

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, 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-4152-313
Universitätsbibliothek Bern
Zentralbibliothek
2007



Spektrum
AKADEMISCHER VERLAG

IT 450 12

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

5.1

Einführung

In Kapitel 4 haben wir gesehen, dass das Leistungsverhalten eines Rechners von drei Schlüssel-Faktoren bestimmt wird: Befehlszahl, Taktzykluszeit und Taktzyklen pro Befehl (CPI, *clock cycles per instruction*). Der Compiler und die Befehlssatzarchitektur, die wir in Kapitel 2 und 3 untersucht haben, sind für die für ein bestimmtes Programm erforderliche Befehlszahl verantwortlich. Sowohl die Taktzykluszeit als auch die Anzahl der Taktzyklen pro Befehl hängen dagegen von der Implementierung des Prozessors ab. In diesem Kapitel erstellen wir den Datenpfad und das Steuerwerk für zwei verschiedene Implementierungen des MIPS-Befehlssatzes.

In diesem Kapitel werden die Prinzipien und Methoden erläutert, die beim Implementieren eines Prozessors verwendet werden. Die Erläuterung beginnt mit einer sehr abstrakten und vereinfachten Übersicht in diesem Abschnitt. Es folgen Abschnitte, in denen ein Datenpfad erstellt und eine einfache Version eines Prozessors entwickelt wird, die zum Implementieren von Befehlssätzen wie MIPS ausreicht, und mit der Entwicklung des Konzepts schließt, das zum Implementieren komplexerer Befehlssätze wie IA-32 erforderlich ist.

Leser, die die Interpretation von Befehlen auf höherer Ebene und deren Auswirkungen auf die Leistungsfähigkeit von Programmen verstehen möchten, finden in diesem Abschnitt genügend Hintergrundinformationen, um diese Konzepte sowie die grundlegenden Konzepte für das Pipelining zu verstehen, das in Abschnitt 6.1 des nächsten Kapitels erläutert wird.

Leser, die wissen möchten, wie die Hardware Befehle implementiert, finden alle erforderlichen weiterführenden Informationen in den Abschnitten 5.3 und 5.4. Außerdem enthalten diese beiden Abschnitte alle Informationen, die zum Verstehen der Konzepte für das Pipelining in Kapitel 6 erforderlich sind. Nur die Leser, die sich für das Hardwaredesign interessieren, müssen mehr lesen.

In den restlichen Abschnitten dieses Kapitels wird beschrieben, wie moderne Hardware normalerweise implementiert wird, z.B. komplexe Prozessoren der Intel-Pentium-Reihe. Dabei werden die grundlegenden Prinzipien der Steuerung eines endlichen Automaten sowie die unterschiedlichen Implementierungsmethoden wie Mikroprogrammierung erläutert. Leser, die über den Prozessor und die Leistungsfähigkeit von Prozessoren mehr wissen möchten, finden in den Abschnitten 5.4 und 5.5 die entsprechenden Informationen. Leser, die sich für das moderne Hardwaredesign interessieren, lernen in **Abschnitt 5.7** auf CD die Mikroprogrammierung kennen. Dabei handelt es sich um eine Technik zum Implementieren komplexerer Steuerwerke wie die in IA-32-Prozessoren. Und in **Abschnitt 5.8** (ebenfalls auf CD) wird beschrieben, wie Hardware mithilfe von Hardwarebeschreibungssprachen und CAD-Tools implementiert wird.



Eine einfache MIPS-Implementierung

Wir werden eine Implementierung untersuchen, die einen Teil des zentralen MIPS-Befehlssatzes enthält:

- Die Speicherzugriffsbefehle `load word (lw)` und `store word (sw)`
- Die arithmetisch-logischen Befehle `add`, `sub`, `and`, `or` und `slt`
- Die Befehle `branch on equal (beq)` und `jump (j)`, die wir als letzte betrachten werden

Dieser Teil des Befehlssatzes enthält weder alle Ganzzahlbefehle (z.B. fehlen `shift`, `multiply` und `divide`), noch Gleitkommabefehle. Die wichtigsten Prinzipien zum Erstellen eines Datenpfads und zum Entwickeln des Steuerwerks werden jedoch dargestellt. Die Implementierung der restlichen Befehle erfolgt auf ähnliche Weise.

Bei der Betrachtung der Implementierung haben wir die Gelegenheit zu sehen, wie sich die Befehlssatzarchitektur auf viele Aspekte der Implementierung auswirkt, und welche Auswirkungen die Wahl unterschiedlicher Implementierungsstrategien auf die Taktfrequenz und den CPI-Wert des Rechners hat. Viele der zentralen, in Kapitel 4 vorgestellten Prinzipien für die Hardwareentwicklung wie die Richtlinien *Optimiere den häufig vorkommenden Fall* und *Einfachheit bevorzugt Regelmäßigkeit* lassen sich anhand der Implementierung veranschaulichen. Die meisten Konzepte, die in diesem und im nächsten Kapitel zum Implementieren des MIPS-Teilbefehlssatzes verwendet werden, sind dieselben Ideen, die der Erstellung eines breiten Spektrums an Rechnern zugrunde liegen, von Hochleistungsservern über Allzweckmikroprozessoren bis hin zu eingebetteten Prozessoren, die zunehmend in Produkten wie Videorekordern und Kraftfahrzeugen eingesetzt werden.

Übersicht über die Implementierung

In Kapitel 2 und 3 wurden die zentralen MIPS-Befehle wie die arithmetisch-logischen Ganzzahlbefehle, die Speicherzugriffsbefehle und die Sprungbefehle beschrieben. Beim Implementieren dieser Befehle wiederholt sich vieles unabhängig von der Befehlsklasse. So sind bei allen Befehlen die ersten beiden Schritte dieselben:

1. Senden des Befehlszählers an den Speicher, der den Code enthält, und Holen des Befehls aus diesem Speicher.
2. Lesen eines oder zweier Register wobei die Auswahl des zu lesenden Registers mithilfe von Feldern des Befehls erfolgt. Beim `load word`-Befehl muss nur ein Register gelesen werden, bei den meisten anderen Befehlen dagegen zwei.

Welche Schritte nach diesen beiden Schritten zum Durchführen des Befehls erforderlich sind, hängt von der Befehlsklasse ab. Erfreulicherweise sind das für die drei Befehlsklassen (Speicherzugriff, arithmetisch-logische Befehle und Sprünge) unabhängig vom exakten Opcode im Großen und Ganzen dieselben Schritte.

Sogar über unterschiedliche Befehlsklassen hinweg gibt es Ähnlichkeiten. So verwenden beispielsweise außer dem Sprung alle Befehlsklassen nach dem Lesen der Register die ALU (Arithmetic Logical Unit, arithmetisch-logische Einheit) bzw. das Rechenwerk. Die Speicherzugriffsbefehle verwenden die ALU für die Adressberechnung, die arithmetisch-logischen Befehle für die Ausführung von Operationen und die Sprünge für Vergleiche. Die Einfachheit und Regelmäßigkeit des Befehlssatzes vereinfacht somit die Implementierung, da viele Befehlsklassen ähnlich ausgeführt werden.

Nach dem Einsatz der ALU sind unterschiedliche Schritte zur Beendigung der verschiedenen Befehle erforderlich. Ein Speicherzugriffsbefehl muss entweder im Rahmen eines `store`-Befehls zum Schreiben von Daten oder im Rahmen eines `load`-Befehls zum Lesen von Daten auf den Speicher zugreifen. Ein arithmetisch-logischer Befehl muss die Daten von der ALU zurück in ein Register schreiben. Und bei einem Sprungbefehl schließlich müssen wir die nachfolgende Befehlsadresse je nach dem Ergebnis des Vergleichs möglicherweise ändern. Andernfalls muss der Befehlszähler um 4 erhöht werden, um so die Adresse des nachfolgenden Befehls zu erhalten.

In Abbildung 5.1 ist die abstrakte Sicht einer MIPS-Implementierung mit den unterschiedlichen Funktionseinheiten und ihren Verbindungen dargestellt. Hier wird zwar der Großteil des Datenflusses durch den Prozessor gezeigt. Zwei wichtige Aspekte der Befehlausführung fehlen jedoch.

Zum einen ist in Abbildung 5.1 an verschiedenen Stellen dargestellt, dass Daten von zwei verschiedenen Quellen kommend zu einer bestimmten Einheit gelangen. So kann beispielsweise der Wert, der in den Befehlszähler geschrieben wird, von einem von zwei möglichen Addierern stammen, und die Daten, die in den Registersatz geschrieben werden, können entweder von der ALU oder vom Datenspeicher stammen. In der Praxis können diese Datenleitungen nicht einfach miteinander verdrahtet werden. Wir müssen ein Element bereitstellen, das unter den verschiedenen Quellen eine auswählt und eine dieser Quellen an ihr Ziel führt. Diese Auswahl wird üblicherweise von einer Einheit getroffen, die als *Multiplexer (MUX)* bezeichnet wird, obwohl die Bezeichnung *Datenselektor* besser passen würde. Der auf CD in **Appendix B** ausführlich beschriebene Multiplexer wählt je nach der Festlegung seiner Steuerleitungen aus verschiedenen Eingängen einen aus. Dabei werden die Steuerleitungen in erster Linie anhand von Informationen aus dem ausgeführten Befehl festgelegt.

Zum anderen müssen einige der Einheiten je nach Befehlstyp unterschiedlich angesteuert werden. So muss der Datenspeicher beispielsweise bei einem Ladebefehl lesen und bei einem Speicherbefehl schreiben. Bei einem Ladebefehl und bei einem arithmetisch-logischen Befehl muss in den Registersatz geschrieben werden. Und die ALU muss, wie wir in Kapitel 3 bereits gesehen haben, eine von mehreren möglichen

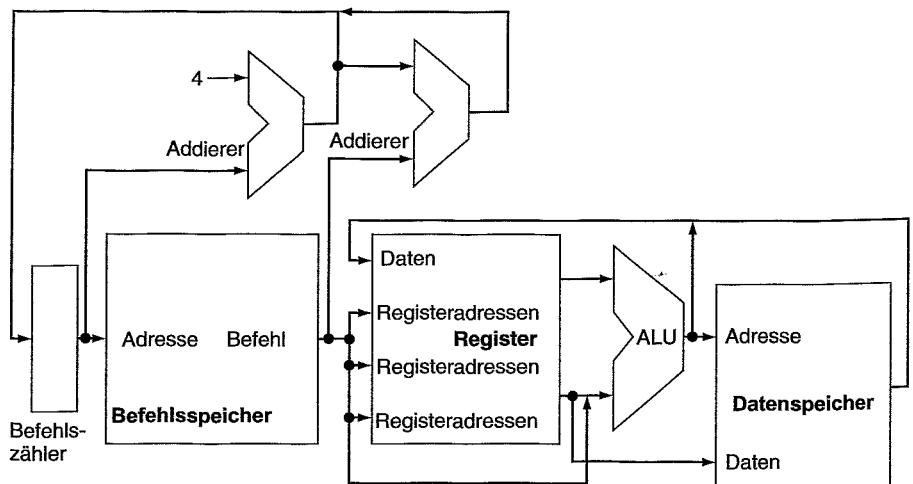


Abb. 5.1 Eine abstrakte Darstellung der Implementierung eines Teils des MIPS-Befehlsatzes mit den wichtigsten Funktionseinheiten und den wichtigsten Verbindungen der Funktionseinheiten untereinander. Alle Befehle beginnen mit der Verwendung des Befehlszählers, um die Befehlsadresse in den Befehlsspeicher zu laden. Nach dem Laden des Befehls werden die von einem Befehl verwendeten Registeroperanden durch Felder dieses Befehls bestimmt. Nach dem Laden der Registeroperanden kann mit diesen eine Speicheradresse (für einen Lade- oder Speicherbefehl), ein arithmetisches Ergebnis (für einen arithmetisch-logischen Integer-Befehl) oder ein Vergleich (für einen Sprung) berechnet werden. Wenn es sich um einen arithmetisch-logischen Befehl handelt, muss das Ergebnis der ALU in ein Register geschrieben werden. Wenn es sich um eine Lade- oder Speicheroperation handelt, wird das Ergebnis der ALU als Adresse zum Speichern eines Werts aus den Registern oder zum Laden eines Werts aus dem Speicher in die Register verwendet. Das Ergebnis aus der ALU oder aus dem Speicher wird in den Registersatz zurückgeschrieben. Bei Sprüngen wird mit dem Ergebnis der ALU die nächste Befehlsadresse ermittelt, die entweder von der ALU (Befehlszählerwert und Sprung-Offset werden addiert) oder von einem Addierer stammt, der den aktuellen Befehlszählerwert um 4 erhöht. Die dicken Linien, mit denen die Funktionseinheiten miteinander verbunden sind, stellen Busse dar, die aus mehreren Signalleitungen bestehen. Die Pfeile zeigen die Richtung des Datenflusses an. Da Signalleitungen einander kreuzen können, ist durch einen Punkt dargestellt, wenn einander kreuzende Leitungen miteinander verbunden sind.



Operationen ausführen. (In Appendix B auf CD wird der logische Aufbau der ALU ausführlich beschrieben.) Wie die Multiplexer werden diese Operationen durch Steuerleitungen gesteuert, die auf der Grundlage von verschiedenen Feldern im Befehl belegt werden.

In Abbildung 5.2 ist der Datenpfad aus Abbildung 5.1 mit den drei erforderlichen Multiplexern sowie mit den Steuerleitungen für die wichtigsten Funktionseinheiten dargestellt. Ein Steuerwerk mit dem Maschinenbefehl als Eingangssignal bestimmt, wie die Steuerleitungen für die Funktionseinheiten und für zwei der Multiplexer belegt werden. Der dritte Multiplexer legt anhand des Zero-Ausgangs der ALU fest, ob der Befehlszählerwert + 4 oder die Sprungzieladresse in den Befehlszähler geschrieben wird, um bedingte Sprünge mit Vergleich (beq-Befehl) durchzuführen. Aufgrund der Regelmäßigkeit und Einfachheit des MIPS-Befehlssatzes kann die Belegung der Steuerleitungen mit einem einfachen Decodervorgang bestimmt werden.

