

### Weite Verzweigung

Gegeben sei eine Verzweigung, die prüft, ob Register `$s0` gleich Register `$s1` ist:

```
beq $s0,$s1, L1
```

Ersetzen Sie die Verzweigung durch zwei Befehle, mit denen über eine wesentliche größere Distanz gesprungen werden kann.

Diese Befehle ersetzen den bedingten Sprung mit kurzer Adresse:

```
bne $s0,$s1, L2
j L1
L2 :
```

### BEISPIEL

### ANTWORT

## MIPS-Adressierungsarten – eine Übersicht

Verschiedene Formen der Adressberechnung werden im Allgemeinen als **Adressierungsarten** (*addressing mode*) bezeichnet. Die MIPS-Architektur kennt folgende Adressierungsarten:

1. **Registeradressierung.** Der Operand steht in einem Register.
2. **Basis- oder Displacement-Adressierung.** Der Operand befindet sich im Speicher an einer Stelle, deren Adresse sich aus der Summe des Inhalts eines Registers und einer konstanten Abstandsgröße im Befehl ergibt.
3. **Direkte Adressierung.** Der Operand ist eine Konstante im Befehl selbst.
4. **Befehlszählerrelative Adressierung.** Die Adresse wird aus der Summe des Befehlszählers und einer konstanten Abstandsgröße im Befehl gebildet.
5. **Pseudo-direkte Adressierung.** Die Sprungadresse wird durch Konkatenation der 26 Bits des Befehls mit den oberen Bits des Befehlszählers gebildet.

**Adressierungsart (addressing mode)** Eine von mehreren Möglichkeiten zur Adressberechnung. Die Adressierungsarten unterscheiden sich in der Verwendung von Operanden und/oder Adressen.

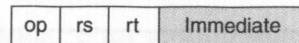
## Hardware-Software-Schnittstelle



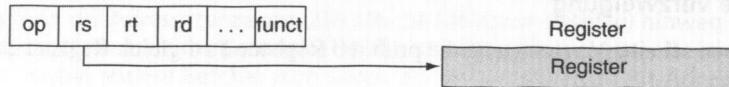
Wir gehen bei der Beschreibung der MIPS-Architektur von 32-Bit-Adressen aus. Nahezu alle Mikroprozessoren (auch MIPS-Prozessoren) verfügen über eine Erweiterung auf 64-Bit-Adressen (siehe Appendix D). Diese Erweiterungen sind die Antwort auf den Bedarf der Software im Hinblick auf größere Programme. Die Befehlssatzerweiterung ermöglicht, die Architekturen in einer Weise weiterzuentwickeln, mit der Software unter Wahrung der Aufwärtskompatibilität auf die nächste Generation einer Architektur portiert werden kann.

Für eine Operation können mehrere Adressierungsarten verwendet werden. So kann beispielsweise für eine Add-Operation sowohl die direkte (addi) als auch die Registeradressierung (add) verwendet werden. In Abbildung 2.9 ist dargestellt, wie die Operanden mit der jeweiligen Adressierungsart spezifiziert werden.

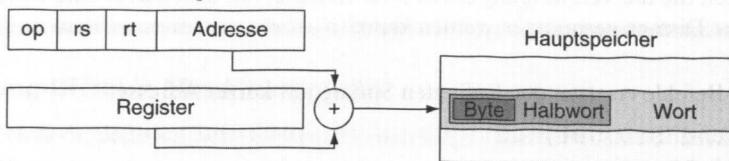
### 1. Direkte Adressierung



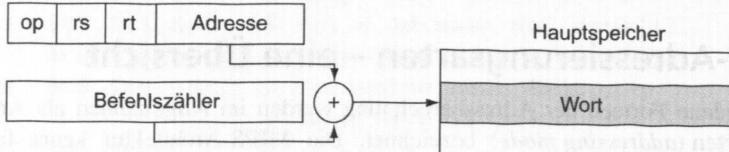
### 2. Registeradressierung



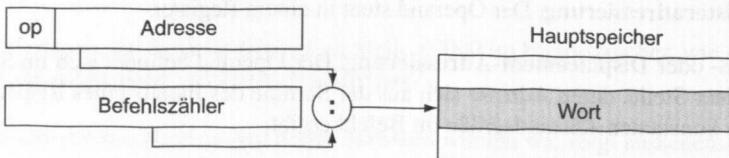
### 3. Basisadressierung



### 4. Befehlszählerrelative Adressierung



### 5. Pseudodirekte Adressierung



**Abb. 2.9 Darstellung der fünf MIPS-Adressierungsarten.** Die Operanden sind mit unterschiedlichen Hintergrundfarben hervorgehoben. Der Operand der Adressierungsart 3 befindet sich im Hauptspeicher, während sich der Operand der Adressierungsart 2 in einem Register befindet. Die verschiedenen Versionen der Load- und Store-Befehle greifen auf Bytes, Halbwörter bzw. Wörter zu. Bei der Adressierungsart 1 steht der Operand im 16-Bit-Feld des Befehls. Die Adressierungsarten 4 und 5 adressieren Befehle im Hauptspeicher, wobei bei der Adressierungsart 4 eine um 2 Bit nach links zum Befehlszähler hin verschobene 16-Bit-Adresse addiert wird und bei der Adressierungsart 5 eine um 2 Bit nach links verschobene 26-Bit-Adresse mit den oberen 4 Bit des Befehlszählers verknüpft wird.

# Entschlüsseln der Maschinensprache

Es gibt Situationen, in denen Maschinensprache in die ursprüngliche Assemblersprache „rückübersetzt“ werden muss, wenn Sie beispielsweise einen Hauptspeicherauszug betrachten möchten. In Tabelle 2.16 ist die MIPS-Codierung der Felder für die MIPS-Maschinensprache dargestellt. Diese Abbildung erleichtert das manuelle Übersetzen zwischen Assembler- und Maschinensprache.

**Tab. 2.16 MIPS-Befehlscodierung.** Diese Notation liefert den Wert eines Felds über die Zeilen- und Spaltennummer. Beispiel: Im oberen Teil der Abbildung steht `load` `word` in Zeile 4 (100<sub>B</sub> für Bit 31–29 des Befehls) und Spalte 3 (011<sub>B</sub> für Bit 28–26 des Befehls), so dass der entsprechende Wert des op-Felds (Bit 31–26) 100011<sub>B</sub> lautet. Ein farblich nicht hervorgehobener Wert bedeutet, dass das Feld an einer anderen Stelle erklärt wird. Beispiel: R-format in Zeile 0 und Spalte 0 (`op = 000000B`) ist im unteren Teil der Abbildung definiert. Somit bedeutet `subtract` in Zeile 4 und Spalte 2 im unteren Bereich, dass das funct-Feld (Bit 5–0) des Befehls 100010<sub>B</sub> ist und das op-Feld (Bit 31–26) 000000<sub>B</sub> ist. Der F1Pt-Wert in Zeile 2, Spalte 1 wird in Tabelle 3.9 in Kapitel 3 definiert. `Bltz/gez` ist der Opcode für vier Befehle in **Appendix A**: `bltz`, `bgez`, `bltzal` und `bgezal`. In Kapitel 2 werden Befehle mit vollständigem Namen in Farbe beschrieben, während in Kapitel 3 Befehle mit mnemonischer Bezeichnung in Farbe beschrieben werden. In **Appendix A** sind alle Befehle aufgeführt. ☀

op(31:26):								
28-26 31-29	0 (000)	1 (001)	2 (010)	3 (011)	4 (100)	5 (101)	6 (110)	7 (111)
0(000)	R-format	Bltz/gez	jump	jump & link	branch eq	branch ne	blez	bgtz
1(001)	add immediate	addiu	set less than imm.	sltiu	andi	ori	xori	load upper imm
2(010)	TLB	F1Pt						
3(011)								
4(100)	load byte	load half	lw1	load word	lbu	lhu	lwr	
5(101)	store byte	store half	sw1	store word			swr	
6(110)	lwcl0	lwcl1						
7(111)	swcl0	swcl1						

**BEISPIEL****Entschlüsseln des Maschinencodes**

Wie lautet die Anweisung in Assemblersprache, die diesem Befehl in Maschinen-sprache entspricht?

00af8020hex

**ANTWORT**

Der erste Schritt beim Konvertieren des Hexadezimalcodes in Binärkode besteht darin, die op-Felder zu suchen:

Bit:	31	28	26			5	2	0
	0000	0000	1010	1111	1000	0000	0010	0000

Über das op-Feld kann die Operation bestimmt werden. Nach Tabelle 2.16 handelt es sich um einen Befehl im R-Format, wenn die Bitstellen 31–29 und die Bitstellen 28–26 jeweils 000 sind. Gemäß Tabelle 2.17 kann der binäre Befehl in die Felder des R-Formats umgeformt werden:

op	rs	rt	rd	shamt	funct
000000	00101	01111	10000	00000	100000

Der untere Teil von Tabelle 2.16 bestimmt die Operation eines Befehls im R-Format. In diesem Fall enthalten die Bitstellen 5–3 100 und die Bitstellen 2–0 000, d.h. dieses Binärmuster repräsentiert einen add-Befehl.

Durch Betrachtung der Werte in den Feldern wird der Rest des Befehls entschlüsselt. Der Dezimalwert für das rs-Feld ist 5, der für das rt-Feld 15 und der für das rd-Feld 16. (shamt ist nicht belegt.) Nach Tabelle 2.18 stehen diese Zahlen für die Register \$a1, \$t7 und \$s0. Der Assemblerbefehl ist:

add \$s0,\$a1,\$t7

In Tabelle 2.17 sind alle MIPS-Befehlsformate dargestellt. Die in Kapitel 2 behandelte MIPS-Assemblersprache zeigt Tabelle 2.18. Der noch verbleibende, noch nicht erläuterte Teil der MIPS-Befehle umfasst hauptsächlich arithmetische Operationen, die im nächsten Kapitel behandelt werden.

**Tab. 2.17 MIPS-Befehlsformate in Kapitel 2.** In diesem Abschnitt neu eingeführte Befehlsformate sind farblich hervorgehoben.

Name	Felder							Anmerkungen
Feldgröße	6 Bit	5 Bit	5 Bit	5 Bit	5 Bit	6 Bit	Alle MIPS-Befehle 32 Bit lang	
R-Format	op	rs	rt	rd	shamt	funct	Format für arithmetische Befehle	
I-Format	op	rs	rt	address/immediate			Datentransport, Sprung, Immediate-Format	
J-Format	op	Zieladresse						Sprungbefehlsformat

Tab. 2.18 In Kapitel 2 bearbeitete MIPS-Assemblersprache. Die Teile aus den Abschnitten 2.8 und 2.9 sind farblich hervorgehoben.

**MIPS-Operanden**

Name	Beispiel	Anmerkungen
32 Register	\$s0-\$s7, \$t0-\$t9, \$zero, \$a0-\$a3, \$v0-\$v1, \$gp, \$fp, \$sp, \$ra, \$at	Speicherort für schnellen Zugriff auf Daten. Bei MIPS müssen die Daten für die arithmetischen Operationen in Registern stehen. MIPS-Register \$zero ist immer 0. Register \$at ist für den Assembler für die Arbeit mit langen Konstanten reserviert.
$2^{30}$ Speicherwörter	Speicher[0], Speicher[4], ..., Speicher[4294967292]	Zugriff nur durch Datentransport-Befehle. Die MIPS-Architektur verwendet Byteadressen. Daher unterscheiden sich aufeinander folgende Wortadressen um 4. Im Hauptspeicher werden Datenstrukturen, Felder und ausgelagerte Register gespeichert, wie die, die bei einem Prozederaufruf gespeichert werden.