In den restlichen Abschnitten des Kapitels vervollständigen wir diese Darstellung mit weiteren Details, die es erforderlich machen, dass weitere Funktionseinheiten eingefügt, die Anzahl der Verbindungen zwischen den Einheiten erhöht und ein Steuer-

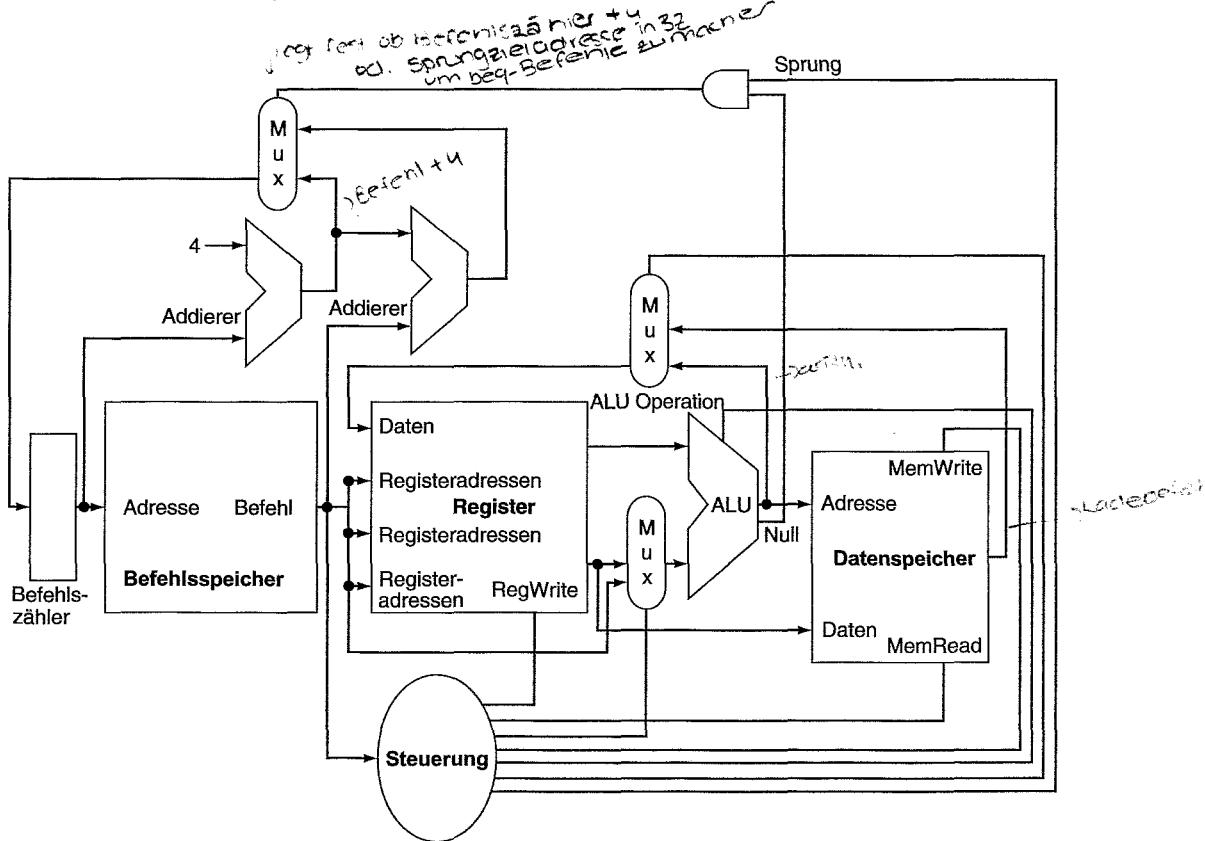


Abb. 5.2 Die einfache Implementierung eines Teils des MIPS-Befehlssatzes mit den erforderlichen Multiplexern und Steuerleitungen. Der oberste Multiplexer bestimmt, welcher Wert den Befehlszähler (Befehlszählerwert + 4 oder die Sprungzieladresse) ersetzt. → Der Multiplexer wird durch das Gatter gesteuert, das das Signal am Zero-Ausgang der ALU und ein Steuersignal mittels AND-Verknüpfung miteinander verknüpft, wobei das Steuersignal angibt, dass es sich bei dem Befehl um einen Sprung handelt. Der Multiplexer, dessen Ausgang auf den Dateneingang des Registersatzes geht, verbindet diesen mit dem ALU-Ausgang (bei einem arithmetisch-logischen Befehl) oder dem Datenspeicher-Ausgang (bei einem Ladebefehl). Der unterste Multiplexer bestimmt, ob der zweite ALU-Eingang mit den Registern (bei einem arithmetisch-logischen Nicht-immediate-Befehl) oder mit dem Offset-Feld des Befehls (bei einer Immediate-Operation, einem Lade- oder Speicherbefehl oder einem Sprung) belegt wird. Die weiteren Steuerleitungen sind unkompliziert und bestimmen die Operation, die in der ALU ausgeführt wird, ob Daten aus dem Datenspeicher ausgelesen oder in den Datenspeicher geschrieben werden und ob die Register eine Schreiboperation durchführen sollen. Die Steuerleitungen sind zum leichteren Erkennen farblich hervorgehoben.

werk ergänzt wird. Dieses bestimmt, welche Schritte für unterschiedliche Befehlsklassen durchgeführt werden. In den Abschnitten 5.3 und 5.4 wird eine einfache Implementierung beschrieben, bei der für jeden Befehl ein langer Taktzyklus verwendet und die allgemeine Form aus Abbildung 5.1 und 5.2 befolgt wird. Bei diesem ersten Entwurf beginnt die Ausführung jedes Befehls an einer Taktflanke und endet an der nächsten Taktflanke.

Dieser Entwurf ist zwar leichter verständlich, jedoch nicht sinnvoll, da er langsamer als eine Implementierung wäre, bei der unterschiedliche Befehlsklassen unterschiedlich viele Taktzyklen beanspruchen können, die teilweise deutlich kürzer sein könnten. Nach dem Entwurf der Steuerung für diese einfache Architektur werden wir eine Implementierung betrachten, bei der für die einzelnen Befehle mehrere Taktzyklen verwendet werden. Dieser Mehrzyklenentwurf wird bei der Beschreibung komplexerer Steuerungskonzepte, bei der Beschreibung der Ausnahmebehandlung und der Verwendung von Hardwarebeschreibungssprachen in den **Abschnitten 5.5 und 5.6** sowie in den **Abschnitten 5.7 und 5.8** auf CD verwendet.



Die in diesem Abschnitt konzeptionell beschriebene Eintaktimplementierung des Datenpfads *muss* getrennte Befehls- und Datenspeicher aufweisen, weil

1. MIPS-Daten und MIPS-Befehle unterschiedliche Formate haben und daher unterschiedliche Speicher benötigt werden,
2. getrennte Speicher kostengünstiger sind,
3. der Prozessor mit einem Zyklus arbeitet und daher keinen Ein-Port-Speicher für zwei verschiedene Zugriffe innerhalb dieses Zyklus verwenden kann.

5.2

Konventionen für den Entwurf von Logikschaltungen

Wenn wir den Entwurf einer Architektur beschreiben möchten, müssen wir festlegen, wie die Logikschaltungen, mit denen die Architektur implementiert wird, funktionieren sollen und wie der Rechner getaktet werden soll. In diesem Abschnitt werden einige zentrale Begriffe der digitalen Logikschaltungen vorgestellt, die in diesem Kapitel häufig verwendet werden. Wenn Sie von digitalen Logikschaltungen nur wenig oder noch gar nichts wissen, ist es hilfreich, vor dem Weiterlesen zunächst auf CD **Appendix B** zu lesen.

Die Funktionseinheiten in der MIPS-Implementierung bestehen aus zwei verschiedenen Arten von Logikbausteinen: Bausteine, die Datenwerte verarbeiten, und Bausteine, die Zustände speichern. Bei den Bausteinen, die Datenwerte verarbeiten, handelt es sich immer um *Schaltnetze* (bzw. *kombinatorische Elemente*, *combinational*).

Das bedeutet, dass deren Ausgangssignale ausschließlich von den aktuellen Eingangssignalen abhängen. Ein gleiches Eingangssignal ergibt bei einem Schaltnetz immer dasselbe Ausgangssignal. Bei den in Abbildung 5.1 dargestellten und in Kapitel 3 und **Appendix B** beschriebenen ALU handelt es sich um ein Schaltnetz. Bei gleichen Eingangssignalen erzeugt diese ALU immer dieselben Ausgangssignale, da sie über keinen internen Speicher verfügt.

Die anderen Bausteine im Entwurf sind keine Schaltnetze, sondern *sie beinhalten Zustände*. Ein Baustein kann Zustände speichern, wenn er über internen Speicher verfügt. Diese Bausteine werden als **Schaltwerke** (*state elements*) bezeichnet, denn der



Rechner kann nach einer Unterbrechung der Stromversorgung erneut gestartet werden, indem die Schaltwerke mit den Werten geladen werden, die zum Zeitpunkt der Unterbrechung der Stromversorgung gespeichert waren. Wenn die Zustände gespeichert und wiederhergestellt werden, ist es so, als wäre die Stromversorgung nie unterbrochen gewesen. Diese Zustände definieren den Rechner also vollständig. Die Befehls- und Datenspeicher sowie die Register in Abbildung 5.1 sind Beispiele für Schaltwerke.

Ein Schaltwerk hat mindestens zwei Eingänge und einen Ausgang. An den Eingängen müssen die Leitungen für den Datenwert, der im Schaltwerk gespeichert wird, und für das Taktsignal, das bestimmt, wann der Datenwert gespeichert wird, anliegen. Am Ausgang eines Schaltwerkes wird der Wert bereitgestellt, der in einem früheren Taktzyklus geschrieben wurde. Ein Beispiel für ein logisch sehr einfaches Schaltwerk ist ein D-Flip-Flop (siehe **Appendix B**), das exakt diese beiden Eingänge (Wert und Takt) und einen Ausgang aufweist. Neben den Flip-Flops enthält unsere MIPS-Implementierung zwei weitere Arten von Schaltwerken: Speicher und Register. Beide sind in Abbildung 5.1 dargestellt. Das Taktsignal bestimmt, wann in das Schaltwerk geschrieben werden soll. Ausgelesen werden kann das Schaltwerk jederzeit.

Logikbausteine, die Zustände speichern, werden auch als *sequentielle Logik* bezeichnet, da die Ausgänge sowohl von den Eingängen als auch vom Inhalt des internen Speichers abhängen. So hängt beispielsweise der Ausgang einer Funktionseinheit, die die Register darstellt, sowohl von der Registernummer (bzw. -adresse) als auch von dem zuvor in den Registern gespeicherten Inhalt ab. Die Funktionsweise von Schaltnetzen und Schaltwerken sowie deren Entwurf wird in **Appendix B** ausführlich beschrieben.

Wir verwenden die Bezeichnung *auf logisch 1 gesetzt* für ein Signal mit dem logischen Wert 1 und *auf logisch 1 setzen (aktivieren)*, um anzugeben, dass ein Signal auf logisch 1 gesetzt werden muss. Mit *auf logisch 0 setzen* bzw. *auf logisch 0 gesetzt* wird ein Signal mit dem logischen Wert 0 bezeichnet.



Taktverfahren

Ein **Taktverfahren (clocking methodology)** bestimmt, wann Signale gelesen und wann sie geschrieben werden können. Es ist wichtig, den zeitlichen Ablauf von Lese- und Schreibvorgängen festzulegen. Denn wenn das Signal gleichzeitig geschrieben und gelesen wird, kann es vorkommen, dass der Wert des Lesevorgangs dem alten Wert, dem neu geschriebenen Wert oder sogar einer Mischung aus den beiden Werten entspricht! Es muss wohl nicht darauf hingewiesen werden, dass Rechnerentwürfe eine derartige Unvorhersagbarkeit nicht tolerieren können. Ein Taktverfahren wird entwickelt, um diesen Umstand zu verhindern.

Der Einfachheit halber gehen wir von einem **flankengesteuerten Taktverfahren (edge triggered clocking)** aus. Ein flankengesteuertes Taktverfahren bedeutet, dass sämtliche in einem sequentiellen Logikbaustein gespeicherten Werte nur während einer Taktflanke aktualisiert werden. Da nur Schaltwerke einen Datenwert speichern können, müssen die Eingänge sämtlicher Schaltnetze aus Schaltwerken kommen und die Ausgaben wieder in Schaltwerke geschrieben werden. An den Eingängen liegen die Werte an, die in einem vorhergehenden Taktzyklus geschrieben wurden, während an den Ausgängen die Werte anliegen, die in einem nachfolgenden Taktzyklus verwendet werden können.

In Abbildung 5.3 sind die beiden Schaltwerke dargestellt, die einen Block mit Schaltnetzen umgeben, der in einem Taktzyklus arbeitet. Alle Signale müssen sich innerhalb eines Taktzyklus vom Schaltwerk 1 über das Schaltnetz zum Schaltwerk 2 fortpflanzen. Die Zeit, die die Signale benötigen, bis sie bei Schaltwerk 2 ankommen, bestimmt die Länge des Taktzyklus.

Der Einfachheit halber stellen wir **kein Steuersignal (control signal)** dar, wenn ein Schaltwerk bei jeder aktiven Taktflanke beschrieben wird. Wenn ein Schaltwerk jedoch

Taktverfahren (clocking methodology) Das Verfahren, mit dem bestimmt wird, wann Daten relativ zum Takt gültig und stabil sind.

Flankengesteuertes Taktverfahren (edge triggered clocking) Ein Taktverfahren, bei dem alle Zustandsänderungen während einer Taktflanke erfolgen.

Steuersignal (control signal) Ein Signal, das für die Multiplexerauswahl oder für die Steuerung der Funktionsweise einer Funktionseinheit verwendet wird. Steht im Gegensatz zum Datensignal, das Daten enthält, die von der Funktionseinheit verarbeitet werden.

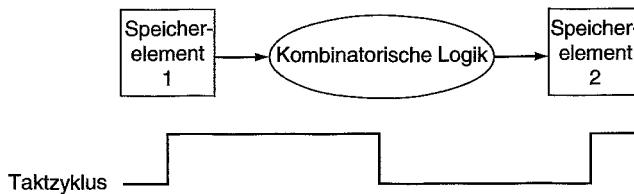


Abb. 5.3 Schaltnetze, Schaltwerke und Taktzyklus stehen in engem Zusammenhang. In einem synchronen digitalen System bestimmt der Taktzyklus, wann Elemente mit Zustand, also Schaltwerke, Werte in den internen Speicher schreiben. Alle Eingangssignale an einem Schaltwerk müssen einen stabilen Wert erreichen (d.h. sie müssen einen Wert erreicht haben, der sich bis nach der Taktflanke nicht verändert), bevor der Zustand bei einer aktiven Taktflanke verändert wird. Alle Schaltwerke sowie der Speicher werden als flankengesteuert betrachtet.

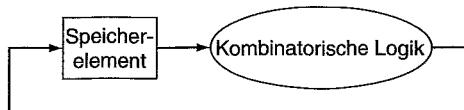


Abb. 5.4 Mithilfe eines flankengesteuerten Taktverfahrens kann ein Schaltwerk in einem Taktzyklus gelesen und beschrieben werden, ohne dass eine Wetttrennbedingung (race) entsteht, die zu undefinierten Datenwerten führen könnte. Der Taktzyklus muss jedoch lange genug sein, damit die Eingangswerte stabil sind, wenn die aktive Taktflanke erscheint. Ein Feedback kann aufgrund der flankengesteuerten Aktualisierung des Schaltwerkes nicht innerhalb eines Taktzyklus erfolgen. Wenn ein Feedback möglich wäre, würde dieser Entwurf nicht einwandfrei funktionieren. Unser Entwurf in diesem und im nachfolgenden Kapitel beruht auf dem flankengesteuerten Taktverfahren und auf einer Struktur wie der in dieser Abbildung.

nicht bei jeder Taktflanke aktualisiert wird, ist ein ausdrückliches Schreibsteuersignal erforderlich. Sowohl das Taktsignal als auch das Schreibsteuersignal sind Eingangssignale und das Schaltwerk wird nur überschrieben, wenn das Schreibsteuersignal auf logisch 1 gesetzt ist und eine Taktflanke vorliegt.

Mit einem flankengesteuerten Verfahren ist es möglich, in einem Taktzyklus den Inhalt eines Registers zu lesen, den Wert durch ein Schaltnetz zu senden und dieses Register zurückzuschreiben (siehe Abbildung 5.4). Dabei spielt es keine Rolle, ob wir annehmen, dass alle Schreibvorgänge bei der steigenden Taktflanke oder bei der fallenden Taktflanke stattfinden, da die Eingänge am kombinatorischen Logikbaustein nur bei der ausgewählten Taktflanke geändert werden können. Beim flankengesteuerten Taktverfahren erfolgt innerhalb eines Taktzyklus *kein* Feedback und die Logik in Abbildung 5.4 funktioniert ordnungsgemäß. Auf CD in **Appendix B** werden weitere Taktbeschränkungen – wie Voreinstell-, Setup- und Haltezeiten (hold) – sowie weitere Taktverfahren beschrieben.

Fast alle Schaltwerke und Logikbausteine haben 32 Bit breite Eingänge und Ausgänge, da die meisten der vom Prozessor verarbeiteten Daten 32 Bit breit sind. Wenn eine Einheit einen anderen als einen 32 Bit breiten Eingang oder Ausgang aufweist, werden wir darauf hinweisen. In den Abbildungen sind *Busse*, d.h. Signalleitungen, die breiter als 1 Bit sind, mit dicken Linien dargestellt. Gelegentlich werden mehrere Busse zu einem breiteren Bus zusammengefasst. So kann es beispielsweise vorkommen, dass wir einen 32-Bit-Bus durch Zusammenfassen von zwei 16-Bit-Bussen erhalten möchten. In diesem Fall sind die zu einem breiteren Bus zusammengefassten Busleitungen mit einer diagonal kreuzenden Linie und einer Markierung gekennzeichnet. Die Richtung des Datenflusses zwischen Elementen wird mithilfe von Pfeilen angegeben. Und schließlich sind Steuersignale im Gegensatz zu Datensignalen farblich gekennzeichnet. Diese Unterscheidung wird im Verlauf dieses Kapitels klarer werden.



Richtig oder falsch: Da der Registersatz in einem Taktzyklus gelesen und beschrieben wird, muss ein MIPS-Datenpfad, der mit flankengesteuerten Schreibvorgängen arbeitet, über mehrere Registersätze verfügen.



5.3

Aufbau eines Datenpfads

Am vernünftigsten ist es, einen Datenpfadentwurf damit zu beginnen, zu überprüfen, welche Hauptkomponenten zum Ausführen der einzelnen MIPS-Befehlsklassen erforderlich sind. Betrachten wir also zunächst, welche **Bausteine im Datenpfad (data path elements)** für die einzelnen Befehle erforderlich sind. Bei der Betrachtung der Bausteine im Datenpfad schildern wir auch die zugehörigen Steuersignale.

In Abbildung 5.5 ist das erste Element dargestellt, das wir benötigen: Eine Speicherleinheit zum Speichern der Befehle eines Programms und zum Bereitstellen von Befehlen für eine gegebene Adresse. In Abbildung 5.5 ist außerdem ein Register dargestellt, das als **Befehlszähler (program counter)** bezeichnet und zum Speichern der Adresse des aktuellen Befehls verwendet wird. Schließlich benötigen wir noch einen Addierer zum Inkrementieren des Befehlszählers, damit dieser die Adresse des nächsten Befehls angibt. Bei diesem Addierer handelt es sich um eine kombinatorische Logik, die aus der in Kapitel 3 beschriebenen und auf CD in **Appendix B** ausführlich entworfenen ALU aufgebaut wird, indem die Steuerleitungen so miteinander verdrahtet werden, dass das Steuerwerk immer eine Addition vorgibt. Eine ALU dieser Art wird wie in Abbildung 5.5 mit **Addierer** gekennzeichnet, um zu verdeutlichen, dass diese ALU ein permanenter Addierer ist und keine der anderen ALU-Funktionen ausführen kann.

Um einen Befehl auszuführen, muss der Befehl zunächst aus dem Speicher geholt werden. Um die Ausführung des nächsten Befehls vorzubereiten, muss zudem der Befehlszähler inkrementiert werden, so dass er auf den nächsten Befehl 4 Byte weiter zeigt. In Abbildung 5.6 ist dargestellt, wie die drei Elemente aus Abbildung 5.5 zu einem Datenpfad zusammengefügt werden, der Befehle aus dem Speicher lädt und den

Baustein im Datenpfad (data path element) Eine Funktionseinheit zum Verarbeiten oder Speichern von Daten in einem Prozessor. Bei der MIPS-Implementierung zählen die Befehls- und Datenspeicher, der Registersatz, die ALU (*Arithmetic Logical Unit*, arithmetisch-logische Einheit) und Addierer zu den Bausteinen im Datenpfad.

Befehlszähler (program counter, pc) Auch **Befehlszeiger** genannt. Das Register, das die Adresse des Befehls im aktuell ausgeführten Programm enthält, der gerade ausgeführt wird.

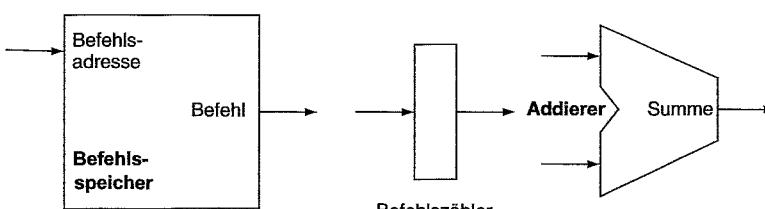


Abb. 5.5 Zum Speichern und Laden von Befehlen sind zwei Schaltwerke erforderlich, und ein Addierer wird benötigt, um die nächste Befehlsadresse zu berechnen. Bei den beiden Schaltwerken handelt es sich um den Befehlsspeicher und den Befehlszähler. Der Befehlsspeicher muss lediglich den Lesezugriff ermöglichen, da der Datenpfad keine Befehle schreibt. Da der Befehlsspeicher nur liest, wird er wie ein Schaltwerk behandelt: Der Ausgang gibt jederzeit den Inhalt der im Adresseingang angegebenen Position an. Ein Lesesteuerzeichen ist nicht erforderlich. (Beim Laden des Programms muss in den Befehlsspeicher geschrieben werden. Dies ist nicht schwierig hinzuzufügen. Daher ignorieren wir diesen Umstand der Einfachheit halber.) Beim Befehlszähler handelt es sich um ein 32-Bit-Register, in das am Ende von jedem Taktzyklus geschrieben wird und das daher kein Schreibsteuersignal benötigt. Der Addierer ist eine ALU, die so verdrahtet ist, dass sie immer ihre beiden 32-Bit-Eingänge addiert und das Ergebnis an ihrem Ausgang ausgibt.

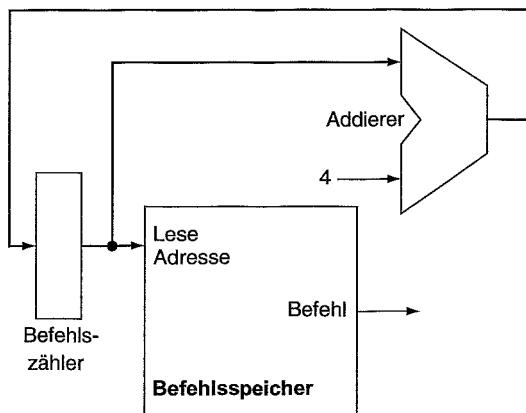


Abb. 5.6 Der Teil des Datenpfads, der zum Holen von Befehlen und zum Inkrementieren des Befehlszählers verwendet wird. Der geholte Befehl wird von anderen Teilen des Datenpfads verwendet.

Befehlszähler inkrementiert, so dass dieser auf die Adresse des nächsten Befehls in Folge zeigt.

Betrachten wir nun die Befehle in R-Format (siehe Tabelle 2.5 auf Seite 53). Diese Befehle lesen zwei Register, führen mit dem Inhalt der Register eine ALU-Operation durch und schreiben das Ergebnis zurück. Diese Befehle werden als *R-Befehle* oder *arithmetisch-logische Befehle* bezeichnet, da sie arithmetische oder logische Operationen durchführen. Zu dieser Befehlsklasse zählen die in Kapitel 2 vorgestellten Befehle add, sub, and, or und slt. Denken Sie daran, dass add \$t1,\$t2,\$t3 ein typisches Beispiel für einen Befehl dieser Art ist und bedeutet, dass \$t2 und \$t3 gelesen werden und \$t1 geschrieben wird.

Die 32 Allzweckregister des Prozessors werden in einer Struktur gespeichert, die als **Registersatz (register file)** bezeichnet wird. Ein Registersatz besteht aus mehreren Registern. Aus dem Registersatz können Register gelesen oder in den Registersatz können Register geschrieben werden, indem die Nummer bzw. Adresse des Registers im Registersatz angegeben wird. Im Registersatz ist der Registerzustand des Rechners gespeichert. Zudem wird eine ALU zum Verarbeiten der aus den Registern gelesenen Werte benötigt.

Da die R-Befehle drei Registeroperanden aufweisen, müssen pro Befehl zwei Datenwörter aus dem Registersatz gelesen und ein Datenwort in den Registersatz geschrieben werden. Für jedes Datenwort, das aus den Registern gelesen wird, benötigen wir ein Eingangssignal an den Registersatz, das die Adresse des Registers angibt, aus dem gelesen werden soll, und ein Ausgangssignal vom Registersatz, das den aus den Registern gelesenen Wert überträgt. Zum Schreiben eines Datenworts sind zwei Eingänge erforderlich: ein Eingang zum Angeben der *Adresse des Registers*, in das geschrieben werden soll, und ein Eingang zum Bereitstellen der *Daten*, die in das Register geschrieben werden sollen. Der Registersatz gibt immer den Inhalt des Registers aus, dessen Adresse sich am Lese Register-Eingang befindet. Schreibvorgänge werden dagegen durch das Schreibsteuersignal gesteuert, das auf logisch 1 gesetzt sein muss, damit bei der Taktflanke ein Schreibvorgang erfolgt. Somit benötigen wir, wie in Abbildung 5.7 dargestellt, insgesamt vier Eingänge (drei für Registeradressen und einen für Daten) und zwei Ausgänge (beide für Daten). Die Eingänge für Registeradressen sind 5 Bit breit und geben eines von 32 Registern ($32 = 2^5$) an, während der Dateneingangsbus und die beiden Datenausgangsbusse jeweils 32 Bit breit sind.

In Abbildung 5.7 ist die ALU dargestellt, die aus zwei 32-Bit-Eingängen ein 32-Bit-Ergebnis sowie ein 1-Bit-Signal generiert, falls das Ergebnis 0 ist. Das 4-Bit-Steuersignal für die ALU wird in **Appendix B** ausführlich beschrieben. Die Steuerung



Registersatz (register file) Ein Schaltwerk, das aus mehreren Registern besteht, die gelesen und in die geschrieben werden kann, indem die Adresse des Registers angegeben wird, auf das zugegriffen werden soll.