**MIPS-Assemblersprache**

Kategorie	Befehl	Beispiel	Bedeutung	Anmerkungen
Arithmetischer Befehl	add	add \$s1, \$s2, \$s3	$\$s1 = \$s2 + \$s3$	Drei Registeroperanden
	subtract	sub \$s1, \$s2, \$s3	$\$s1 = \$s2 - \$s3$	Drei Registeroperanden
	add immediate	addi \$s1, \$s2, 100	$\$s1 = \$s2 + 100$	Zum Addieren von Konstanten verwendet
Datentransfer	load word	lw \$s1, 100(\$s2)	$\$s1 = \text{Speicher}[\$s2 + 100]$	Wort vom Hauptspeicher in ein Register
	store word	sw \$s1, 100(\$s2)	$\text{Speicher}[\$s2 + 100] = \$s1$	Wort von einem Register in den Hauptspeicher
	load half	lh \$s1, 100(\$s2)	$\$s1 = \text{Speicher}[\$s2 + 100]$	Halbwort vom Hauptspeicher in ein Register
	store half	sh \$s1, 100(\$s2)	$\text{Speicher}[\$s2 + 100] = \$s1$	Halbwort von einem Register in den Hauptspeicher
	load byte	lb \$s1, 100(\$s2)	$\$s1 = \text{Speicher}[\$s2 + 100]$	Byte vom Hauptspeicher in ein Register
	store byte	sb \$s1, 100(\$s2)	$\text{Speicher}[\$s2 + 100] = \$s1$	Byte von einem Register in den Hauptspeicher
	load upper immed.	lui \$s1, 100	$\$s1 = 100 * 2^{16}$	Lädt Konstante in obere 16 Bit
Logischer Befehl	and	and \$s1, \$s2, \$s3	$\$s1 = \$s2 \& \$s3$	Drei Registeroperanden; bitweise UND-Verknüpfung
	or	or \$s1, \$s2, \$s3	$\$s1 = \$s2   \$s3$	Drei Registeroperanden; bitweise ODER-Verknüpfung
	nor	nor \$s1, \$s2, \$s3	$\$s1 = \sim (\$s2   \$s3)$	Drei Registeroperanden; bitweise NOR-Verknüpfung
	and immediate	andi \$s1, \$s2, 100	$\$s1 = \$s2 \& 100$	Bitweise UND-Verknüpfung von Registeroperanden mit Konstante
	or immediate	ori \$s1, \$s2, 100	$\$s1 = \$s2   100$	Bitweise ODER-Verknüpfung von Registeroperanden mit Konstante
	shift left logical	sll \$s1, \$s2, 10	$\$s1 = \$s2 << 10$	Linksverschieben um Konstante
	shift right logical	srl \$s1, \$s2, 10	$\$s1 = \$s2 >> 10$	Rechtsverschieben um Konstante
Verzweigung	branch on equal	beq \$s1, \$s2, 25	wenn ( $\$s1 == \$s2$ ), verzweige zu PC + 4 + 100	Überprüfen auf Gleichheit; befehlssählerrelative Verzweigung
	branch on not equal	bne \$s1, \$s2, 25	wenn ( $\$s1 != \$s2$ ), verzweige zu PC + 4 + 100	Überprüfen auf Ungleichheit; befehlssählerrelative Verzweigung
	set on less than	slt \$s1, \$s2, \$s3	wenn ( $\$s2 < \$s3$ ), setze \$s1 = 1, ansonsten \$s1 = 0	Vergleich: kleiner als; verwendet mit beq, bne
	set less than immediate	slti \$s1, \$s2, 100	wenn ( $\$s2 < 100$ ), setze \$s1 = 1, ansonsten \$s1 = 0	Vergleich: kleiner als Konstante
Unbedingter Sprung	jump	j 2500	springe zu 10000	Sprung zu Zieladresse
	jump register	jr \$ra	springe zu \$ra	Für Switch-Anweisung, Prozedur-rücksprung
	jump and link	jal 2500	$\$ra = PC + 4$ , springe zu 10000	Für Prozederaufruf



Wie lautet der Adressbereich für bedingte Sprünge bei MIPS ( $K = 1024$ )?

1. Adressen zwischen 0 und 64 K – 1
2. Adressen zwischen 0 und 256 K – 1
3. Adressen bis zu etwa 32 K vor der Verzweigung bis etwa 32 K nach der Verzweigung
4. Adressen bis zu etwa 128 K vor der Verzweigung bis etwa 128 K nach der Verzweigung

Wie lautet der Adressbereich für *Sprung*- und *Jump-and-Link*-Befehle bei MIPS ( $M = 1024$  K)?

1. Adressen zwischen 0 und 64 M – 1
2. Adressen zwischen 0 und 256 M – 1
3. Adressen bis zu etwa 32 M vor der Verzweigung bis etwa 32 M nach der Verzweigung
4. Adressen bis zu etwa 128 M vor der Verzweigung bis etwa 128 M nach der Verzweigung
5. An einer beliebigen Stelle in einem Block von 64 M-Adressen, wobei der Befehlszähler die oberen 6 Bits bereitstellt
6. An einer beliebigen Stelle in einem Block von 256 M-Adressen, wobei der Befehlszähler die oberen 4 Bits bereitstellt

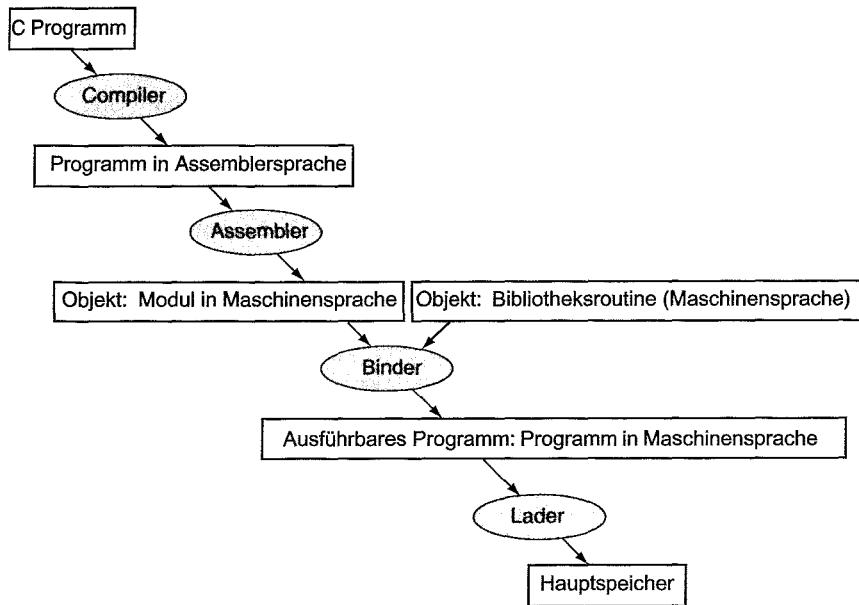
Wie lautet der MIPS-Befehl in Assemblersprache, der dem Maschinenbefehl mit dem Wert  $0000\ 0000_H$  entspricht?

1. j
2. R-format
3. addi
4. sll
5. mfc0
6. Nicht definierter Opcode: Es gibt keinen zulässigen Befehl mit dem Wert 0.

## 2.10

## Übersetzen und Starten eines Programms

In diesem Abschnitt werden die vier Schritte beschrieben, die erforderlich sind, um ein C-Programm in einer Datei auf der Festplatte in ein Programm umzuwandeln, das auf einem Computer ausgeführt werden kann. In Abbildung 2.10 ist die Übersetzungs-hierarchie dargestellt. Bei einigen Systemen sind diese Schritte zusammengefasst, um die Übersetzungszeit zu reduzieren. Dennoch durchlaufen alle Programme diese vier logischen Phasen. Daher halten wir uns in diesem Abschnitt an diese Übersetzungs-hierarchie.



**Abb. 2.10 Eine Übersetzungshierarchie für C.** Ein Programm in einer höheren Programmiersprache wird zunächst in ein Programm in Assemblersprache übersetzt und anschließend in ein Objektmodul in Maschinensprache assembliert. Der Binder fügt mehrere Module mit Bibliotheks Routinen zusammen, um alle Referenzen aufzulösen. Der Lader lädt den Maschinencode an die entsprechende Stelle im Hauptspeicher für die Ausführung durch den Prozessor. Um die Übersetzung zu beschleunigen, werden einige Schritte übersprungen oder zusammengefasst. Manche Compiler erstellen Objektmodule direkt, manche Systeme führen die letzten beiden Schritte mithilfe von bindenden Ladern in einem Schritt aus. UNIX folgt zur Kennzeichnung der verschiedenen Typen der Dateien der folgenden Konvention für die Erweiterungen der Dateinamen: C-Quelldateien haben die Bezeichnung `x.c`, Assemblerdateien `x.s`, Objektdateien `x.o`, statisch gebundene Bibliotheks Routinen `x.a`, dynamisch gebundene Bibliotheks Routinen `x.so`, und ausführbare Programmdateien haben die Standardbezeichnung `a.out`. Bei MS-DOS werden die Erweiterungen `.C`, `.ASM`, `.OBJ`, `.LIB`, `.DLL` und `.EXE` entsprechend verwendet.

## Compiler

Der Compiler wandelt das C-Programm in ein *Programm in Assemblersprache* um, d.h. in eine symbolische Darstellung dessen, was die Maschine versteht. Programme, die in einer höheren Programmiersprache geschrieben sind, brauchen deutlich weniger Codezeilen als Programme in Assemblersprache, weshalb die Produktivität der Programmierer weitaus höher ist.

1975 waren viele Betriebssysteme und Assembler wegen der geringen Hauptspeicherkapazitäten und wegen ineffizienter Compiler in **Assemblersprache (assembly language)** geschrieben. Dank einer 128 000fachen Zunahme der Speicherkapazität pro DRAM-Chip stellt die Größe von Programmen heute kein Problem mehr dar. Optimierende Compiler können heute nahezu so guten Code in Assemblersprache generieren wie ausgewiesene Experten in Assembler-Programmierung und bei großen Programmen in vielen Fällen sogar besseren.

**Assemblersprache (assembly language)** Eine symbolische Sprache, die in Binär code übersetzt werden kann.

## Assembler

**Pseudobefehle (*pseudoinstruction*)** Eine allgemeine Variation von Befehlen in der Assemblersprache, die wie ein tatsächlicher Befehl behandelt wird.

Wie auf Seite 80 erwähnt, bildet die Assemblersprache die Schnittstelle hin zu den höheren Ebenen der Software. Der Assembler kann daher auch allgemeine Variationen von Maschinenbefehlen behandeln, so als ob sie tatsächliche Befehle wären. Diese Befehle müssen nicht unbedingt in Hardware implementiert sein. Deren Darstellung in der Assemblersprache erleichtert jedoch die Übersetzung und das Programmieren. Befehle dieser Art werden als **Pseudobefehle (*pseudoinstruction*)** bezeichnet.