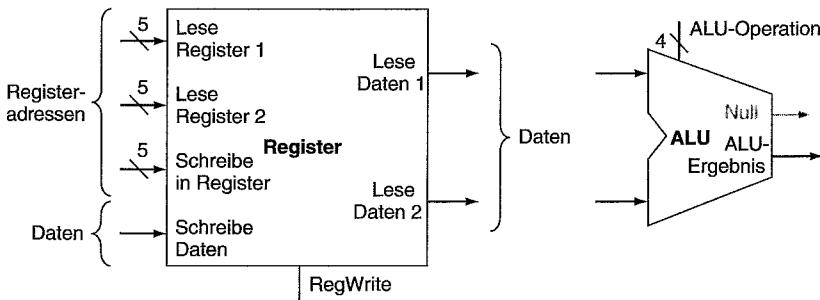


Abb. 5.7 Der Registersatz und die ALU sind die beiden Elemente, die zum Implementieren von ALU-Operationen im R-Format erforderlich sind. Der Registersatz enthält alle Register und verfügt über zwei Leseports und einen Schreibport. Der Aufbau von Multiport-Registersätzen wird in **Appendix B.8** auf der CD beschrieben. Der Registersatz gibt immer den Inhalt der Register entsprechend den an den Ausgängen anliegenden Lese-Register-Eingangssignalen aus. Dazu sind keine weiteren Steuereingänge erforderlich. Ein Registerlesevorgang muss dagegen durch Setzen des Schreibsteuersignals auf 0 explizit angegeben werden. Schreibvorgänge sind flankengesteuert, so dass alle Schreibeingänge (d.h., der zu schreibende Wert, die Registeradresse und das Schreibsteuersignal) bei der Taktflanke gültig sein müssen. Da Schreibvorgänge im Registersatz flankengesteuert sind, sind in unserem Entwurf Lese- und Schreibvorgänge im selben Register innerhalb eines Taktzyklus zulässig: Beim Lesevorgang wird der Wert gelesen, der in einem früheren Taktzyklus geschrieben wurde, während der Wert, der beim Schreibvorgang geschrieben wird, in einem nachfolgenden Taktzyklus zum Lesen bereitsteht. Die Eingangsleitungen, über die die Registeradresse an den Registersatz übertragen wird, sind 5 Bit breit, während die Leitungen, die die Datenwerte übertragen, 32 Bit breit sind. Die von der ALU durchzuführende Operation wird unter Verwendung der in **Appendix B** auf der CD entworfenen ALU mit dem 4 Bit breiten ALU-Operationssignal gesteuert. Wir werden den Zero-Detect-Ausgang der ALU später zum Implementieren von Sprüngen verwenden. Der Überlaufausgang wird erst in Abschnitt 5.6 bei der Beschreibung von Unterbrechungen benötigt und an dieser Stelle eingeführt und beschrieben.



der ALU wird später beschrieben, wenn wir wissen müssen, wie diese gesetzt werden muss.

Als Nächstes betrachten wir die MIPS-Befehle `load word` und `store word`, die die allgemeine Form `lw $t1, offset_value($t2)` oder `sw $t1, offset_value($t2)` aufweisen. Diese Befehle berechnen eine Speicheradresse, wobei das Basisregister `$t2` und das im Befehl enthaltene vorzeichenbehaftete 16-Bit-Offset-Feld addiert werden. Wenn es sich um einen Speicherbefehl handelt, muss der zu speichernde Wert ebenfalls aus dem Registersatz gelesen werden, wo er sich in Register `$t1` befindet. Wenn es sich um einen Ladebefehl handelt, muss der aus dem Speicher gelesene Wert in das angegebene Register `$t1` im Registersatz geschrieben werden. Somit benötigen wir sowohl den Registersatz als auch die ALU aus Abbildung 5.7.

Darüber hinaus benötigen wir eine Einheit zur **Vorzeichenerweiterung (sign-extend)** des 16-Bit-Offset-Felds im Befehl auf einen vorzeichenbehafteten 32-Bit-Wert sowie eine Datenspeichereinheit, aus der Daten gelesen und in die Daten geschrieben werden können. In den Datenspeicher werden Daten bei Speicherbefehlen geschrieben. Der Datenspeicher benötigt somit sowohl Lese- als auch Schreibsteuersignale, einen Adresseingang sowie einen Eingang für die Daten, die in den Datenspeicher geschrieben werden. In Abbildung 5.8 sind diese beiden Elemente dargestellt.

Der `beq`-Befehl enthält drei Operanden, zwei Register, die miteinander auf Gleichheit verglichen werden, und ein 16-Bit-Offset zum Berechnen der **Sprungzieladresse (branch target address)** relativ zur Sprungbefehladresse. Dieser Befehl hat die Form `beq $t1, $t2, offset`. Um diesen Befehl zu implementieren, müssen wir die Sprungzieladresse berechnen, indem wir das vorzeichenerweiterte Offset-Feld



Vorzeichenerweiterung (sign-extend) Vergrößerung eines Datenelements durch Wiederholen des höchstwertigen Vorzeichenbits des ursprünglichen Datenelements in die höchstwertigen Bits des größeren Zieldatenelements.

Sprungzieladresse (branch target address) Die in einem Sprung angegebene Adresse, die zum neuen Befehlszählerwert wird, wenn der Sprung ausgeführt wird. In der MIPS-Architektur ergibt sich das Sprungziel aus der Summe des Offset-Feldes des Befehls und der Adresse des nächsten Befehls nach dem Sprung.

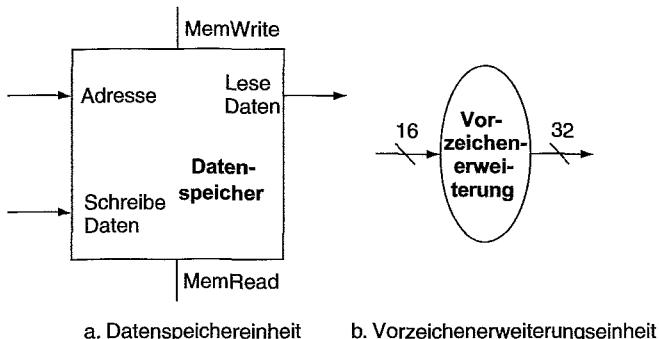


Abb. 5.8 Die beiden Einheiten, die neben dem Registersatz und der ALU aus Abbildung 5.7 zum Implementieren von Lade- und Speicherbefehlen benötigt werden, sind die Datenspeichereinheit und die Vorzeichenerweiterungseinheit. Die Speichereinheit ist ein Schaltwerk mit Eingängen für die Adresse und die Schreibdaten und mit einem Ausgang für das Leseergebnis. Für die Lese- und Schreibvorgänge gibt es getrennte Steuersignale, obgleich bei einem gegebenen Takt jeweils nur eines der beiden Signale auf logisch 1 gesetzt werden kann. Die Speichereinheit erfordert ein Lesesignal, da es (wie wir in Kapitel 7 sehen werden) im Gegensatz zum Registersatz bei der Speichereinheit zu Fehlern kommen kann, wenn der Wert einer ungültigen Adresse gelesen wird. Die Vorzeichenerweiterungseinheit weist einen 16-Bit-Eingang auf, der auf ein 32-Bit-Ergebnis vorzeichenverweitert am Ausgang abgebildet wird (siehe Kapitel 3). Wir setzen voraus, dass der Datenspeicher bei Schreibvorgängen flankengesteuert ist. Gebräuchliche Speicherchips verwenden für Schreibvorgänge tatsächlich ein Schreibaktivierungssignal (Write Enable). Obwohl das Schreibaktivierungssignal nicht flankengesteuert ist, kann unser flankengesteuerter Entwurf problemlos so angepasst werden, dass er in echten Speicherchips realisiert werden kann. Auf CD in Appendix B.8 finden Sie weitere Informationen zur Funktionsweise von echten Speicherchips.



des Befehls zum Befehlszähler hinzuzaddieren. Bei der Definition von Sprungbefehlen (siehe Kapitel 2) gibt es zwei Dinge zu beachten:

- Die Befehlssatzarchitektur gibt vor, dass die Basis für die Berechnung der Sprungadresse die Adresse des Befehls nach dem Sprung ist. Da im Datenpfad zum Holen des Befehls zum Wert des Befehlszählers ohnehin 4 addiert wird (die Adresse des nächsten Befehls), kann dieser Wert leicht als Basis für die Berechnung der Sprungzieladresse verwendet werden.
- Die Architektur gibt außerdem vor, dass das Offset-Feld um 2 Bit nach links verschoben wird, so dass es sich um ein Wort-Offset handelt. Durch diese Verschiebung wird der Wirkungsbereich des Offset-Felds um den Faktor vier erweitert.

Um dieses zweite Problem zu berücksichtigen, müssen wir das Offset-Feld um zwei Bit verschieben.

Wir müssen nicht nur die Sprungzieladresse berechnen, sondern auch feststellen, ob der nächste Befehl der in der Reihe folgende Befehl oder der Befehl an der Sprungzieladresse ist. Wenn die Bedingung wahr ist (d.h., die Operanden sind gleich), wird die Sprungzieladresse zum neuen Befehlszählerwert, und wir sagen, dass der **Sprung ausgeführt (branch taken)** wird. Wenn die Operanden nicht gleich sind, wird der aktuelle Befehlszähler durch den inkrementierten Befehlszähler ersetzt (wie bei jedem anderen normalen Befehl). In diesem Fall sagen wir, dass der **Sprung nicht ausgeführt (branch not taken)** wird.

Der Sprungdatenpfad muss somit zwei Operationen durchführen: Er muss die Sprungzieladresse berechnen und die Registerinhalte vergleichen. (Sprünge wirken sich auch auf den Befehlsholteil des Datenpfads aus, wie wir in Kürze sehen werden.) Aufgrund der komplexen Struktur von Sprüngen, ist in Abbildung 5.9 die Struktur des Datenpfadsegments dargestellt, das Sprünge verarbeitet. Um Sprungzieladressen berechnen zu können, enthält der Sprungdatenpfad eine Vorzeichenerweiterungseinheit wie die in Abbildung 5.8 sowie einen Addierer. Zum Durchführen des Vergleichs

Sprung ausgeführt (branch taken) Ein Sprung, bei dem die Sprungbedingung erfüllt ist und das Sprungziel zum Befehlszählerwert wird. Alle unbedingten Sprünge sind ausgeführte Sprünge.

Sprung nicht ausgeführt (branch not taken) Ein Sprung, bei dem die Sprungbedingung nicht erfüllt ist und die Adresse des Befehls zum Befehlszählerwert wird, der dem Sprung als nächster folgt.

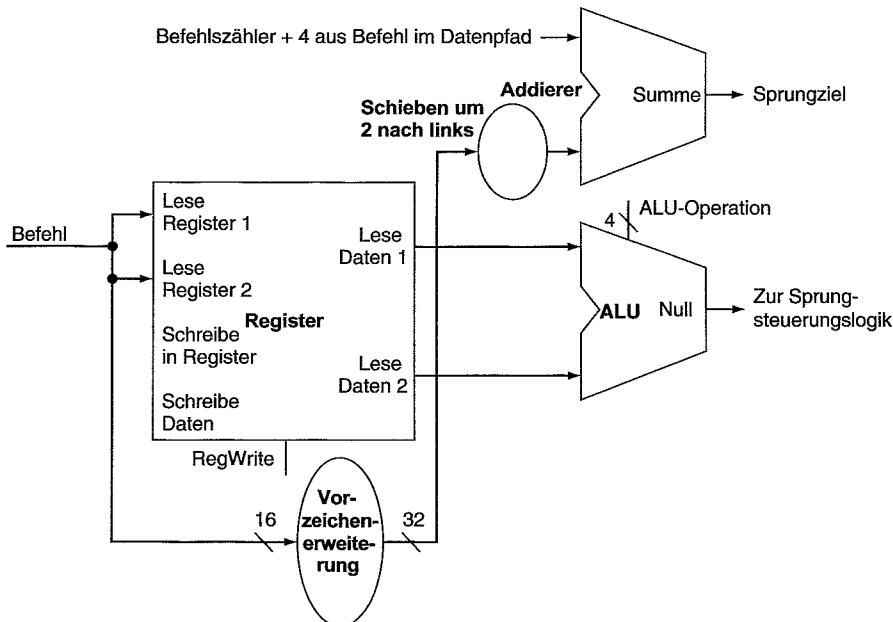


Abb. 5.9 Der Datenpfad für einen Sprung verwendet die ALU zum Auswerten der Sprungbedingung und einen separaten Addierer zum Berechnen des Sprungziels als die Summe aus inkrementiertem Befehlszähler und die um zwei Bit nach links verschobenen vorzeichenerweiterten unteren 16 Bits des Befehls (Sprung-Displacement). Die mit Verschieben um 2 nach links gekennzeichnete Einheit stellt die Übertragung der Signale zwischen Eingang und Ausgang dar, bei der 00_B zum niedrigstwertigen Ende des vorzeichenerweiterten Offset-Felds hinzugefügt wird. Eine Verschiebehardware wird nicht benötigt, da die Anzahl der verschobenen Bits konstant ist. Da wir wissen, dass das Offset-Feld von 16 Bits vorzeichenerweitert wurde, gehen durch die Verschiebung nur Vorzeichenbits verloren. Mithilfe einer Steuerlogik wird anhand des Zero-Ausgangs der ALU entschieden, ob der Befehlszählerwert durch einen inkrementierten Befehlszählerwert oder ein Sprungziel ersetzt wird.

müssen wir den in Abbildung 5.7 dargestellten Registersatz verwenden, um die beiden Registeroperanden bereitzustellen zu können (obgleich wir nicht in den Registersatz schreiben müssen). Zudem kann der Vergleich mithilfe der in **Appendix B** entworfenen ALU durchgeführt werden. Da diese ALU ein Ausgangssignal bereitstellt, das angibt, ob das Ergebnis 0 ist, können wir die zwei Registeroperanden an die ALU senden, wobei das Steuersignal so gesetzt ist, dass eine Subtraktion durchgeführt wird. Wenn das Zero-Signal am Ausgang der ALU-Einheit auf logisch 1 gesetzt ist, wissen wir, dass die beiden Werte gleich sind. Der Zero-Ausgang zeigt zwar immer an, wenn das Ergebnis 0 ist. Dennoch werden wir diesen Ausgang nur zum Implementieren der Gleichheitsprüfung bei Sprüngen verwenden. Später werden wir ausführlich darlegen, wie die Steuersignalleitungen der ALU für die Verwendung im Datenpfad verbunden werden.