Wie bereits erwähnt, ist durch die MIPS-Hardware sichergestellt, dass Register \$zero immer den Wert 0 enthält. Register \$zero liefert damit bei jeder Verwendung den Wert 0 und der Programmierer kann den Wert von Register \$zero nicht ändern. Register \$zero wird zur Bildung des Assembler-Befehls move verwendet, mit dem der Inhalt eines Registers in ein anderes kopiert wird. Der MIPS-Assembler akzeptiert somit den folgenden Befehl, obwohl dieser in der MIPS-Architektur nicht enthalten ist:

```
move $t0,$t1 # Inhalt von Reg. $t1 nach Reg. $t0
```

Der Assembler wandelt diesen Befehl der Assemblersprache in die äquivalente Maschinendarstellung des folgenden Befehls um:

```
add $t0,$zero,$t1 # 0 + Reg. $t1 nach Reg. $t0
```

Der MIPS-Assembler wandelt ebenso den Pseudobefehl blt (branch on less than) in die beiden im Beispiel auf Seite 69 genannten Maschinenbefehle slt und bne um. Die Befehle bgt, bge und ble sind weitere Beispiele. Außerdem setzt er Sprünge an weit entfernte Stellen in eine Verzweigung und einen unbedingten Sprung um. Wie bereits erwähnt, ermöglicht der MIPS-Assembler das Laden von 32-Bit-Konstanten in ein Register trotz der bestehenden 16-Bit-Beschränkung bei Immediate-Befehlen.

Aufgrund der Pseudobefehle kann MIPS auf einen größeren Satz an Befehlen in Assemblersprache zurückgreifen als nur auf die durch die Hardware implementierten Befehle. Der einzige Nachteil dabei ist, dass ein Register, nämlich \$at, für den Assembler reserviert werden muss. Wenn Sie Assemblerprogramme schreiben, erleichtern Sie sich diese Aufgabe durch die Verwendung von Pseudobefehlen. Um die MIPS-Architektur zu verstehen und die beste Leistung zu erzielen, sollten Sie jedoch die echten MIPS-Befehle in Tabelle 2.16 und 2.18 beachten.

Assembler erlauben außerdem Zahlen mit unterschiedlicher Basis. Neben Binär- und Dezimalzahlen akzeptieren sie üblicherweise eine kürzere als die binäre Basis, die sich aber leicht in ein Bitmuster konvertieren lässt. MIPS-Assembler verwenden Hexadezimalzahlen.

Diese Eigenschaften sind praktisch, aber die Hauptaufgabe eines Assemblers besteht darin, Maschinencode zu erzeugen. Der Assembler übersetzt ein Programm in Assemblersprache in eine *Objektdatei*, die sich aus Befehlen in **Maschinensprache (*machine language*)**, Daten und Informationen zum Ablegen der Befehle an die richtigen Positionen im Hauptspeicher zusammensetzt.

Um die binäre Version für jeden Befehl im Assembler-Programm zu generieren, muss der Assembler die entsprechenden Adressen für alle Marken ermitteln. Assembler halten mithilfe einer **Symboltabelle (*symbol table*)** die in Sprüngen und Datentransfer-Befehlen verwendeten Marken fest. Die Tabelle enthält Paare aus jeweils einem Symbol und einer Adresse.

Die Objektdatei für UNIX-Systeme besteht üblicherweise aus sechs verschiedenen Teilen:

- Der *Header* der Objektdatei beschreibt die Größe und Position der anderen Teile der Objektdatei.
- Das *Textsegment* enthält den Code in Maschinensprache.

**Maschinensprache (*machine language*)** Binäre Darstellung, die für die Kommunikation in einem Rechnersystem verwendet wird.

**Symboltabelle (*symbol table*)** Eine Tabelle, mit deren Hilfe die Namen der Marken den Adressen der Wörter im Speicher zugeordnet werden können.

- Das *statische Datensegment* enthält die Daten, die für die Dauer des Programms zugeteilt werden. (UNIX erlaubt Programmen die Verwendung entweder von *statischen Daten*, die für die Dauer der Programmausführung zugeteilt sind, oder von *dynamischen Daten*, die ihre Größe je nach Anforderung des Programms ändern.)
- Mit der *Relocation Information* werden Befehls- und Datenwörter identifiziert, die beim Laden des Programms in den Hauptspeicher von absoluten Adressen abhängen.
- Die *Symboltabelle* enthält die restlichen, nicht definierten Marken, wie z.B. externe Referenzen.
- Die *Debug-Informationen* enthalten eine kurze Beschreibung, wie die Module übersetzt wurden, so dass ein Debugger die Maschinenbefehle den C-Quelldateien zuordnen und Datenstrukturen lesbar machen kann.

Im nächsten Abschnitt wird beschrieben, wie bereits assemblierte Routinen wie z.B. Bibliotheksroutine hinzugebunden werden.

## Binder

Aus dem bisher Erläuterten kann der Eindruck entstehen, als müsse aufgrund einer einzigen Änderung in einer Zeile einer Prozedur das gesamte Programm neu übersetzt und assembliert werden. Eine vollständige Neuübersetzung ist eine unnötige Vergeudung von Rechenressourcen. Eine solche Wiederholung des Übersetzungsorgangs stellt insbesondere bei Standardbibliotheksroutine eine Verschwendug dar, da Programmierer Routinen kompilieren und assemblieren würden, die sich per Definition praktisch nie ändern. Eine Alternative hierzu ist, jede Prozedur unabhängig zu übersetzen und zu assemblieren, so dass bei einer Änderung in einer Zeile nur eine Prozedur neu kompiliert und assembliert werden muss. Diese Möglichkeit erfordert jedoch ein neues Systemprogramm, das als **Binder (linker)** bezeichnet wird und dafür verantwortlich ist, alle unabhängig voneinander assemblierten Maschinenprogramme zusammenzufügen.

Dazu benötigt der Binder drei Schritte:

1. Ablegen der Code- und Datenmodule symbolisch in den Hauptspeicher.
2. Bestimmung der Adressen der Marken für Daten und Befehle.
3. Anpassen der internen und externen Referenzen.

**Binder (linker)** Ein Systemprogramm, das unabhängig voneinander assemblierte Maschinenprogramme zusammenfügt und alle nicht definierten Marken in einer ausführbaren Programmdatei auflöst.

Der Binder löst mithilfe der Relocation Information und der Symboltabelle in jedem Objektmodul alle nicht definierten Marken auf. Referenzen dieser Art kommen in Verzweigungen, unbedingten Sprüngen und Datenadressen vor. Die Aufgabe dieses Programms gleicht somit der eines Editors: Es findet die alten Adressen und ersetzt diese gegen die neuen Adressen. Im Englischen enthält der Name dieses Programms auch einen Hinweis auf diese Aufgabe: „*link editor*“. Der Einsatz eines Binders ist sinnvoll, da das Anpassen von Code viel schneller vonstatten geht als das erneute Kompilieren und Assemblieren von Code.

Wenn alle externen Referenzen aufgelöst sind, legt der Binder als Nächstes die Speicherpositionen für die einzelnen Module fest. Zur Erinnerung sei auf Abbildung 2.7 auf Seite 72 verwiesen, in der nach der MIPS-Konvention die Speicherzuteilung von Programmen und Daten dargestellt ist. Da die Dateien unabhängig voneinander assembliert werden, kann der Assembler nicht wissen, an welcher Stelle sich die Befehle und Daten eines Moduls relativ zu den anderen Modulen befinden werden. Wenn der Binder ein Modul in den Hauptspeicher ablegt, müssen alle *absoluten* Referenzen, d.h.

**Ausführbare Programmdatei (executable file)** Ein funktionsfähiges Programm im Format einer Objektdatei, das keine nicht aufgelösten Referenzen, Relocation Information, Symboltabellen oder Debug-Informationen enthält.

Speicheradressen, die nicht relativ zu einem Register angegeben sind, *reloziert* werden, um so die tatsächliche Position anzugeben.

Der Binder erstellt eine **ausführbare Programmdatei (executable file)**, die auf einem Computer ausgeführt werden kann. Diese Datei hat in der Regel das Format einer Objektdatei, enthält jedoch jedoch keine nicht aufgelösten Referenzen. Es gibt aber auch teilweise gebundene Dateien wie z.B. Bibliotheksroutine, die noch nicht aufgelöste Adressen enthalten, und somit Objektdateien sind.

## BEISPIEL

### Objektdateien binden

Binden Sie die beiden folgenden Objektdateien. Geben Sie die aktualisierten Adressen der ersten Befehle der endgültigen ausführbaren Programmdatei an. Wegen der besseren Lesbarkeit sind die Befehle in Assemblersprache dargestellt. In Wirklichkeit bestehen die Befehle aus Zahlen.

In den Objektdateien sind die Adressen und Symbole, die beim Binden aktualisiert werden müssen, hervorgehoben: die Befehle, die auf die Adressen der Prozeduren A und B verweisen, und die Befehle, die auf die Adressen der Datenwörter X und Y verweisen.

Objektdatei-Header			
	Name	Prozedur A	
	Textgröße	100 <sub>H</sub>	
	Datengröße	20 <sub>H</sub>	
Textsegment	Adresse	Befehl	
	0	lw\$0, 0(\$gp)	
	4	jal 0	
	...	...	
Datensegment	0	(X)	
	...	...	
Relocation Information	Adresse	Befehlstyp	Abhängigkeit
	0	lw	X
	4	jal	B
Symboltabelle	Marke	Adresse	
	X	—	
	B	—	
Objektdatei-Header			
	Name	Prozedur B	
	Textgröße	200 <sub>H</sub>	
	Datengröße	30 <sub>H</sub>	
Textsegment	Adresse	Befehl	
	0	sw\$al, 0(\$gp)	
	4	jal 0	
	...	...	
Datensegment	0	(Y)	
	...	...	
Relocation Information	Adresse	Befehlstyp	Abhängigkeit
	0	sw	Y
	4	jal	A
Symboltabelle	Marke	Adresse	
	Y	—	
	A	—	

Prozedur A benötigt die Adresse der Variablen mit der Bezeichnung X für den *Load*-Befehl und die Adresse der Prozedur B für den *jal*-Befehl. Für Prozedur B ist die Adresse der Variablen mit der Bezeichnung Y für den *sw*-Befehl und die Adresse der Prozedur A für den *jal*-Befehl zu bestimmen.

Der Abbildung 2.7 auf Seite 72 können wir entnehmen, dass das Textsegment bei Adresse  $40\ 0000_H$  und das Datensegment bei Adresse  $1000\ 0000_H$  beginnt. Der Text von Prozedur A wird an der ersten Adresse und die Daten an der zweiten Adresse abgelegt. Der Header für die Objektdatei der Prozedur A gibt die Länge seines Textes mit  $100_H$  Byte und die seiner Daten mit  $20_H$  Byte an. Somit ist die Startadresse für den Text von Prozedur B bei  $40\ 0100_H$ , und die Daten beginnen bei  $1000\ 0020_H$ .

## ANTWORT

Header der ausführbaren Programmdatei		
	Textgröße	$300_H$
	Datengröße	$50_H$
Textsegment	Adresse	Befehl
	$0040\ 0000_H$	<i>lw\$a0, 8000<sub>H</sub>(\$gp)</i>
	$0040\ 0004_H$	<i>jal 40 0100<sub>H</sub></i>
	...	...
	$0040\ 0100_H$	<i>sw\$al, 8020<sub>H</sub>(\$gp)</i>
	$0040\ 0104_H$	<i>jal 40 0000<sub>H</sub></i>
	...	...
Datensegment	Adresse	
	$1000\ 0000_H$	(X)
	...	...
	$1000\ 0020_H$	(Y)
	...	...