Der Sprungbefehl wird durch Ersetzen der unteren 28 Bits des Befehlszählers durch die unteren, um zwei Bit nach links verschobenen 26 Bits des Befehls realisiert. Diese Verschiebung wird, wie in Kapitel 2 beschrieben, durch Verknüpfen von 00 mit dem Sprung-Offset erzielt.

Vertiefung: Im MIPS-Befehlssatz werden Sprünge verzögert, was bedeutet, dass der Befehl direkt nach dem Sprung immer ausgeführt wird, unabhängig davon, ob die Sprungbedingung erfüllt ist oder nicht. Wenn die Bedingung nicht erfüllt ist, sieht die Ausführung wie ein normaler Sprung aus. Wenn die Bedingung er-



Verzögerter Sprung (*delayed branch*) Ein Sprung, bei dem der Befehl, der dem Sprung direkt folgt, immer ausgeführt wird, unabhängig davon, ob die Sprungbedingung erfüllt ist oder nicht.

füllt ist, führt ein verzögerter Sprung zunächst den Befehl aus, der dem Sprung in einer sequenziellen Befehlsfolge direkt folgt und springt dann zur angegebenen Sprungzieladresse. Verzögerte Sprünge sind wegen der Art und Weise, wie sich das Pipelining auf Sprünge auswirkt, sinnvoll (siehe Abschnitt 6.6). Der Einfachheit halber lassen wir verzögerte Sprünge in diesem Kapitel außer Acht und implementieren einen nicht verzögerten `beq`-Befehl.

Entwurf eines einfachen Datenpfads

Nun, da wir die für die einzelnen Befehlsklassen erforderlichen Komponenten eines Datenpfads kennen, können wir diese zu einem einfachen Datenpfad zusammenfügen und die Implementierung mit der Steuerung vervollständigen. Der einfachste Datenpfad würde versuchen, alle Befehle in einem Taktzyklus auszuführen. Das bedeutet, dass keine Ressource im Datenpfad mehr als einmal pro Befehl verwendet werden kann, so dass jedes Element, das mehr als einmal benötigt wird, mehrfach vorhanden sein muss. Daher müssen wir Befehlsspeicher und Datenspeicher voneinander trennen. Auch wenn einige Funktionseinheiten mehrfach vorhanden sein müssen, so können doch viele der Elemente von unterschiedlichen Befehlen gemeinsam genutzt werden.

Damit ein Element im Datenpfad von zwei verschiedenen Befehlsklassen gemeinsam genutzt werden kann, müssen wir mithilfe eines Multiplexers mehrere Verbindungen zum Eingang eines Elements zulassen und mithilfe eines Steuersignals zwischen den verschiedenen Eingängen auswählen.

BEISPIEL

Aufbau eines Datenpfads

Die Operationen arithmetisch-logischer Befehle (oder R-Befehle) und der Datenpfad von Speicherbefehlen sind einander recht ähnlich und unterscheiden sich im Wesentlichen nur durch folgende Punkte:

- Die arithmetisch-logischen Befehle verwenden die ALU mit den Eingangssignalen von den beiden Registern. Die Speicherbefehle können die ALU auch zur Berechnung von Adressen verwenden, wenngleich der zweite Eingang das vorzeichenweitere 16-Bit-Offset-Feld aus dem Befehl ist.
- Der in einem Zielregister gespeicherte Wert stammt bei einem R-Befehl von der ALU und bei einem Ladebefehl aus dem Speicher.

Zeigen Sie, wie für den Ausführungsteil des Speicherreferenzbefehls und des arithmetisch-logischen Befehls ein Datenpfad erstellt wird, der nur einen Registersatz und eine ALU für beide Befehlstypen verwendet, und verwenden Sie dabei so viele Multiplexer wie nötig.

ANTWORT

Um einen Datenpfad mit nur einem Registersatz und einer ALU zu entwerfen, müssen wir zwei verschiedene Quellen für den zweiten ALU-Eingang sowie zwei verschiedene Quellen für die im Registersatz gespeicherten Daten verwenden. Somit wird ein Multiplexer an den ALU-Eingang gelegt und ein weiterer an den Dateneingang am Registersatz. In Abbildung 5.10 ist der Ausführungsteil des gesamten Datenpfads dargestellt.

Nun können wir alle Teile zu einem einfachen Datenpfad für die MIPS-Architektur zusammenfügen, indem wir den Datenpfad für das Holen des Befehls (Abbildung 5.6), den Datenpfad von R-Befehlen und Speicherbefehlen (Abbildung 5.10) und den Datenpfad für Sprünge (Abbildung 5.9) hinzufügen. In Abbildung 5.11 ist der Datenpfad

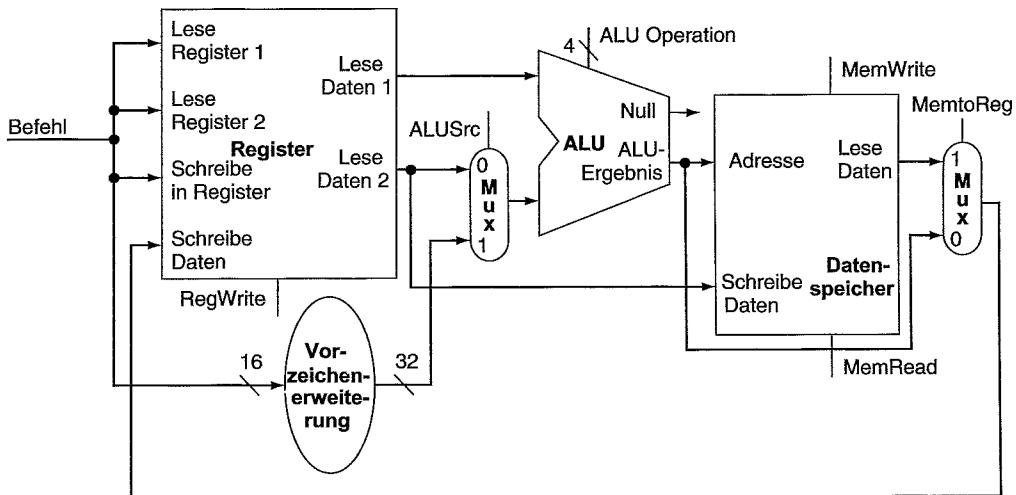


Abb. 5.10 Der Datenpfad für die Speicherbefehle und die R-Befehle. Dieses Beispiel zeigt, wie ein einfacher Datenpfad aus den in Abbildung 5.7 und 5.8 dargestellten Teilen mithilfe von Multiplexern zusammengesetzt werden kann. Wie im Beispiel beschrieben, werden zwei Multiplexer benötigt.

dargestellt, den wir durch Zusammensetzen der einzelnen Teile erhalten. Der Sprungbefehl verwendet die Haupt-ALU für den Vergleich der Registeroperanden, so dass wir den Addierer in Abbildung 5.9 zum Berechnen der Sprungzieladresse beibehalten müssen. Ein weiterer Multiplexer ist erforderlich, um entweder die in der Reihenfolge folgende Befehlsadresse (Befehlszählerwert + 4) oder die Sprungzieladresse zum Schreiben in den Befehlszähler auszuwählen.

Nun, da wir diesen einfachen Datenpfad erstellt haben, können wir ein Steuerwerk ergänzen. Das Steuerwerk muss Eingangssignale aufnehmen und für jedes Schaltwerk ein Schreibsignal, für jeden Multiplexer ein Auswahlsteuersignal und das Signal für die ALU-Steuerung erstellen können. Die ALU-Steuerung unterscheidet sich von den anderen Elementen in mehreren Punkten und es empfiehlt sich, dieses Element vor den anderen Elementen der Steuereinheit zu entwerfen.

Welche der folgenden Aussagen trifft auf einen Ladebefehl zu?

1. MemtoReg muss so gesetzt werden, dass Daten aus dem Speicher an den Registersatz gesendet werden.
2. MemtoReg muss so gesetzt werden, dass das richtige Registerziel an den Registersatz gesendet wird.
3. Wir kümmern uns nicht um das Setzen von MemtoReg.



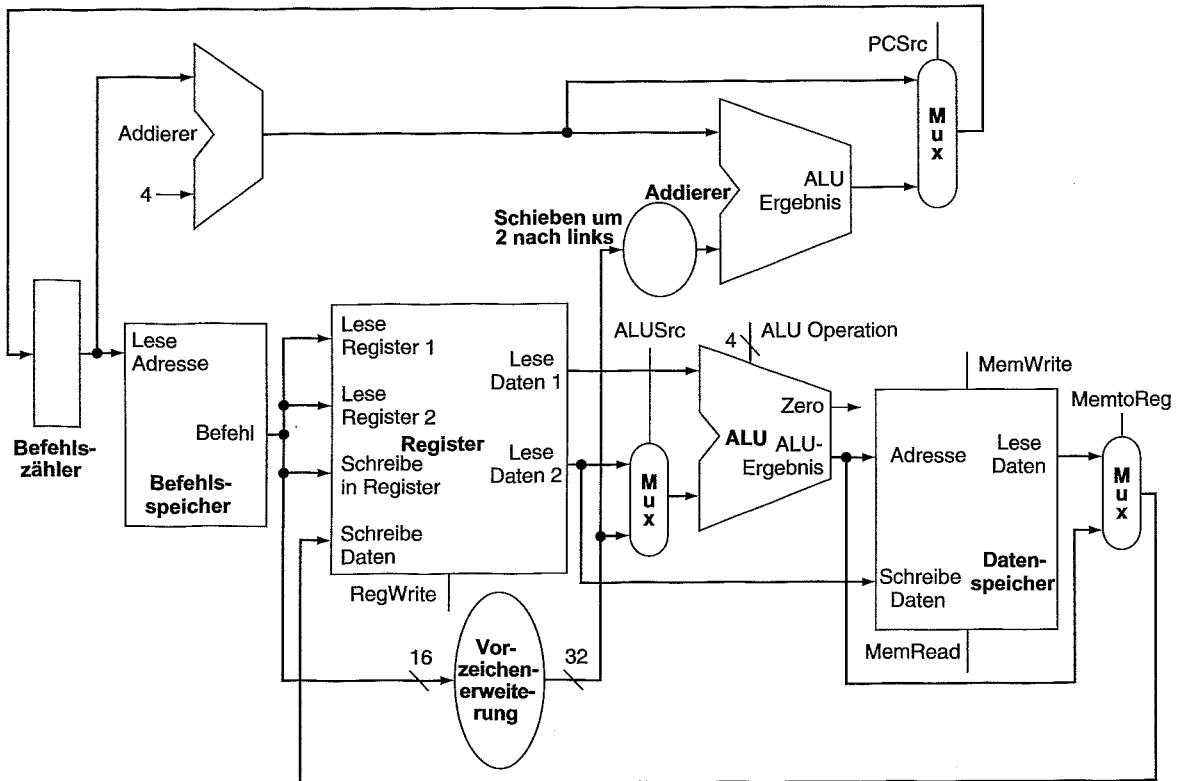


Abb. 5.11 Der einfache Datenpfad für die MIPS-Architektur besteht aus den Elementen, die von den unterschiedlichen Befehlsklassen benötigt werden. Dieser Datenpfad kann die einfachen Befehle (load word, store word, ALU-Operationen und Sprünge) in einem einzigen Taktzyklus ausführen. Für die Integration von Verzweigungen ist ein weiterer Multiplexer erforderlich. Die Unterstützung von unbedingten Sprüngen wird später ergänzt.

5.4

Eine einfache Implementierungsmethode

In diesem Abschnitt werden wir das untersuchen, was vielleicht als die einfachste Implementierung unseres MIPS-Befehlssatzes betrachtet werden kann. Wir erstellen diese einfache Implementierung mit dem Datenpfad des letzten Abschnitts und fügen eine einfache Steuerfunktion ein. Diese einfache Implementierung umfasst die Befehle `load word (lw)`, `store word (sw)`, `branch on equal (beq)` sowie die arithmetisch-logischen Befehle `add`, `sub`, `and`, `or` und `set on less than`. Wir werden den Entwurf später mit einem Sprungbefehl (`j`) erweitern.

Die ALU-Steuerung



Wie auf CD in **Appendix B** zu sehen ist, verfügt die ALU über vier Steuereingänge. In diesem Befehlssatz werden nur sechs der 16 möglichen Eingangskombinationen verwendet. Die in **Appendix B** beschriebene MIPS-ALU weist die folgenden sechs Kombinationen auf:

ALU-Steuerleitungen	Funktion
0000	and
0001	or
0010	add
0110	subtract
0111	set on less than
1100	nor

Je nach Befehlsklasse muss die ALU eine dieser ersten fünf Funktionen ausführen. (nor wird für andere Bereiche des MIPS-Befehlssatzes benötigt.) Bei `load-word`- und `store-word`-Befehlen verwenden wir die ALU zum Berechnen der Speicheradresse durch Addition. Bei R-Befehlen muss die ALU je nach dem Wert des 6-Bit-funct-Felds in den niederwertigen Bits des Befehls (siehe Kapitel 2) eine der fünf Aktionen (and, or, subtract, add oder set on less than) ausführen. Bei `branch on equal` muss die ALU eine Subtraktion durchführen.