Nun aktualisiert der Binder die Adressfelder der Befehle. Das Format der zu ersetzenden Adresse entnimmt er dem Feld für den Befehlstyp. In unserem Beispiel gibt es zwei Typen:

1. Die *jal*-Befehle sind wegen ihrer pseudodirekten Adressierung einfach. In das Adressfeld des *jal*-Befehls bei Adresse  $40\ 0004_H$  wird die Adresse  $40\ 0100_H$  (die Adresse von Prozedur B) geschrieben, und das Adressfeld des *jal*-Befehls bei Adresse  $40\ 0104_H$  erhält die Adresse  $40\ 0000_H$  (die Adresse von Prozedur A).
2. Die Load- und Store-Adressen sind schwieriger, da diese relativ zu einem Basisregister angegeben werden. In diesem Beispiel wird das globale Zeigerregister als Basisregister verwendet. Nach Abbildung 2.7 wird  $\$gp$  mit  $1000\ 8000_H$  initialisiert. Um die Adresse  $1000\ 0000_H$  (die Adresse des Wortes X) zu erhalten, setzen wir  $8000_H$  in das Adressfeld von *lw* bei Adresse  $40\ 0000_H$ . In Kapitel 3 wird die Arithmetik mit 16-Bit-Zahlen in Zweierkomplement-Darstellung erläutert. Sie ist dafür verantwortlich, dass  $8000_H$  im Adressfeld  $1000\ 0000_H$  als Adresse ergibt. Entsprechend erhält man mit  $8020_H$  im Adressfeld des *sw*-Befehls bei Adresse  $40\ 0100_H$  die Adresse  $1000\ 0020_H$  (die Adresse des Wortes Y).

## Lader

Die ausführbare Programmdatei befindet sich nun auf der Festplatte, das Betriebssystem liest sie in den Hauptspeicher ein und startet das Programm. Bei UNIX-Systemen werden dabei folgende Schritte ausgeführt:

1. Lesen des Headers der ausführbaren Programmdatei, um die Größe der Text- und Datensegmente zu ermitteln.
2. Festlegen eines ausreichend großen Adressbereichs für den Text und die Daten.
3. Kopieren der Befehle und Daten aus der ausführbaren Programmdatei in den Hauptspeicher.
4. Kopieren der Parameter (sofern vorhanden) für das Hauptprogramm auf den Keller.
5. Initialisieren der Maschinenregister und Setzen des Kellerzeigers auf die erste freie Position.
6. Verzweigen zu einer Startroutine, die die Parameter in die Argumentregister kopiert und die Hauptroutine des Programms aufruft. Beim Rücksprung aus der Hauptroutine beendet die Startroutine das Programm mit dem Systemaufruf `exit`.

**Lader (*loader*)** Ein Systemprogramm, das ein Objektprogramm in den Hauptspeicher lädt, damit es ausgeführt werden kann.



In den Abschnitten A.3 und A.4 in **Appendix A** werden Binder und Lader (*loader*) ausführlicher beschrieben.

## Dynamisch gebundene Bibliotheken (DLLs, Dynamically Linked Libraries)

Im ersten Teil dieses Abschnitts wird die herkömmliche Vorgehensweise beschrieben, bei der Bibliotheken vor dem Ausführen des Programms gebunden werden. Dieser statische Ansatz bietet die schnellste Möglichkeit, Bibliotheks Routinen aufzurufen, er bringt jedoch auch einige Nachteile mit sich:

- Die Bibliotheks Routinen werden Teil des ausführbaren Codes. Wenn eine neue Version der Bibliothek freigegeben wird, mit der Fehler behoben oder neue Hardwareeinheiten unterstützt werden, verwendet das statisch gebundene Programm weiterhin die alte Version.
- Es wird die ganze Bibliothek geladen, auch wenn die ganze Bibliothek nicht benötigt wird, wenn das Programm ausgeführt wird. Die Bibliothek kann im Verhältnis zum Programm sehr groß sein. So umfasst die C-Standardbibliothek beispielsweise 2,5 MB.

Diese Nachteile führten zur Entwicklung dynamisch gebundener Bibliotheken (DLLs, Dynamically Linked Libraries), bei denen die Bibliotheks Routinen erst gebunden und geladen werden, wenn das Programm ausgeführt wird. Sowohl die Programm- als auch die Bibliotheks Routinen enthalten zusätzliche Informationen zur Position von nicht lokalen Prozeduren sowie zu deren Namen. Bei der ersten Version von DLLs führte der Lader einen dynamischen Binder aus, der mithilfe der zusätzlichen Informationen in der Datei die entsprechenden Bibliotheken gefunden und alle externen Referenzen aktualisiert hat.

Diese erste Version von DLLs hatte jedoch den Nachteil, dass nach wie vor alle möglicherweise benötigten Routinen der Bibliothek gebunden werden, und nicht nur die, die während der Programmausführung wirklich aufgerufen werden. Diese Beobachtung führte zur DLL-Version mit dynamischer Prozedurbindung (lazy procedure linkage), bei der die einzelnen Routinen nur *nach Aufruf* gebunden werden.

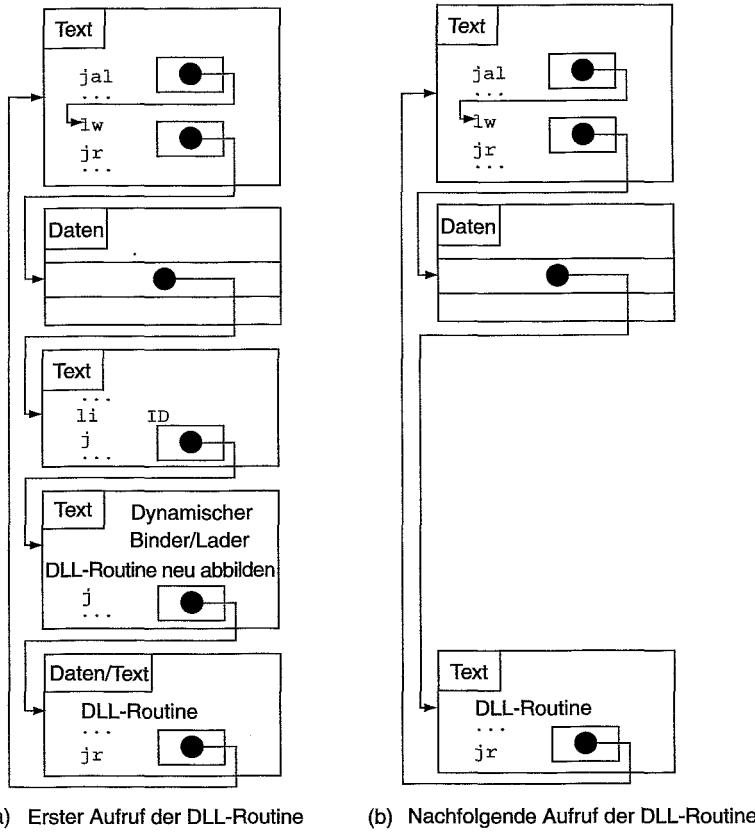
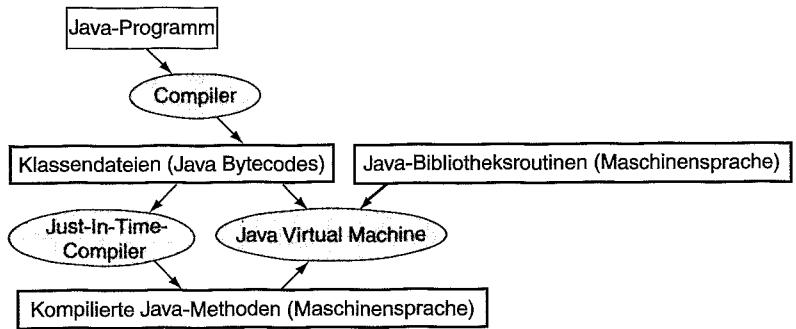


Abb. 2.11 DLL (*Dynamically Linked Library*) mittels dynamischer Prozedurbindung (*lazy procedure linkage*). (a) Schritte für den ersten Aufruf der DLL-Routine. (b) Die Schritte zum Suchen, Neuabbilden und Binden der Routine werden bei nachfolgenden Aufrufen übersprungen. Wie wir in Kapitel 7 sehen werden, kann das Betriebssystem verhindern, dass die gewünschte Routine kopiert werden muss, indem es diese mithilfe der virtuellen Speicherverwaltung neu abbildet.

Wie in vielen Fällen beruht der Trick hierbei auf Indirektion. In Abbildung 2.11 ist der Ablauf dargestellt. Zunächst rufen die nicht lokalen Routinen eine Menge von Platzhalterroutinen am Ende des Programms auf, die jeweils einen Eintrag für jede nicht lokale Routine enthalten. In jedem dieser Platzhalter steht ein indirekter Sprung.

Wenn die Bibliotheksroutine zum ersten Mal aufgerufen wird, verzweigt das Programm zu dem Platzhalter und folgt dem indirekten Sprung. Dieser zeigt auf den Programmabschnitt, in dem zur Identifikation der gewünschten Bibliotheksroutine eine Nummer in einem Register abgelegt und dann zum dynamischen Binde- und Laderprogramm gesprungen wird. Das Binde- und Laderprogramm findet die gewünschte Routine, bildet diese ab und ändert die Adresse an der Stelle des indirekten Sprungs, so dass dieser auf eben diese Routine zeigt. Anschließend springt das Programm zu dieser Routine. Wenn die Routine ausgeführt ist, kehrt das Programm an die ursprünglich aufrufende Instanz zurück. Bei nachfolgenden Aufrufen erfolgt der indirekte Sprung zu der Routine ohne die zusätzlichen Zwischenschritte.

Zusammenfassend sei angemerkt, dass mit DLLs zusätzlicher Speicherplatz für die zum dynamischen Binden notwendigen Informationen erforderlich ist. Dafür müssen nicht ganze Bibliotheken kopiert oder gebunden werden. Bei DLLs kostet der erste Aufruf einer Routine einen erheblichen Aufwand, danach jedoch nur noch einen indirekten Sprung. Der Rücksprung aus einer Bibliothek erfordert keinen zusätzlichen



**Abb. 2.12 Eine Übersetzungshierarchie für Java.** Ein Java-Programm wird zunächst in eine binäre Version von Java-Bytecodes kompiliert, wobei alle Adressen vom Compiler definiert werden. Danach kann das Java-Programm auf dem Interpreter ausgeführt werden. Dieser Interpreter wird als Java Virtual Machine (JVM) bezeichnet. Die JVM verweist auf die gewünschten Methoden in der Java-Bibliothek, während das Programm ausgeführt wird. Um eine bessere Leistung zu erzielen, kann die JVM den Just-in-Time-Compiler (JIT) aufrufen, der wahlweise Methoden in die Maschinensprache der Maschine übersetzt, auf der die JVM ausgeführt wird.

Aufwand. Bei Windows von Microsoft werden dynamische DLLs ausgiebig genutzt und auch bei UNIX-Systemen werden Programme heute üblicherweise mithilfe von DLLs ausgeführt.

## Starten eines Java-Programms

Die Diskussion im vorangegangenen Abschnitt behandelt den traditionellen Ansatz für die Übersetzung eines Programms, wobei die Betonung auf der schnellen Ausführung eines Programms für eine spezielle Befehlssatzarchitektur oder einer speziellen Implementierung dieser Architektur liegt. So ist es zwar möglich, Java-Programme so wie C-Programme auszuführen, jedoch wurde Java mit anderen Zielsetzungen entwickelt. Eines der Ziele bei der Entwicklung von Java bestand darin, Programme auf jedem beliebigen Computer schnell und sicher ausführen zu können, auch wenn dies auf Kosten der Ausführungszeit geschieht.

Abbildung 2.12 zeigt die typischen Übersetzungs- und Ausführungsschritte für Java. In Java wird nicht in die Assemblersprache für einen Zielrechner übersetzt. Vielmehr werden in Java Befehle generiert, die einfach zu interpretieren sind: den **Java-Bytecode**. Dieser Befehlssatz ist der Java-Sprache sehr ähnlich, weshalb dieser Übersetzungsschritt einfach ist. Es werden nahezu keine Optimierungen durchgeführt. Wie der C-Compiler so überprüft auch der Java-Compiler die Datentypen und generiert jeweils die für die einzelnen Typen entsprechende Operation. Java-Programme werden in der Binärversion dieser Bytecodes verbreitet.