Wir können den 4 Bit breiten ALU-Steuereingang mit einer kleinen Steuereinheit realisieren, die als Eingänge das funct-Feld des Befehls und ein 2 Bit breites Steuerfeld verwendet. Dieses Steuerfeld wird als ALUOp bezeichnet und gibt an, ob es sich bei der durchzuführenden Operation um eine Addition (00) für Lade- und Speicherbefehle oder um eine Subtraktion (01) für den `beq`-Befehl handelt oder ob diese durch die im funct-Feld (10) codierte Operation bestimmt wird. Beim Ausgang der ALU-Steuereinheit handelt es sich um ein 4 Bit breites Signal, das die ALU durch die Generierung einer der in der Tabelle dargestellten 4-Bit-Kombinationen direkt steuert.

In Tabelle 5.1 ist dargestellt, wie die ALU-Steuereingänge auf der Grundlage des 2 Bit breiten ALUOp-Steuerfelds und des 6 Bit breiten Funktionscodes festgelegt werden. Der Vollständigkeit halber ist die Beziehung zwischen den ALUOp-Bits und dem Opcode des Befehls ebenfalls dargestellt. Weiter unten in diesem Kapitel werden wir sehen, wie die ALUOp-Bits aus der Hauptsteuereinheit generiert werden.

Dieser Stil, bei dem mehrere Stufen der Decodierung (d.h., die Hauptsteuereinheit generiert die ALUOp-Bits, die dann als Eingangssignale für die ALU-Steuerung ver-

Tab. 5.1 Wie die ALU-Steuerbits gesetzt werden, hängt von den ALUOp-Steuerbits und den unterschiedlichen Funktionscodes für den R-Befehl ab. Der in der ersten Spalte angegebene Opcode bestimmt die Festlegung der ALUOp-Bits. Der gesamte Code ist in Binärform dargestellt. Wenn der ALUOp-Code 00 oder 01 ist, hängt die gewünschte ALU-Aktion nicht vom Funktionscodefeld ab. In diesem Fall spielt der Wert des Funktionscodes keine Rolle („Don't-care-Term“) und das funct-Feld wird als XXXXXX dargestellt. Wenn der ALUOp-Wert 10 ist, wird der Funktionscode zum Setzen des ALU-Steuereingangs verwendet.

Opcode des Befehls	ALUOp	Befehlsoperation	funct-Feld	Gewünschte ALU-Aktion	ALU-Steuer-eingang
LW	00	load word	XXXXXX	Addition	0010
SW	00	store word	XXXXXX	Addition	0010
Branch on equal	01	branch on equal	XXXXXX	Subtraktion	0110
R-Befehl	10	add	100000	Addition	0010
R-Befehl	10	subtract	100010	Subtraktion	0110
R-Befehl	10	AND	100100	UND	0000
R-Befehl	10	OR	100101	ODER	0001
R-Befehl	10	set on less than	101010	kleiner als	0111

Tab. 5.2 Die Wahrheitstabelle für die drei ALU-Steuerbits (Operation genannt). Als Eingänge dienen die ALUOp-Bits und das Funktionscodefeld. Es sind nur die Einträge dargestellt, für die das ALU-Steuersignal logisch 1 ist. Zudem wurden einige Don't-care-Einträge eingefügt. ALUOp verwendet beispielsweise den Code 11 nicht, so dass die Wahrheitstabelle anstelle von 10 und 01 die Einträge 1X und X1 enthalten kann. Wenn das funct-Feld verwendet wird, sind die ersten beiden Bits (F5 und F4) dieser Befehle immer 10, so dass es sich bei diesen um Don't-care-Terme handelt, die in der Wahrheitstabelle durch XX ersetzt werden.

ALUOp		funct-Feld						Operation
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0	
0	0	X	X	X	X	X	X	0010
X	1	X	X	X	X	X	X	0110
1	X	X	X	0	0	0	0	0010
1	X	X	X	0	0	1	0	0110
1	X	X	X	0	1	0	0	0000
1	X	X	X	0	1	0	1	0001
1	X	X	X	1	0	1	0	0111

wendet werden, die die eigentlichen Signale zum Steuern der ALU generiert) verwendet werden, stellt eine gebräuchliche Implementierungsmethode dar. Durch die Verwendung mehrerer Steuerstufen kann die Hauptsteuereinheit verkleinert werden. Wenn mehrere kleine Steuereinheiten verwendet werden, kann zudem möglicherweise die Geschwindigkeit der Steuereinheit erhöht werden. Optimierungen dieser Art sind wichtig, da die Steuereinheit häufig einen Engpass hinsichtlich der Leistung darstellt.

Es gibt mehrere Möglichkeiten, die Abbildung des 2 Bit breiten ALUOp-Felds und des 6 Bit breiten funct-Felds auf die drei ALU-Operationssteuerungsbits zu implementieren. Da nur wenige der 64 möglichen Werte des funct-Felds von Interesse sind und das funct-Feld nur verwendet wird, wenn die ALUOp-Bits gleich 10 sind, können wir einen kleinen Logikbaustein verwenden, der den Teil der möglichen Werte erkennt und dafür sorgt, dass die ALU-Steuerbits richtig gesetzt werden.

Beim Entwurf dieser Logik ist es hilfreich, eine Wahrheitstabelle für die betreffenden Kombinationen des Funktionscodefelds und der ALUOp-Bits wie in Tabelle 5.2 zu erstellen. Anhand dieser Tabelle wird deutlich, wie die 4 Bit breite ALU-Steuerung in Abhängigkeit dieser beiden Eingangsfelder gesetzt wird. Da die vollständige Wahrheitstabelle sehr umfangreich ist ($2^8 = 256$ Einträge) und der ALU-Steuerungswert für viele dieser Eingangskombinationen keine Rolle spielt, sind nur die Einträge der Wahrheitstabelle dargestellt, für die die ALU-Steuerung einen bestimmten Wert annehmen muss. In diesem ganzen Kapitel werden immer jeweils nur die Einträge der Wahrheitstabelle dargestellt, die auf logisch 1 gesetzt sein müssen. Die Einträge, die logisch 0 oder Don't-care-Terme sind, werden nicht dargestellt. (Diese Vorgehensweise hat einen Nachteil, der auf CD in **Appendix C.2** beschrieben wird.)

Da der Wert einiger Eingänge häufig keine Rolle spielt, werden außerdem **Don't-care-Terme** verwendet, um die Tabelle möglichst kompakt zu halten. Ein Don't-care-Term in dieser Wahrheitstabelle (dargestellt durch ein X in einer Eingangsspalte) gibt an, dass der Ausgang nicht vom Wert des zu dieser Spalte gehörenden Eingangs abhängt. Wenn die ALUOp-Bits wie in der ersten Zeile der Tabelle 5.2 beispielsweise 00 sind, setzen wir die ALU-Steuerung unabhängig vom Funktionscode immer auf 010. In diesem Fall sind die Funktionscodeeingänge in dieser Zeile der Wahrheitstabelle Don't-care-Terme. Später werden wir Beispiele für eine andere Art von Don't-care-Term kennen lernen. Wenn Sie mit dem Begriff der Don't-care-Terme nicht vertraut sind, finden Sie entsprechende Informationen in **Appendix B**.

Don't-care-Term Ein Element einer logischen Funktion, bei dem der Ausgang nicht von den Werten aller Eingänge abhängt. Don't-care-Terme können auf unterschiedliche Art und Weise angegeben werden.



Don't-care-Term Ein Element einer logischen Funktion, bei dem der Ausgang nicht von den Werten aller Eingänge abhängt. Don't-care-Terme können auf unterschiedliche Art und Weise angegeben werden.



Feld	0	rs	rt	rd	shamt	funct
Bit-Positionen	31:26	25:21	20:16	15:11	10:6	5:0
a. R-Befehl						
Feld	35 oder 43	rs	rt		Address	
Bit-Positionen	31:26	25:21	20:16		15:0	
b. Lade- oder Speicherbefehl						
Feld	4	rs	rt		Address	
Bit-Positionen	31:26	25:21	20:16		15:0	
c. Sprungbefehl						

Abb. 5.12 Für die drei Befehlsklassen (R-Befehl, Lade- und Speicherbefehl, Sprung) werden zwei verschiedene Befehlsformate verwendet. Für die Sprungbefehle wird ein anderes Format verwendet, das weiter unten beschrieben wird. (a) Befehlsformat für R-Befehle, die alle den Opcode 0 aufweisen. Diese Befehle enthalten drei Registeroperanden: rs, rt und rd. Die Felder rs und rt bezeichnen Quellregister und rd ist das Zielregister. Die ALU-Funktion befindet sich im funct-Feld und wird von der im vorherigen Abschnitt beschriebenen ALU-Steuerung entschlüsselt. Die R-Befehle, die implementiert werden, sind add, sub, and, or und slt. Das shamt-Feld wird nur für Schiebebefehle verwendet. In diesem Kapitel werden wir darauf nicht weiter eingehen. (b) Befehlsformat für Ladebefehle (Opcode = 35_D) und Speicherbefehle (Opcode = 43_D). Das Register rs ist das Basisregister, das aufsummiert mit dem 16-Bit-Adressfeld die Speicheradresse ergibt. Bei Ladebefehlen ist rt das Zielregister für den geladenen Wert. Bei Speicherbefehlen ist rt das Quellregister, dessen Wert im Speicher gespeichert werden soll. (c) Befehlsformat für branch on equal (Opcode = 4). Die Register rs und rt sind die Quellregister, die auf Gleichheit überprüft werden. Das 16-Bit-Adressfeld wird vorzeichenweiter, geschoben und zum Befehlszähler addiert und ergibt so die Sprungzieladresse.

Wenn die Wahrheitstabelle erstellt ist, kann sie optimiert und anschließend in Gatter umgesetzt werden. Dies ist ein rein mechanischer Vorgang. Daher werden wir diese letzten Schritte nicht hier, sondern auf CD in **Appendix C.2** beschreiben.



Entwurf der Hauptsteuereinheit

Nun, da wir beschrieben haben, wie eine ALU entworfen wird, die den Funktionscode und ein 2 Bit breites Signal als Steuereingänge verwendet, können wir zur Betrachtung der restlichen Steuerung zurückkehren. Beginnen wird damit, die Felder eines Befehls und die Steuerleitungen zu identifizieren, die für den in Abbildung 5.11 entworfenen Datenpfad erforderlich sind. Um zu verstehen, wie die Felder eines Befehls mit dem Datenpfad verbunden werden, ist es hilfreich, sich die Formate der drei Befehlsklassen in Erinnerung zu rufen: R-Befehle, Sprünge und Lade-/Speicherbefehle. In Abbildung 5.12 sind diese Formate dargestellt.

Bei diesem Befehlsformat, auf das wir uns hier beziehen, gibt es einige wichtige Dinge zu erläutern:

- Das Op-Feld, auch als **Opcode** bezeichnet, befindet sich immer in den Bits 31:26. Wir bezeichnen dieses Feld Op[5:0].
- Die beiden zu lesenden Register werden immer durch die Felder rs und rt an den Positionen 25:21 und 20:16 angegeben. Dies gilt für R-Befehle, für Branch-equal-Befehle und für Speicherbefehle.
- Das Basisregister für Lade- und Speicherbefehle befindet sich immer an den Bit-Positionen 25:21 (rs).
- Der 16-Bit-Offset für branch-on-equal-, Lade- und Speicherbefehle befindet sich immer in den Positionen 15:0.

Opcode Auch Operationscode genannt. Das Feld, das die Operation und das Format eines Befehls angibt.

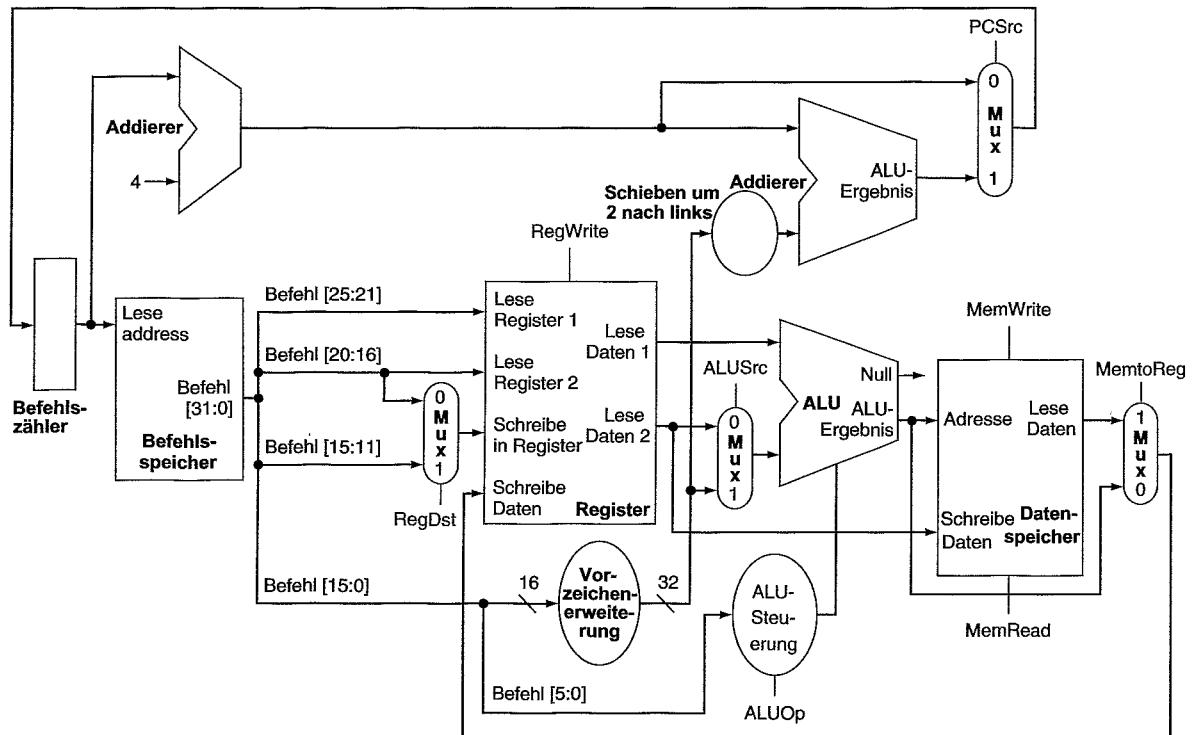


Abb. 5.13 Der Datenpfad aus Tabelle 5.1 mit allen erforderlichen Multiplexern und allen notwendigen Steuerleitungen. Die Steuerleitungen sind farblich hervorgehoben. Auch der ALU-Steuerblock ist hier enthalten. Für den Befehlszähler ist keine Schreibsteuerung erforderlich, da er einmal am Ende jedes Taktzyklus beschrieben wird. Die Logik für die Steuerung der Sprünge bestimmt, ob der inkrementierte Befehlszählerwert oder ob die Sprungzieladresse in den Befehlszähler geschrieben wird.