Ein Software-Interpreter, der als **Java Virtual Machine (JVM)** bezeichnet wird, kann Java-Bytecodes ausführen. Ein Interpreter ist ein Programm, das eine Befehlssatzarchitektur simuliert. Der in diesem Buch verwendete MIPS-Simulator ist beispielsweise ein Interpreter. Ein eigener Assemblerschritt ist hier nicht erforderlich, da die Übersetzung so einfach ist, dass entweder der Compiler die Adressen einfügt oder die JVM diese zur Laufzeit ermittelt.

Die Interpretation hat den Vorteil der Portierbarkeit. Die Verfügbarkeit der Software der Java Virtual Machines bedeutete, dass die meisten schon kurz nach der Ankündigung von Java bereits Java-Programme schreiben und ausführen konnten. Heute finden wir Java Virtual Machines in Millionen von Geräten, angefangen von Mobiltelefonen bis hin zu Internet-Browsern.

**Java-Bytecode** Befehl aus einem Befehlssatz, der für die Interpretation von Java-Programmen entwickelt worden ist.

**Java Virtual Machine (JVM)**  
Das Programm, das Java-Bytecodes interpretiert.

Der Nachteil der Interpretation ist die schwache Leistungsfähigkeit. Aufgrund der unglaublichen Fortschritte hinsichtlich der Leistungsfähigkeit in den 80er- und 90er-Jahren des letzten Jahrhunderts ist die Interpretation für viele wichtige Anwendungen eine interessante Alternative. Aber der Faktor 10, um den die in herkömmlicher Weise übersetzten C-Programme schneller sind, macht Java für bestimmte Anwendungen wenig attraktiv.

Um auf der einen Seite die Portierbarkeit zu gewährleisten und gleichzeitig die Ausführungsgeschwindigkeit zu steigern, ging es bei der Java-Entwicklung in einem nächsten Schritt darum, Compiler zu konzipieren, die übersetzten, während das Programm ausgeführt wurde. Diese **Just-in-Time-Compiler (JIT-Compiler)**, auch *dynamische Übersetzer* genannt, erstellen ein Profil des Programms, das gerade ausgeführt wird, um so die relevanten Methoden zu finden und diese in den Befehlssatz des Computers zu übersetzen, auf dem die Virtual Machine läuft. Der kompilierte Teil wird für das nächste Mal gespeichert, wenn das Programm auszuführen ist. Bei der wiederholten Ausführung ist es damit schneller. Mit der Zeit entwickelt sich ein Gleichgewicht zwischen Interpretation und Übersetzung, so dass häufig ausgeführte Java-Programme kaum noch Einbußen aufgrund der Interpretation aufweisen.

Da Rechner immer schneller werden und damit auch Compiler immer aufwendigere Aufgaben erledigen können und da Forscher bessere Techniken entwickeln, um Java zur Laufzeit zu kompilieren, wird der Leistungsunterschied zwischen Java und C oder C++ immer geringer. In Abschnitt 2.14 wird die Implementierung von Java, Java-Bytecodes, JVM und JIT-Compiler ausführlicher beschrieben.

**Just-in-Time-Compiler (JIT-Compiler)** Die Bezeichnung, die sich für einen Compiler eingebürgert hat, der zur Laufzeit die interpretierten Codesegmente in Ziel-Code für die Maschine übersetzt.



Was meinen Sie, welche Vorteile eines Interpreters gegenüber einem Übersetzer stehen für die Entwickler von Java im Vordergrund?

1. Leichteres Schreiben eines Interpreters
2. Bessere Fehlermeldungen
3. Kleinerer Objektcode
4. Plattformunabhängigkeit

## 2.11 Optimierung durch den Compiler

Der Compiler hat einen entscheidenden Einfluss auf die Leistung eines Rechners. Für das Verständnis des Leistungsverhaltens ist es daher von entscheidender Bedeutung, die moderne Compilertechnik zu kennen. Ziel dieses Abschnitts ist einen kurzen Überblick über Optimierungen zu geben, die ein Compiler zur Verbesserung der Leistung einsetzt. Im folgenden Abschnitt wird der innere Aufbau eines Computers vorgestellt. Den Ausgangspunkt bildet die in Abbildung 2.13 dargestellte Struktur aktueller Compiler. Die Optimierungen werden gemäß der Reihenfolge der Phasen dieser Struktur beschrieben.

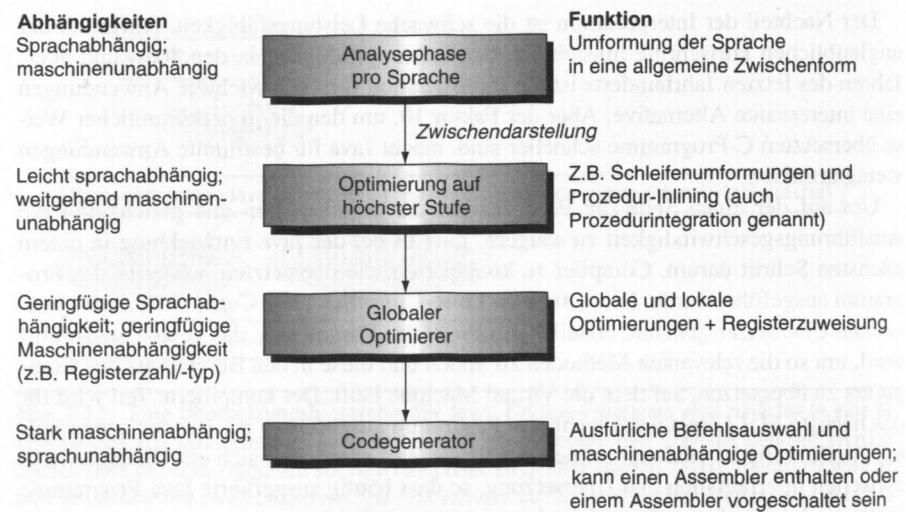


Abb. 2.13 Die Struktur eines modernen optimierenden Compilers besteht aus mehreren Schritten oder Phasen. Jeder Schritt wird logisch gesehen abgeschlossen, bevor ein neuer begonnen wird. In der Praxis kann eine Prozedur im Rahmen mehrerer Schritte gleichzeitig bearbeitet werden, so dass sich die Schritte im Prinzip überschneiden.

## Optimierungen auf der höchsten Stufe

Optimierungen auf der höchsten Stufe sind Transformationen, die auf einer Ebene nahe der Quellsprache durchgeführt werden.

Die wohl am häufigsten angewendete Art von Transformationen auf dieser Stufe ist das *Prozedur-Inlining*, auch als *Prozedurerersetzung* oder *Prozedurintegration* bezeichnet. Dabei wird ein Funktionsaufruf durch den Funktionsrumpf und die Parameter der Prozedur durch die Argumente der aufrufenden Prozedur ersetzt.

Darüber hinaus zählen auch Schleifentransformationen bzw. Schleifenumformungen zu den Optimierungen auf dieser Stufe. Mit Schleifentransformationen kann der Aufwand für Schleifen reduziert, der Speicherzugriff verbessert und die Hardware effizienter genutzt werden. Beispielsweise bietet sich bei Schleifen mit vielen Iterationen, wie sie häufig bei *For*-Anweisung auftreten, die Optimierung des *Schleifenabrollens (loop unrolling)* an.

Beim Schleifenabrollen wird jeweils eine Schleife betrachtet, deren Schleifenkörper mehrmals repliziert wird. Dies hat zur Folge, dass die transformierte Schleife bei der Ausführung weniger oft durchlaufen wird. Mit dem Schleifenabrollen wird der Schleifen-Overhead reduziert. Außerdem bietet dieses Verfahren Möglichkeiten für viele weitere Optimierungen. Daneben zählen auch komplexe Schleifentransformationen wie das Vertauschen von geschachtelten Schleifen und das Blocking von Schleifen zu den Transformationen auf höchster Stufe. Beispiele hierzu finden Sie in Kapitel 7.

## Lokale und globale Optimierungen

Während der Phase der lokalen und globalen Optimierung werden drei Arten von Optimierungen durchgeführt:

1. *Lokale Optimierung* erfolgt innerhalb eines Basisblocks. Ein lokaler Optimierungsschritt wird häufig zum „Aufräumen“ des Codes jeweils vor und nach einer globalen Optimierung durchgeführt.

**Schleifenabrollen (*loop unrolling*)** Eine Technik zur Verbesserung der Leistung bei der Abarbeitung von Schleifen, in denen auf Felder zugegriffen wird, bei der mehrere Kopien des Schleifenrumpfs erstellt und Befehle aus unterschiedlichen Iterationen miteinander angeordnet werden.

2. *Globale Optimierung* erfolgt über mehrere Basisblöcke hinweg. Ein Beispiel hierfür finden Sie weiter unten.
3. Die globale *Registerzuteilung* weist den Variablen Register für Bereiche des Codes zu. Die Registerzuteilung ist für moderne Prozessoren von entscheidender Bedeutung, um eine hohe Leistung erzielen zu können.

Einige Optimierungen werden sowohl lokal als auch global durchgeführt. Beispiele hierfür sind die Eliminierung gemeinsamer Teilausdrücke (CSE, Common Subexpression Elimination), Konstantenpropagation, Copy Propagation, Dead Store Elimination und Strength Reduction. Im Folgenden werden wir einige einfache Beispiele für diese Optimierungsmöglichkeiten etwas näher betrachten.

Bei der *Eliminierung gemeinsamer Teilausdrücke* wird nach mehreren Instanzen eines Ausdrucks gesucht und die zweite Instanz durch eine Referenz zur ersten ersetzt. Gegeben sei beispielsweise ein Codesegment zum Addieren von 4 zu einem Feldelement:

```
x[i] = x[i] + 4
```

Die Adressberechnung für `x[i]` tritt zweimal auf und die erste Berechnung ist mit der zweiten identisch, da sich weder die Startadresse von `x` noch der Wert von `i` ändert. So mit kann die Berechnung wieder verwendet werden. Betrachten wir den Zwischencode für dieses Codefragment, denn hierauf können noch weitere Optimierungen vorgenommen werden. Auf der linken Seite befindet sich der nicht optimierte Zwischencode. Rechts ist der Code aufgeführt, bei der mithilfe der *Eliminierung gemeinsamer Teilausdrücke* die zweite Adressberechnung durch die erste ersetzt worden ist. Es ist zu beachten, dass die Registerzuteilung noch nicht stattgefunden hat, weshalb der Compiler hier noch virtuelle Registernummern wie `R100` verwendet.

<pre># x[i] + 4 li R100,x lw R101,i mult R102,R101,4 add R103,R100,R102 lw R104,0(R103) # Wert von x[i] ist in R104 add R105,R104,4 # x[i] = li R106,x lw R107,i mult R108,R107,4 add R109,R106,R107 sw R105,0(R109)</pre>	<pre># x[i] + 4 li R100,x lw R101,i mult R102,R101,4 add R103,R100,R102 lw R104,0(R103) # Wert von x[i] ist in R104 add R105,R104,4 # x[i] = sw R105,0(R103)</pre>
--	--

Wenn diese Optimierung über zwei Basisblöcke hinweg möglich wäre, hätte man den Fall einer *globaler Eliminierung gemeinsamer Teilausdrücke*.

Betrachten wir nun einige der anderen Optimierungsmöglichkeiten etwas näher:

- Die *Strength Reduction* ersetzt komplexe Operationen durch einfachere. Sie kann auf dieses Codesegment angewendet werden und den `mult`-Befehl durch eine Schiebeoperation nach links ersetzen.
- Die *Konstantenpropagation* und die mit ihr verwandte *Konstantenfaltung* suchen im Code nach Ausdrücken, deren Wert bei jeder Ausführung des Programms immer gleich ist. Solche konstanten Ausdrücke können zur Übersetzungszeit ausgewertet und durch ihre Werte ersetzt werden.

- Die *Copy Propagation* verbreitet Werte, bei denen es sich um einfache Kopien handelt. So müssen Werte nicht immer wieder neu geladen werden. Außerdem werden dadurch weitere Optimierungen wie die Eliminierung gemeinsamer Teilausdrücke möglich.
- Die *Dead Store Elimination* sucht nach Speicherbefehlen für Werte, die nicht mehr verwendet werden, und entfernt diese. Eine verwandte Optimierung ist die Elimination von „totem“ Code (*Dead Code Elimination*), die bei keiner Programmausführung verwendete Codeteile findet, also Codeteile, die keine Auswirkungen auf das Endergebnis des Programms haben, und entfernt diese. Mit der häufigen Verwendung von Makros, Vorlagen (*templates*) und ähnlichen Techniken, die speziell für die Wiederverwendung von Code in höheren Programmiersprachen konzipiert sind, tritt „toter“ Code überraschend häufig auf.

## Grundlegendes zur Leistungs- fähigkeit von Programmen

Programmierer, die sich insbesondere bei Echtzeit- oder eingebetteten Anwendungen besonders auf das Leistungsverhalten von kritischen Schleifen konzentrieren, wundern sich angesichts des von einem Compiler generierten Assembler-Codes häufig darüber, warum der Compiler eine bestimmte globale Optimierung nicht durchgeführt oder eine Variable in einer Schleife einem Register nicht zugeordnet hat. Der Grund hierfür liegt vielfach darin, dass der Compiler gezwungen ist, konservative Annahmen über das Laufzeitverhalten zu treffen. Die Möglichkeiten, den Code zu verbessern, können für den Programmierer ganz offensichtlich sein. Aber der Programmierer verfügt dann über Wissen, das dem Compiler nicht zugänglich ist. So ist dem Programmierer beispielsweise bekannt, wenn zwischen zwei Zeigern kein Aliasing auftritt, d.h. der Zugriff auf dieselbe Speicherzelle ausgeschlossen ist, oder wenn ein Funktionsaufruf keine Nebeneffekte mit sich bringt. Dabei kann der Compiler sehr wohl in der Lage sein, die Transformation mit etwas Unterstützung durchzuführen. Dazu muss jedoch die konservative Annahme, von der der Compiler zunächst ausgeht, ausgeschlossen werden. Diese Erkenntnis macht eine wichtige Beobachtung deutlich: Programmierer, die mithilfe von Zeigern versuchen, das Leistungsverhalten beim Zugriff auf Variablen zu verbessern, und dies insbesondere mithilfe von Zeigern auf Werte im Keller versuchen, die wie Variablen oder Feldelemente Namen haben, verhindern viele Compiler-optimierungen. Das führt letztlich dazu, dass der Zeigercode auf niedererer Ebene nicht besser oder sogar schlechter ausgeführt wird als der vom Compiler optimierte Code auf höherer Ebene.

Compiler müssen von *konservativen* Annahmen ausgehen. Die wichtigste Aufgabe eines Compilers besteht darin, korrekten Code zu generieren. Erst an zweiter Stelle folgt die Aufgabe, schnellen Code zu generieren, wenngleich auch andere Faktoren wie die Codegröße ebenfalls eine wichtige Rolle spielen können. Code, der zwar schnell, aber für eine mögliche Kombination von Eingaben, fehlerhaft ist, ist einfach falsch. Wenn wir also sagen, ein Compiler ist „konservativ“, meinen wir, dass er eine Optimierung nur durchführt, wenn er mit 100%iger Sicherheit weiß, dass sich der Code unabhängig von der Eingabe immer wie vom Benutzer geschrieben verhält. Da die meisten Compiler zu einem Zeitpunkt jeweils eine Funktion oder Prozedur übersetzen und optimieren, gehen die meisten Compiler, insbesondere bei Optimierungen auf niedrigeren Stufen, im Hinblick auf Funktionsaufrufe und auf eigene Parameter vom schlechtesten Fall aus.

## Globale Codeoptimierungen

Mit vielen globalen Codeoptimierungen wird dasselbe Ziel wie mit den lokalen Optimierungen verfolgt, einschließlich der Eliminierung gemeinsamer Teilausdrücke, Konstantenpropagation, Copy Propagation sowie Dead Store und Dead Code Elimination.

Es gibt zwei weitere wichtige globale Optimierungen: die Codeverschiebung (Code Motion) und die Eliminierung der Induktionsvariablen (Induction Variable Elimination). Bei beiden Optimierungen handelt es sich um Schleifenoptimierungen, d.h. beide Fälle betrachten den Code in Schleifen. Bei der *Codeverschiebung* wird schleifeninvariante Code gefunden. Ein solcher Programmabschnitt berechnet in jeder Iteration denselben Wert. Es genügt, wenn dies nur einmal außerhalb der Schleife erfolgt. Die *Eliminierung der Induktionsvariablen* ist eine Kombination aus verschiedenen Transformationen, mit deren Hilfe der durch das Indizieren von Feldern entstehende Aufwand reduziert wird, indem anstelle der Feldindizierung Zeigerzugriffe verwendet werden. Statt die Eliminierung von Induktionsvariablen hier ausführlicher zu betrachten, verweisen wir den Leser auf Abschnitt 2.15. Dort werden Feldindizierung und Zeiger miteinander verglichen. Bei den meisten Schleifen kann die Transformation des Feldcodes in den Zeigercode mithilfe eines modernen optimierenden Compilers durchgeführt werden.

## Optimierung im Überblick

In Tabelle 2.19 sind Beispiele für gängige Optimierungen aufgeführt. In der letzten Spalte ist angegeben, bei welcher Einstellung beim gcc-Compiler die Optimierung durchgeführt wird. Die einfacheren lokalen und prozessorabhängigen Optimierungen lassen sich nicht immer ganz leicht von den im Codegenerator durchgeföhrten Transformationen unterscheiden, und einige Optimierungen werden mehrere Male durchgeführt. Das ist insbesondere bei lokalen Optimierungen der Fall, die vor und nach globalen Optimierungen sowie während der Codegenerierung durchgeführt werden können.

Programme für Arbeitsplatzrechner- und Serveranwendungen werden heute ebenso wie die meisten Programme für eingebettete Anwendungen überwiegend in höheren Programmiersprachen geschrieben. Da es sich bei den meisten ausgeführten Befehlen um die Ausgabe eines Compilers handelt, bedeutet diese Entwicklung, dass eine Befehlssatzarchitektur im Prinzip das Ziel für den Compiler darstellt. Das Gesetz von Moore verleitet dazu, in den Befehlssatz komplexe Operationen zu integrieren. Das Problem dabei ist, dass diese möglicherweise nicht exakt dem entsprechen, was der Compiler generieren muss, oder dass diese so allgemein sind, dass sie nicht schnell sind. Stellen Sie sich beispielsweise spezielle Schleifenbefehle vor, wie sie in einigen Rechnern vorkommen. Nehmen wir nun an, dass der Compiler nicht um eins dekrementiert, sondern stattdessen um vier inkrementieren möchte, oder dass er nicht aufgrund der Bedingung ungleich null verzweigen möchte, sondern wenn der Index kleiner als oder gleich dem Grenzwert ist. Der Schleifenbefehl kann dann eventuell ungeeignet sein. Angesichts dieser Einschränkungen mag der Architekt eines Befehlssatzes versucht sein, die Operation zu verallgemeinern, indem er einen weiteren Operand zur Spezifikation des Inkrements hinzufügt oder eine Option zur Auswahl der Verzweigungsbedingung vorsieht. Dann besteht jedoch die Gefahr, dass der häufig vorkommende Fall, nämlich das Inkrementieren um eins, langsamer ist als eine Folge einfacher Operationen.

## Hardware-Software-Schnittstelle

**Tab. 2.19 Wichtigste Optimierungen und Beispiele für jede Klasse.** In der dritten Spalte ist angegeben, bei welchen Optimierungsoptionen diese im gcc-Compiler auftreten. Nach Gnu heißen die drei Optimierungsstufen O1 (mittel), O2 (vollständig) und O3 (vollständig mit Integration kleiner Prozeduren).

Name der Optimierung	Erläuterung	gcc-Stufe
Höchste Stufe	Auf oder nahe der Quellsprachenebene; prozessorunabhängig	
Prozedurintegration	Ersetzen von Prozederaufruf durch Prozedurrumpf	O3
Lokal	In linearem Code	
Eliminierung gemeinsamer Teilausdrücke	Ersetzen zweier Instanzen derselben Berechnung durch eine Kopie	O1
Konstantenpropagation	Ersetzen aller Instanzen einer Variablen, denen eine Konstante zugewiesen ist, durch eine Konstante	O1
Reduzierung der Kellertiefe	Neuanordnen des Ausdrucksbaums, um die für die Ausdrucksauswertung erforderlichen Ressourcen zu minimieren	O1
Global	Über einen Sprung hinweg	
Globale Eliminierung gemeinsamer Teilausdrücke	Wie lokal, jedoch über Sprung hinweg	O2
Copy Propagation	Ersetzen aller Instanzen einer Variablen $A$ , der $X$ zugewiesen wurde (d.h., $A = X$ ) durch $X$	O2
Code Motion	Entfernen von Code aus einer Schleife, die denselben Wert bei jeder Iteration berechnet	O2
Eliminierung der Induktionsvariablen	Vereinfachen/eliminieren von Feldadressberechnungen innerhalb von Schleifen	O2
Prozessorabhängig	Hängt vom Wissen über den Prozessor ab	
Strength Reduction	Viele Beispiele; ersetzen einer Multiplikation durch eine Schiebeoperation mit einer Konstanten	O1
Pipeline-Scheduling	Umordnen von Befehlen zum Verbessern des Leistungsverhaltens der Pipeline	O1
Optimierung des Sprung-Offsets	Auswählen des kürzesten Sprung-Offsets, mit dem das Ziel erreicht wird	O1

## 2.12

# Wie arbeiten Compiler: Eine Einführung

In diesem Abschnitt finden Sie eine kurze Übersicht über die Funktionsweise eines Compilers. Sie soll dem Leser helfen, zu verstehen, wie der Compiler ein in einer höheren Programmiersprache geschriebenes Programm in Maschinbefehle übersetzt. Beachten Sie, dass der Compilerbau üblicherweise im Rahmen einer ein- bzw. zweisemestrigen Vorlesung behandelt wird und diese Einführung notwendigerweise nur an der Oberfläche bleiben kann. Den Rest dieses Abschnitts finden Sie auf der CD.



**2.13**

## Zusammenfassung am Beispiel eines Sortierprogramms in C

Wenn in Assemblersprache geschriebener Code nur in Ausschnitten dargestellt wird, besteht die Gefahr, dass Sie als Leser keine Vorstellung davon vermittelt bekommen, wie das gesamte Programm in Assemblersprache aussieht. In diesem Abschnitt werden wir den MIPS-Code von zwei in C geschriebenen Prozeduren ableiten: einer Prozedur zum Vertauschen und eine zum Sortieren von Feldelementen.

### Die Prozedur swap

Beginnen wir mit dem Code für die Prozedur swap in Abbildung 2.14. Diese Prozedur tauscht einfach die Inhalte zweier Speicherzellen aus. Beim manuellen Übersetzen von C in Assemblersprache gehen wir wie folgt vor:

1. Zuteilung von Registern an Programmvariablen.
2. Generierung des Codes für den Rumpf der Prozedur.
3. Beibehalten der Register über den Prozederaufruf hinweg.