- Das Zielregister befindet sich an einer von zwei möglichen Stellen. Bei einem Ladebefehl befindet es sich in den Bit-Positionen 20:16 (rt), während es sich bei R-Befehlen an den Bit-Positionen 15:11 (rd) befindet. Daher müssen wir einen Multiplexer verwenden, um auszuwählen, welches Feld des Befehls verwendet wird, um die Adresse des Registers anzugeben, in das ein Wert geschrieben wird.

Mithilfe dieser Informationen können wir den einfachen Datenpfad mit den Befehlsmarken und einem weiteren Multiplexer (für den Schreibe-in-Register-Eingang des Registersatzes) ergänzen. In Abbildung 5.13 sind diese Ergänzungen sowie der ALU-Steuerblock, die Schreibsignale für Schaltwerke, das Lesesignal für den Datenspeicher und die Steuersignale für die Multiplexer dargestellt. Da alle Multiplexer zwei Eingänge aufweisen, benötigen sie alle genau eine Steuerleitung.

In Abbildung 5.13 sind sieben Ein-Bit-Steuerleitungen sowie das 2-Bit-ALUOp-Steuersignal dargestellt. Wir haben bereits festgelegt, wie das ALUOp-Steuersignal funktioniert. Es empfiehlt sich die Funktionsweise der sieben anderen Steuersignale formlos zu definieren, bevor wir bestimmen, wie diese Steuersignale während der Befehlausführung gesetzt werden. In Tabelle 5.3 ist die Funktion dieser sieben Steuerleitungen beschrieben.

Nun, da wir die Funktion der einzelnen Steuersignale kennen, können wir betrachten, wie diese gesetzt werden. Die Steuereinheit kann abhängig vom Opcode-Feld des Befehls bis auf eines alle Steuersignale setzen. Die PCSrc-Steuerleitung stellt die Ausnahme dar. Diese Steuerleitung muss gesetzt werden, wenn es sich um einen branch-on-equal-Befehl handelt (eine Entscheidung, die die Steuereinheit treffen kann) und der Zero-Ausgang der ALU, der für den Gleichheitsvergleich verwendet

Tab. 5.3 Die Auswirkungen der sieben Steuersignale. Wenn die 1-Bit-Steuerleitung zum 2:1-Multiplexer auf logisch 1 gesetzt wird, wählt der Multiplexer den Eingang aus, der dem Wert 1 entspricht. Wenn die Steuerleitung dagegen auf logisch 0 gesetzt wird, wählt der Multiplexer den Zero-Eingang aus. Bei den Zustandselementen dient der Takt als impliziter Eingang und der Takt wird zum Steuern von Schreibvorgängen verwendet. Der Takt wird nie extern an ein Speicherelement angelegt, da dies zu Fehlern beim zeitlichen Ablauf führen kann. (Weitere Informationen zu diesem Problem finden Sie in Appendix B.)

Signal-name	Wirkung, wenn logisch 0	Wirkung, wenn logisch 1
RegDst	Die Adresse des Zielregisters für den Schreibe-in-Register-Befehl wird vom rt-Feld (Bits 20:16) bereitgestellt.	Die Adresse des Zielregisters für den Schreibe-in-Register-Befehl wird vom rd-Feld (Bits 15:11) bereitgestellt.
RegWrite	Keine.	Das Register am Schreibe-in-Register-Eingang wird mit dem Wert am Schreibe-Daten-Eingang beschrieben.
ALUSrc	Der zweite ALU-Operand wird vom zweiten Registersatzausgang (Lese Daten 2) bereitgestellt.	Der zweite ALU-Operand besteht aus den vorzeichenerweiterten, unteren 16 Bits des Befehls.
PCSrc	Der Befehlszählerwert wird durch den Ausgangswert des Addierers ersetzt, der den Befehlszählerwert und 4 addiert.	Der Befehlszählerwert wird durch den Ausgangswert des Addierers ersetzt, der das Sprungziel berechnet.
MemRead	Keine.	Durch den Adresseingang bestimmter Datenspeicherinhalt wird an den Lese-Daten-Ausgang gelegt.
MemWrite	Keine.	Durch den Adresseingang bestimmter Datenspeicherinhalt wird durch den Wert am Schreibe-Daten-Eingang ersetzt.
MemtoReg	Der am Schreibe-Daten-Eingang des Registers angelegte Wert wird von der ALU bereitgestellt.	Der am Schreibe-Daten-Eingang des Registers angelegte Wert wird vom Datenspeicher bereitgestellt.

Tab. 5.4 Die Belegung der Steuerleitungen wird ausschließlich durch die Opcode-Felder des Befehls bestimmt. Die erste Zeile der Tabelle entspricht den R-Befehlen (add, sub, and, or und slt). Bei all diesen Befehlen sind rs und rt die Quellregisterfelder und rd das Zielregisterfeld. Damit ist definiert, wie die Signale ALUSrc und RegDst gesetzt werden. Darüber hinaus schreibt ein R-Befehl zwar Werte in ein Register (RegWrite = 1), liest und beschreibt jedoch keine Speicherstelle. Wenn das Steuersignal für den Sprung auf 0 gesetzt ist, wird der Befehlszähler unbedingt durch Befehlszähler + 4 ersetzt. Andernfalls wird der Befehlszähler durch das Sprungziel ersetzt, wenn der Zero-Ausgang der ALU ebenfalls 1 ist. Das ALUOp-Feld für R-Befehle ist auf 10 gesetzt, um anzugeben, dass die ALU-Steuerung vom funct-Feld generiert werden muss. In der zweiten und dritten Zeile dieser Tabelle ist angegeben, welche Steuersignale für lw und sw gesetzt werden. Diese ALUSrc- und ALUOp-Felder werden zum Berechnen der Adresse gesetzt. MemRead und MemWrite werden zum Durchführen eines Speicherzugriffs gesetzt. Und RegDst und RegWrite werden schließlich für einen Ladebefehl gesetzt, damit das Ergebnis im rt-Register gespeichert wird. Der Sprungbefehl gleicht einer R-Operation, da er die Register rs und rt an die ALU sendet. Das ALUOp-Feld für den Sprung wird für eine Subtraktion (ALU-Steuerung = 01) gesetzt, die für die Überprüfung auf Gleichheit verwendet wird. Das MemtoReg-Feld ist unwichtig, wenn das RegWrite-Signal 0 ist: Da in das Register nicht geschrieben wird, wird der Wert der Daten am Registerdatenschreibport nicht verwendet. Daher wird der Eintrag MemtoReg in den letzten beiden Zeilen der Tabelle auf X für Don't-care gesetzt. Don't-cares können auch in RegDst eingefügt werden, wenn RegWrite 0 ist. Diese Art Don't-care muss vom Entwickler eingefügt werden, da hierzu bekannt sein muss, wie der Datenpfad funktioniert.

Befehl	RegDst	ALUSrc	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUOp0
R-Format	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X	0	X	0	0	0	1	0	1

wird, wahr ist. Um das PCSrc-Signal zu generieren, müssen wir ein Signal von der Steuereinheit, das wir *Branch* nennen, und das Zero-Signal am Ausgang der ALU mit einer UND-Verknüpfung miteinander verknüpfen.

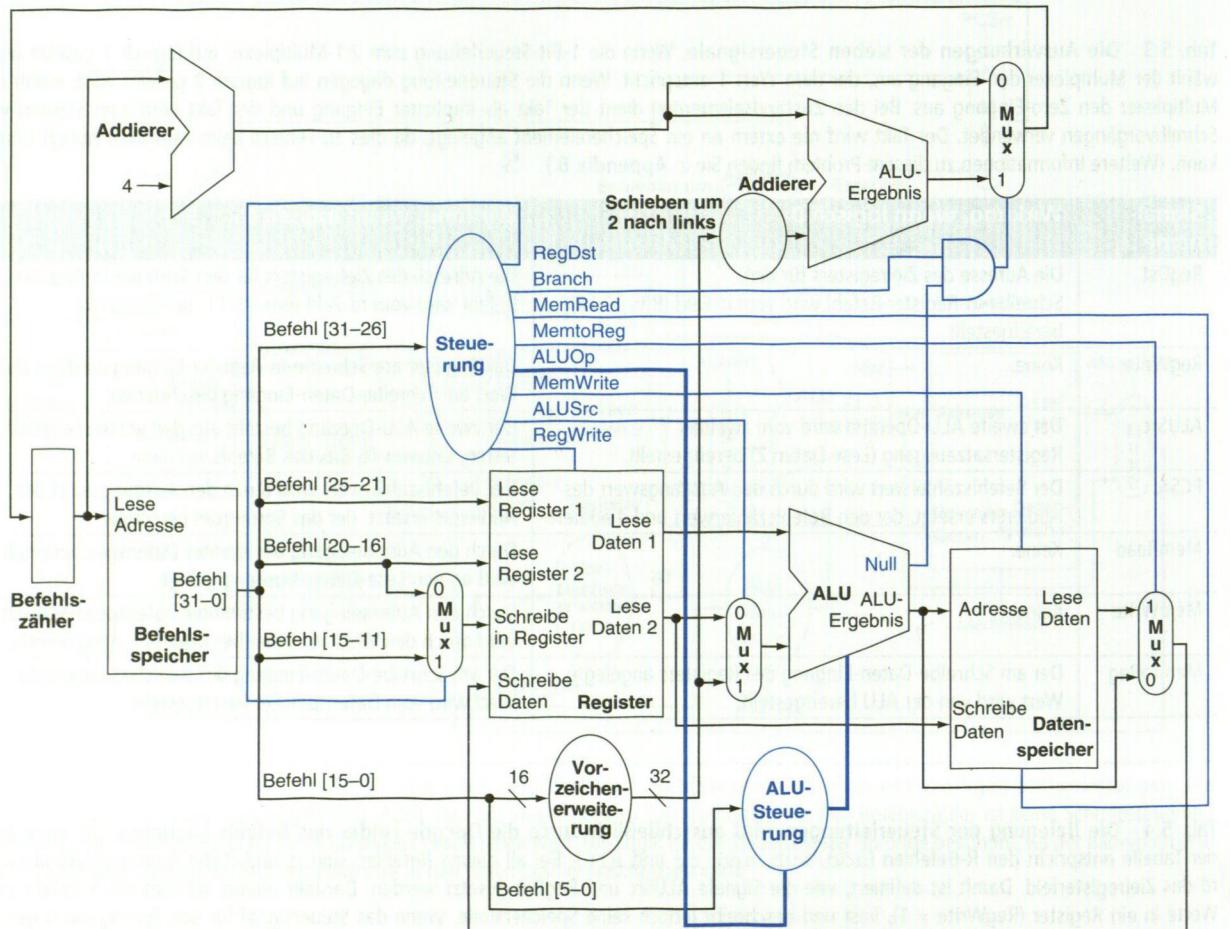


Abb. 5.14 Der einfache Datenpfad mit der Steuereinheit. Die Eingabe zur Steuereinheit ist das 6-Bit-Opcode-Feld aus dem Befehl. Die Ausgänge der Steuereinheit setzen sich aus drei 1-Bit-Signalen zum Steuern von Multiplexern (RegDst, ALUSrc und MemtoReg), drei Signalen zum Steuern von Lese- und Schreibvorgängen im Registersatz und Datenspeicher (RegWrite, MemRead und MemWrite), einem 1-Bit-Signal zum Bestimmen einer möglichen Verzweigung (Branch) und einem 2-Bit-Steuersignal für die ALU (ALUOp) zusammen. Das Signal für die Steuerung der Verzweigung und das Zero-Ausgangssignal von der ALU werden mit einem UND-Gatter verknüpft. Mit dem Ausgangssignal des UND-Gatters wird die Auswahl des nächsten Befehlszählerwerts gesteuert. PCSrc ist nun ein abgeleitetes Signal und wird nicht mehr direkt von der Steuereinheit bereitgestellt. Daher wird der Signalname in den nachfolgenden Abbildungen nicht mehr erscheinen.

Diese neun Steuersignale (sieben aus Tabelle 5.3 und zwei für ALUOp) können nun anhand der sechs Eingangssignale an der Steuereinheit, bei denen es sich um die Opcode-Bits handelt, gesetzt werden. In Abbildung 5.14 ist der Datenpfad mit der Steuereinheit und den Steuersignalen dargestellt.

Bevor wir versuchen, eine Reihe von Gleichungen oder eine Wahrheitstabelle für die Steuereinheit zu schreiben, empfiehlt es sich, die Steuerfunktion formlos zu definieren. Da das Setzen der Steuersignale nur vom Opcode abhängt, legen wir fest, ob das Steuersignal für die einzelnen Opcode-Werte 0, 1 oder Don't-care (X) sein muss. In Tabelle 5.4 ist angegeben, wie die Steuersignale für die einzelnen Opcode-Werte gesetzt werden müssen. Diese Angaben folgen direkt aus den Tabellen 5.1 und 5.3 sowie Abbildung 5.14.