Dieser Abschnitt beschreibt für die swap-Prozedur diese drei Schritte, wobei diese am Ende zusammengefasst werden.

### Registerzuteilung für swap

Wie auf Seite 65 bereits erwähnt, werden nach der Konvention bei MIPS die Register \$a0, \$a1, \$a2 und \$a3 zum Übergeben von Parametern verwendet. Da die swap-Prozedur nur die beiden Parameter v und k hat, befinden sich diese in den Registern \$a0 und \$a1. Die einzige weitere Variable ist die Variable temp, die wir dem Register \$t0 zuordnen, da swap eine Blattprozedur ist (siehe Seite 68). Diese Registerzuteilung entspricht den Variablen Deklarationen im ersten Teil der swap-Prozedur in Abbildung 2.14.

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

---

**Abb. 2.14 Eine C-Prozedur, die die Inhalte zweier Speicherzellen vertauscht. Im nächsten Abschnitt wird diese Prozedur in einem Beispiel zum Sortieren verwendet.**

### Code für den Rumpf der swap-Prozedur

Die restlichen Zeilen des C-Codes in der swap-Prozedur lauten:

```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

Beachten Sie, dass sich die Speicheradressen bei MIPS auf die *Byte*adresse beziehen, wodurch die Wörter jeweils um 4 Byte voneinander entfernt sind.

Daher muss der Index  $k$  vor der Addition mit 4 multipliziert werden. Beim Assembler-Programmieren wird häufig vergessen, dass sich sequenzielle Wortadressen nicht um 1, sondern um 4 unterscheiden. Der erste Schritt besteht also darin, die Adresse von  $v[k]$  durch Multiplikation von  $k$  mit 4 zu ermitteln:

```
sll $t1, $a1, 2    # Reg. $t1 = k * 4
add $t1, $a0, $t1 # Reg. $t1 = v + (k * 4)
                    # Reg. $t1 enthält die Adresse von v[k]
```

Nun laden wir  $v[k]$  mithilfe von  $\$t1$ , und anschließend  $v[k+1]$ , indem wir zu  $\$t1$  4 addieren:

```
lw $t0, 0($t1) # temp. Reg. $t0 = v[k]
lw $t2, 4($t1) # Reg. $t2 = v[k + 1]
                    # referenziert das nächste Element von v
```

Als Nächstes speichern wir  $\$t0$  und  $\$t2$  an den vertauschten Adressen:

```
sw $t2, 0($t1) # v[k] = Reg. $t2
sw $t0, 4($t1) # v[k+1] = temp. Reg. $t0
```

Nun haben wir Register zugewiesen und den Code so geschrieben, dass die Operationen der Prozedur ausgeführt werden. Was noch fehlt, ist der Code zum Beibehalten der zu rettenden Register, die in dieser swap-Prozedur verwendet werden. Da wir allerdings in dieser Blattprozedur keine zu rettenden Register verwenden, gibt es nichts beizubehalten.

### Die vollständige Prozedur swap

Wir haben die ganze Routine mit der Prozedurmarke und dem Rücksprung vorbereitet. Damit die Routine besser nachvollziehbar wird, sind in Tabelle 2.20 die einzelnen Codeblöcke mit ihren jeweiligen Aufgaben in der Prozedur aufgeführt.

### Die Prozedur sort

Damit Sie die Exaktheit der Assembler-Programmierung auch wirklich zu schätzen lernen, geben wir Ihnen ein zweites, ausführlicheres Beispiel. In diesem Beispiel erstellen wir eine Routine, in der die Prozedur swap aufgerufen wird. Dieses Programm

Tab. 2.20 MIPS-Assemblercode der Prozedur swap in Abbildung 2.14.

Prozedurrumpf
<pre>swap: sll \$t1, \$a1, 2    # Reg. \$t1 = k*4       add \$t1, \$a0, \$t1 # Reg. \$t1 = v + (k * 4)                           # Reg. \$t1 enthält die Adresse von v[k]       lw \$t0, 0(\$t1)    # temp. Reg. \$t0 = v[k]       lw \$t2, 4(\$t1)    # temp. Reg. \$t2 = v[k + 1]                           # verweist auf nächstes Element von v       sw \$t2, 0(\$t1)    # v[k] = Reg. \$t2       sw \$t0, 4(\$t1)    # v[k+1] = temp. Reg. \$t0</pre>
Prozedurrücksprung
<pre>jr     \$ra      # springe zurück zur aufrufenden Prozedur</pre>

```

void sort(int v[], int n)
{
    int i, j;
    for (i = 0; i < n; i += 1) {
        for (j = i - 1; j >= 0 && v[j] > v[j + 1]; j -= 1)
            swap (v,j);
    }
}

```

Abb. 2.15 Eine C-Prozedur zum Sortieren der Elemente des Felds v.

sortiert ein Feld von Integer-Zahlen mithilfe des Sortierverfahrens Bubble Sort bzw. Exchange Sort, einem der einfachsten Sortierverfahren. In Abbildung 2.15 ist die C-Version des Programms beschrieben. Auch hier werden die einzelnen Schritte und am Ende die ganze Prozedur dargestellt.

## Registerzuteilung für sort

Die beiden Parameter v und n der Prozedur `sort` befinden sich in den Parameterregistern \$a0 und \$a1, und wir weisen Register \$s0 i und Register \$s1 j zu.

## Code für den Rumpf der Prozedur sort

Der Prozedurrumpf besteht aus zwei geschachtelten *For-Schleifen* und einem Aufruf von `swap` mit Parametern. Untersuchen wir den Code von außen nach innen.

Der erste Übersetzungsschritt beginnt mit der ersten *For-Schleife*:

```
for (i = 0; i < n; i += 1) {
```

Eine *For-Anweisung* in C besteht aus drei Teilen: Initialisierung, Schleifentest und Inkrement der Iteration. Zum Initialisieren von i mit 0, dem ersten Teil der *For-Anweisung*, ist nur ein Befehl erforderlich:

```
move $s0, $zero # i = 0
```

(Denken Sie daran, dass `move` ein Pseudobefehl ist, der dem Programmierer vom Assembler zum leichteren Programmieren in Assemblersprache bereitgestellt wird, siehe Seite 90.) Es ist außerdem auch nur ein Befehl zum Inkrementieren von i, dem letzten Teil der *For-Anweisung*, erforderlich:

```
addi $s0, $s0, 1 # i += 1
```

Die Schleife muss verlassen werden, wenn die Bedingung  $i < n$  nicht wahr ist, oder anders ausgedrückt: wenn  $i \geq n$ . Mit dem *Set-on-less-than*-Befehl wird das Register \$t0 auf 1 gesetzt, wenn  $\$s0 < \$a1$ , andernfalls auf 0. Da wir überprüfen möchten, ob  $\$s0 \geq \$a1$ , verzweigen wir, wenn Register \$t0 0 ist. Für diesen Test sind zwei Befehle erforderlich:

```
forltst: slt $t0, $s0, $a1 # Reg. $t0 = 0,
               # wenn $s0 \geq $a1 (i \geq n)
    beq $t0, $zero, exit1   # verzweige zu exit1,
```

Am Schleifenende erfolgt der Sprung zurück zum Schleifentest:

j forltst # springe zurück zum äußeren Schleifentest  
exit1:

Das Codegerüst der ersten *For*-Schleife lautet somit wie folgt:

```
move $s0, $zero          # i = 0
forltst: slt $t0, $s0, $a1 # Reg. $t0 = 0,
           # wenn $s0 ≥ $a1 (i ≥ n)
    beq $t0, $zero, exit1   # verzweige zu exit1,
                           # wenn $s0 ≥ $a1 (i ≥ n)
    ...
    (Rumpf der ersten Schleife)
    ...
    addi $s0, $s0, 1        # i += 1
    j forltst              # springe zurück
                           # zum äußeren Schleifentest
exit1:
```

Die zweite *For*-Schleife sieht in C wie folgt aus:

```
for (j = i - 1; j >= 0 && v[j] > v[j + 1]; j -= 1) {
```

Der Initialisierungsteil dieser Schleife besteht wiederum aus einem Befehl:

```
addi $s1, $s0, -1 # j = i - 1
```

Zum Dekrementieren von j am Ende der Schleife ist ebenfalls nur ein Befehl notwendig:

```
addi $s1, $s1, -1 # j -= 1
```

Der Schleifentest besteht aus zwei Teilen. Wir verlassen die Schleife, wenn eine der Bedingungen nicht zutrifft. Der erste Test muss somit die Schleife verlassen, wenn ( $j < 0$ ) nicht wahr ist:

```
for2tst: slti $t0, $s1, 0    # Reg. $t0 = 1,
           # wenn $s1 < 0 (j < 0)
    bne $t0, $zero, exit2   # verzweige zu exit2,
                           # wenn $s1 < 0 (j < 0)
```

Diese Verzweigung überspringt den Test der zweiten Bedingung. Falls nicht, ist  $j \geq 0$ .

Der zweite Test verlässt die Schleife, wenn  $v[j] > v[j + 1]$  nicht wahr ist oder wenn  $v[j] \leq v[j + 1]$ . Zuerst berechnen wir die Adresse durch Multiplikation von j mit 4 (da wir eine Byteadresse benötigen) und addieren das Ergebnis zur Basisadresse von v:

```
sll $t1, $s1, 2    # Reg. $t1 = j * 4
add $t2, $a0, $t1 # Reg. $t2 = v + (j * 4)
```

Nun laden wir v[j]:

```
lw $t3, 0($t2) # Reg. $t3 = v[j]
```

Da wir wissen, dass das zweite Element einfach das nachfolgende Wort ist, addieren wir zu der Adresse in Register \$t2 4, um v[j + 1] zu erhalten:

```
lw $t4, 4($t2) # Reg. $t4 = v[j + 1]
```

Der Test von  $v[j] = v[j + 1]$  ist derselbe wie von  $v[j + 1] \geq v[j]$ , so dass die beiden Befehle des Tests zum Verlassen wie folgt lauten:

```
slt $t0, $t4, $t3      # Reg. $t0 = 0, wenn $t4 ≥ $t3
beq $t0, $zero,exit2  # verzweige zu exit2,
                      # wenn $t4 ≥ $t3
```

Am Schleifenende wird zum Test der inneren Schleife zurückgesprungen:

```
j for2tst # springe zurück zum inneren Schleifentest
```

Wenn wir die Teile zusammenfügen, ergibt sich für die zweite *For*-Schleife folgendes Gerüst:

```
addi $s1, $s0, -1      # j = i - 1
for2tst: slti $t0, $s1, 0      # Reg. $t0 = 1,
                           # wenn $s1 < 0 (j<0)
        bne $t0, $zero,exit2 # verzweige zu exit2,
                           # wenn $s1<0 (j<0)
        sll $t1, $s1, 2      # Reg. $t1 = j * 4
        add $t2, $a0, $t1      # Reg. $t2 = v + (j * 4)
        lw $t3, 0($t2)      # Reg. $t3 = v[j]
        lw $t4, 4($t2)      # Reg. $t4 = v[j + 1]
        slt $t0, $t4, $t3      # Reg. $t0 = 0,
                           # wenn $t4 ≥ $t3
        beq $t0, $zero,exit2 # verzweige zu exit2,
                           # wenn $t4 ≥ $t3
        ...
        (Rumpf der zweiten Schleife)
        ...
addi $s1, $s1, -1      # j -= 1
j for2tst              # springe zum Test
                      # der inneren Schleife
```

exit2:

## Der Prozeduraufruf in sort

Der nächste Schritt betrifft den Rumpf der zweiten *For*-Schleife:

```
swap(v, j);
```

Die Prozedur swap aufzurufen, ist einfach:

```
jal     swap
```

## Übergabe von Parametern in sort

Schwieriger wird es, wenn wir Parameter übergeben möchten, da die Prozedur sort die Werte in den Registern \$a0 und \$a1 benötigt und die Prozedur swap ihre Parameter in genau denselben Registern erwartet. Eine Lösungsmöglichkeit besteht darin, die Parameter für die sort-Prozedur in anderen Registern weiter vorne in der Prozedur zu kopieren und die Register \$a0 und \$a1 für den Aufruf von swap zur Verfügung zu stellen. (Dieser Kopiervorgang ist schneller als das Speichern und Wiederherstellen im Keller.) Während der Prozedur kopieren wir zuerst \$a0 und \$a1 nach \$s2 und \$s3:

```
move $s2, $a0 # kopiere Parameter $a0 nach $s2
move $s3, $a1 # kopiere Parameter $a1 nach $s3
```

Anschließend übergeben wir mithilfe der folgenden zwei Befehle die Parameter an swap:

```
move $a0, $s2 # erster Parameter von swap ist v
move $a1, $s1 # zweiter Parameter von swap ist j
```

### Beibehalten von Registern in sort

Nun verbleibt noch der Code zum Retten und Wiederherstellen von Registern. Natürlich müssen wir die Rücksprungadresse in Register \$ra speichern, da sort eine Prozedur ist und selbst aufgerufen wird. Die sort-Prozedur verwendet außerdem die zu rettenden Register \$s0, \$s1, \$s2 und \$s3, so dass diese gesichert werden müssen. Der Prolog der Prozedur sort lautet wie folgt:

```
addi $sp,$sp,-20 # schaffe Platz auf dem Keller für 5 Reg.
sw $ra,16($sp)   # sichere $ra auf dem Keller
sw $s3,12($sp)   # sichere $s3 auf dem Keller
sw $s2, 8($sp)   # sichere $s2 auf dem Keller
sw $s1, 4($sp)   # sichere $s1 auf dem Keller
sw $s0, 0($sp)   # sichere $s0 auf dem Keller
```

Am Ende der Prozedur stehen die entsprechenden Befehle zum Wiederherstellen der Register und der Befehl jr für den Rücksprung.

### Die vollständige Prozedur sort

In Tabelle 2.21 fügen wir alle Teile zusammen, wobei wir sorgfältig darauf achten müssen, dass wir alle Referenzen auf die Register \$a0 und \$a1 in den *For*-Schleifen durch Referenzen auf die Register \$s2 und \$s3 ersetzen. Damit der Code besser nachvollziehbar wird, sind auch hier die einzelnen Programmabschnitte mit ihren jeweiligen Aufgaben in der Prozedur aufgeführt. In diesem Beispiel wurden aus 9 Zeilen der Prozedur sort in C 35 Zeilen in der MIPS-Assemblersprache.



**Vertiefung:** Eine Optimierungsmöglichkeit, die sich auf dieses Beispiel anwenden lässt, ist das in Abschnitt 2.11 erwähnte *Inlining von Prozeduren*. Anstatt Argumente in Parametern zu übergeben und den Code mit einem jal-Befehl aufzurufen, kopiert der Compiler den Code im Rumpf der swap-Prozedur an die Stelle, an der sich der Aufruf von swap befindet. Mit dem Inlining können in diesem Beispiel vier Befehle gespart werden. Diese Optimierung hat jedoch den Nachteil, dass der kompilierte Code länger wird, wenn die eingefügte Prozedur an mehreren Stellen aufgerufen wird. Eine Codeerweiterung dieser Art kann zu einer Leistungsbeeinträchtigung führen, wenn sich dadurch die Cache-Fehlzugriffssrate erhöht (siehe Kapitel 7).

Die MIPS-Compiler reservieren immer Platz auf dem Keller für die Argumente, für den Fall, dass diese gespeichert werden müssen. Daher dekrementieren sie in Wirklichkeit \$sp um 16, um für alle vier Argumentregister (16 Byte) Platz zu schaffen. Ein Grund dafür ist der, dass C eine vararg-Option bereitstellt, mit deren Hilfe ein Zeiger beispielsweise das dritte Argument für eine Prozedur auswählen kann. Wenn der Compiler auf die seltene vararg-Option trifft, kopiert er die vier Argumentregister auf den Keller in die vier reservierten Positionen.

Tab. 2.21 MIPS-Assemblerversion der Prozedur sort in Abbildung 2.15.

Register retten		
sort:	addi \$sp, \$sp, -20 sw \$ra, 16(\$sp) sw \$s3, 12(\$sp) sw \$s2, 8(\$sp) sw \$s1, 4(\$sp) sw \$s0, 0(\$sp)	# schaffe Platz auf dem Keller für 5 Reg. # sichere \$ra auf dem Keller # sichere \$s3 auf dem Keller # sichere \$s2 auf dem Keller # sichere \$s1 auf dem Keller # sichere \$s0 auf dem Keller
Prozedurrumpf		
Parameter kopieren	move \$s2, \$a0 move \$s3, \$a1	# kopiere Parameter \$a0 in \$s2 (sichere \$a0) # kopiere Parameter \$a1 in \$s3 (sichere \$a1)
Äußere Schleife	move \$s0, \$zero for1st: slt \$t0, \$s0, \$s3 beq \$t0, \$zero, exit1	# i = 0 # Reg. \$t0 = 0, wenn \$s0 ≤ \$s3 (i ≤ n) # verzweige zu exit1, wenn \$s0 ≤ \$s3 (i ≤ n)
Innere Schleife	addi \$s1, \$s0, -1 for2tst: slti \$t0, \$s1, 0 bne \$t0, \$zero, exit2 sll \$t1, \$s1, 2 add \$t2, \$s2, \$t1 lw \$t3, 0(\$t2) lw \$t4, 4(\$t2) slt \$t0, \$t4, \$t3 beq \$t0, \$zero, exit2	# j = i - 1 # Reg. \$t0 = 1, wenn \$s1 < 0 (j < 0) # verzweige zu exit2, wenn \$s1 < 0 (j < 0) # Reg. \$t1 = j * 4 # Reg. \$t2 = v + (j * 4) # Reg. \$t3 = v[j] # Reg. \$t4 = v[j + 1] # Reg. \$t0 = 0, wenn \$t4 ≥ \$t3 # verzweige zu exit2, wenn \$t4 ≥ \$t3
Parameter übergeben und Aufruf	move \$a0, \$s2 move \$a1, \$s1 jal swap	# 1. Parameter von swap ist v (alter Wert von \$a0) # 2. Parameter von swap ist j # swap Code siehe Tabelle 2.20
Innere Schleife	addi \$s1, \$s1, -1 j for2tst	# j -= 1 # springe zum Test der inneren Schleife
Äußere Schleife	exit2: addi \$s0, \$s0, 1 j for1st	# i += 1 # springe zum Test der äußeren Schleife
Register wiederherstellen		
exit1:	lw \$s0, 0(\$sp) lw \$s1, 4(\$sp) lw \$s2, 8(\$sp) lw \$s3, 12(\$sp) lw \$ra, 16(\$sp) addi \$sp, \$sp, 20	# stelle \$s0 wieder her # stelle \$s1 wieder her # stelle \$s2 wieder her # stelle \$s3 wieder her # stelle \$ra wieder her # stelle Kellerzeiger wieder her
Prozedurrücksprung		
	jr \$ra	# springe zurück zur aufrufenden Prozedur

Tab. 2.22 Vergleich von Leistung, Anzahl von Befehlen und CPI-Wert unter Verwendung von Compileroptimierungen für Bubble Sort. Die Programme sortierten 100 000 Wörter, wobei das Feld mit zufälligen Werten initialisiert wurde. Diese Programme wurden auf einem Pentium 4 mit einer Taktfrequenz von 3,06 GHz und einem 533-MHz-Systembus mit einem PC2100 DDR SDRAM-Hauptspeicher mit 2 GB ausgeführt. Dabei wurde Linux 2.4.20 verwendet.

gcc-Optimierung	Relative Leistung	Taktzyklen (in Mio.)	Befehlszahl (in Mio.)	CPI-Wert
keine	1,00	158 615	114 938	1,38
O1 (mittel)	2,37	66 990	37 470	1,79
O2 (vollständig)	2,38	66 521	39 993	1,66
O3 (Prozedurintegration)	2,41	65 747	44 993	1,46

## Grundlegendes zur Leistungsfähigkeit von Programmen

In Tabelle 2.22 sind für ein Sortierprogramm die Auswirkungen der Compileroptimierung auf die Leistungsfähigkeit, die Kompilierungszeit, die Taktzyklen, die Anzahl der ausgeführten Befehle und den CPI-Wert dargestellt. Nicht optimierter Code weist den besten CPI-Wert und O1-optimierter Code die geringste Anzahl der ausgeführten Befehle auf. O3-optimierter Code wird dagegen am schnellsten ausgeführt, was uns daran erinnert, dass Zeit das einzige genaue Maß für die Leistungsfähigkeit von Programmen ist.

In Tabelle 2.23 werden die Auswirkungen von Programmiersprachen – Übersetzung im Vergleich zur Interpretation – und Algorithmen auf das Leistungsverhalten von Sortiervorgängen verglichen. Der vierten Spalte ist zu entnehmen, dass das nicht optimierte C-Programm 8,3-mal schneller ist als der interpretierte Java-Code für den Bubble Sort. Mithilfe des Just-in-Time-Java-Compilers wird Java 2,1-mal *schneller* als der nicht optimierte C-Code und 1,13-mal *langsamer* als der maximal optimierte C-Code. (Im nächsten Abschnitt finden Sie ausführlichere Informationen zum Vergleich von Interpretation und Übersetzung von Java sowie zum Java- und MIPS-Code für Bubble Sort.) Für Quicksort in Spalte 5 sind die Quotienten kleiner, wahrscheinlich weil es schwieriger ist, die Kosten für die Laufzeitkomplizierung gegenüber der kürzeren Ausführungszeit auszugleichen. In der letzten Spalte werden die Auswirkungen eines besseren Algorithmus veranschaulicht. Hier ist beim Sortieren von 100 000 Elementen eine Leistungssteigerung um drei Größenordnungen zu verzeichnen. Sogar beim Vergleich von interpretiertem Java-Code in Spalte 5 mit dem C-Compiler bei maximaler Optimierung in Spalte 4 schlägt Quicksort Bubble Sort um den Faktor 50 ( $0,05 \times 2468 / 2,41$  oder 123 zu 2,41).

**Tab. 2.23 Leistungsverhalten zweier Sortialgorithmen in C und Java mit Interpretation und optimierenden Compilern im Vergleich zu nicht optimierter C-Version.** In der letzten Spalte ist der Leistungsvorteil von Quicksort gegenüber Bubble Sort für jede Sprache und Ausführungsmethode dargestellt. Diese Programme wurden auf demselben System wie die Programme in Tabelle 2.22 ausgeführt. Bei der JVM handelt es sich um die Sun-Version 1.3.1 und beim JIT-Compiler um die Sun Hotspot-Version 1.3.1.

Sprache	Ausführungs-methode	Optimierung	Relative Leistung von Bubble Sort	Relative Leistung von Quicksort	Beschleunigung: Quicksort und Bubble Sort im Vergleich
C	Compiler	keine	1,00	1,00	2468
	Compiler	O1	2,37	1,50	1562
	Compiler	O2	2,38	1,50	1555
	Compiler	O3	2,41	1,91	1955
Java	Interpreter	—	0,12	0,05	1050
	Just-in-Time-Compiler	—	2,13	0,29	338