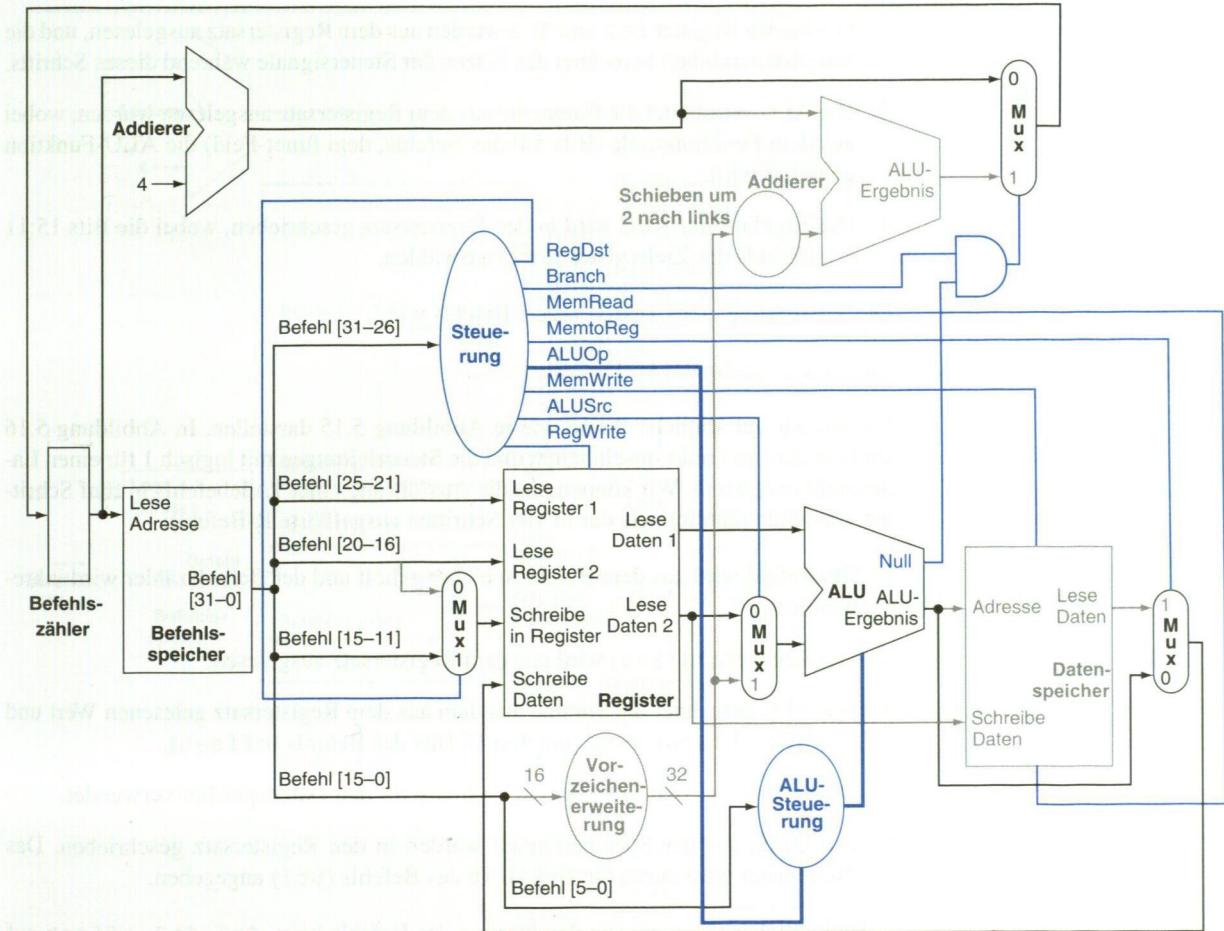


Abb. 5.15 Der Datenpfad bei der Ausführung eines R-Befehls wie `add $t1,$t2,$t3`. Steuerleitungen, Einheiten im Datenpfad und Verbindungen, die aktiv sind, sind farblich hervorgehoben.

Funktionsweise des Datenpfads

Mit den in Tabellen 5.3 und 5.4 enthaltenen Informationen können wir die Logikschaltung der Steuereinheit entwerfen. Zuvor sollten wir jedoch untersuchen, wie die einzelnen Befehle den Datenpfad nutzen. In den nächsten Abbildungen ist der Datenfluss dreier verschiedener Befehlsklassen durch den Datenpfad dargestellt. Die Steuersignale mit logisch 1 und die aktiven Elemente im Datenpfad sind jeweils farblich hervorgehoben. Ein Multiplexer, dessen Steuersignal 0 ist, führt eine eindeutige Aktion aus, auch wenn seine Steuerleitung nicht farblich hervorgehoben ist. Steuersignale mit mehreren Bits sind farblich hervorgehoben, wenn mindestens eines der Signale logisch 1 ist.

In Abbildung 5.15 ist der Datenpfad bei der Ausführung eines R-Befehls wie `add $t1,$t2,$t3` dargestellt. Der gesamte Befehl wird zwar in einem Taktzyklus ausgeführt. Dennoch können wir uns die Ausführung in vier Schritten vorstellen, die in der Reihenfolge des Datenflusses angeordnet sind:

- Der Befehl wird geholt und der Befehlszähler inkrementiert.

2. Die beiden Register \$t2 und \$t3 werden aus dem Registersatz ausgelesen, und die Hauptsteuereinheit berechnet das Setzen der Steuersignale während dieses Schritts.
3. Die ALU verarbeitet die Daten, die aus dem Registersatz ausgelesen wurden, wobei aus dem Funktionscode (Bits 5:0 des Befehls, dem funct-Feld) die ALU-Funktion generiert wird.
4. Das Ergebnis der ALU wird in den Registersatz geschrieben, wobei die Bits 15:11 des Befehls das Zielregister (\$t1) auswählen.

Die Ausführung eines load-word-Befehls wie

```
lw $t1, offset($t2)
```

können wir auf ähnliche Weise wie in Abbildung 5.15 darstellen. In Abbildung 5.16 sind die aktiven Funktionseinheiten und die Steuerleitungen mit logisch 1 für einen Ladenbefehl dargestellt. Wir können uns die Ausführung eines Ladebefehls in fünf Schritten vorstellen (ähnlich wie der in vier Schritten ausgeführte R-Befehl):

1. Ein Befehl wird aus dem Befehlsspeicher geholt und der Befehlszähler wird inkrementiert.
2. Der Registerwert (\$t2) wird aus dem Registersatz ausgelesen.
3. Die ALU berechnet die Summe aus dem aus dem Registersatz gelesenen Wert und den vorzeichenerweiterten, unteren 16 Bits des Befehls (offset).
4. Die Summe aus der ALU wird als Adresse für den Datenspeicher verwendet.
5. Die Daten aus der Speichereinheit werden in den Registersatz geschrieben. Das Zielregister wird durch die Bits 20:16 des Befehls (\$t1) angegeben.

Schließlich können wir die Ausführung des Befehls `beq $t1, $t2, offset` auf ähnliche Weise darstellen. Er funktioniert im Wesentlichen wie ein R-Befehl, wobei der ALU-Ausgang jedoch verwendet wird, um zu bestimmen, ob der Befehlszähler mit Befehlszählerwert + 4 oder mit der Sprungzieladresse geschrieben wird. In Abbildung 5.17 sind die vier Schritte der Ausführung dargestellt:

1. Ein Befehl wird aus dem Befehlsspeicher geholt und der Befehlszähler wird inkrementiert.
2. Die beiden Register \$t1 und \$t2 werden aus dem Registersatz ausgelesen.
3. Die ALU subtrahiert die aus dem Registersatz ausgelesenen Datenwerte. Der Wert Befehlszähler + 4 wird zu den um zwei Stellen nach links verschobenen vorzeichenerweiterten, unteren 16 Bits des Befehls addiert (offset). Daraus ergibt sich die Sprungzieladresse.
4. Mit dem Zero-Ergebnis aus der ALU wird entschieden, welches Addiererergebnis im Befehlszähler gespeichert wird.

Im nächsten Abschnitt werden Maschinen untersucht, die tatsächlich sequenziell arbeiten, d.h. diese Schritte werden jeweils in einem eigenen Taktzyklus ausgeführt.

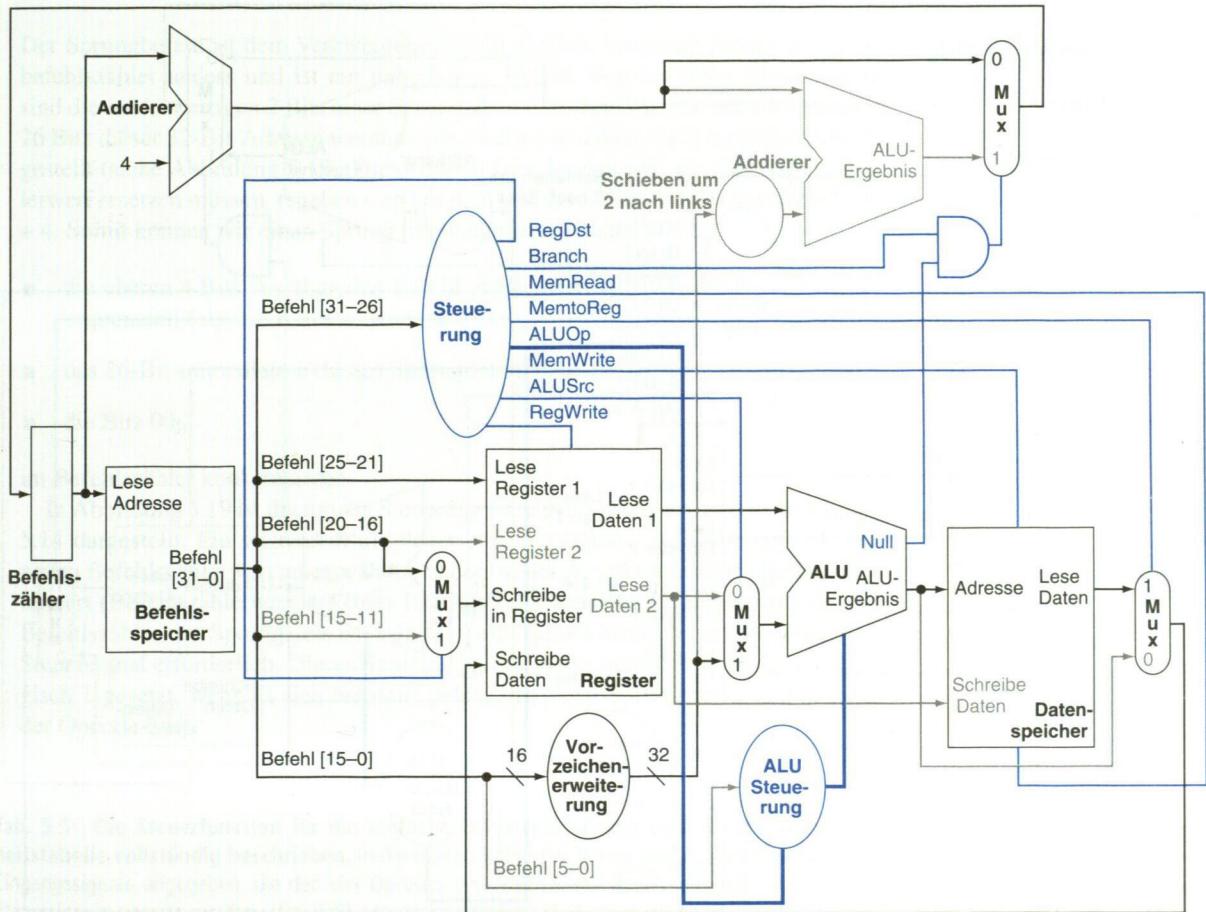


Abb. 5.16 Der Datenpfad bei der Ausführung eines Ladebefehls. Die Steuerleitungen, die Einheiten im Datenpfad und Verbindungen, die aktiv sind, sind farblich hervorgehoben. Ein Speicherbefehl funktioniert sehr ähnlich. Die beiden Befehle unterscheiden sich hauptsächlich dadurch, dass die Speichersteuerung anstelle eines Lesevorgangs einen Schreibvorgang vorgibt, dass der aus dem zweiten Register gelesene Wert zum Speichern der Daten verwendet wird, und dass der Datenspeicherwert nicht in den Registersatz geschrieben wird.

Abschluss des Steuerungsentwurfs

Nun, da wir gesehen haben, wie die Befehle schrittweise abgearbeitet werden, fahren wir mit der Implementierung der Steuerung fort. Die Steuerfunktion kann mithilfe des Inhalts von Tabelle 5.4 präzise definiert werden. Die Ausgänge sind die Steuerleitungen und der Eingang ist das 6-Bit-Opcode-Feld, Op [5:0]. Somit können wir auf der Grundlage der Binärcodierung des Opcodes für jeden Ausgang eine Wahrheitstabelle erstellen.

In Tabelle 5.5 ist die Logikschaltung in der Steuereinheit als eine umfangreiche Wahrheitstabelle dargestellt, in der alle Ausgänge zusammengefasst sind und die Opcode-Bits als Eingänge verwendet werden. Die Tabelle legt die gesamte Steuerfunktion fest, und wir können sie direkt in Gattern implementieren. Dieser letzte Schritt wird auf CD in **Appendix C.2** beschrieben.

Wir werden nun den Sprungbefehl einfügen, um zu zeigen, wie der einfache Datenpfad und die Steuerung für die Bearbeitung anderer Befehle im Befehlssatz erweitert werden können.



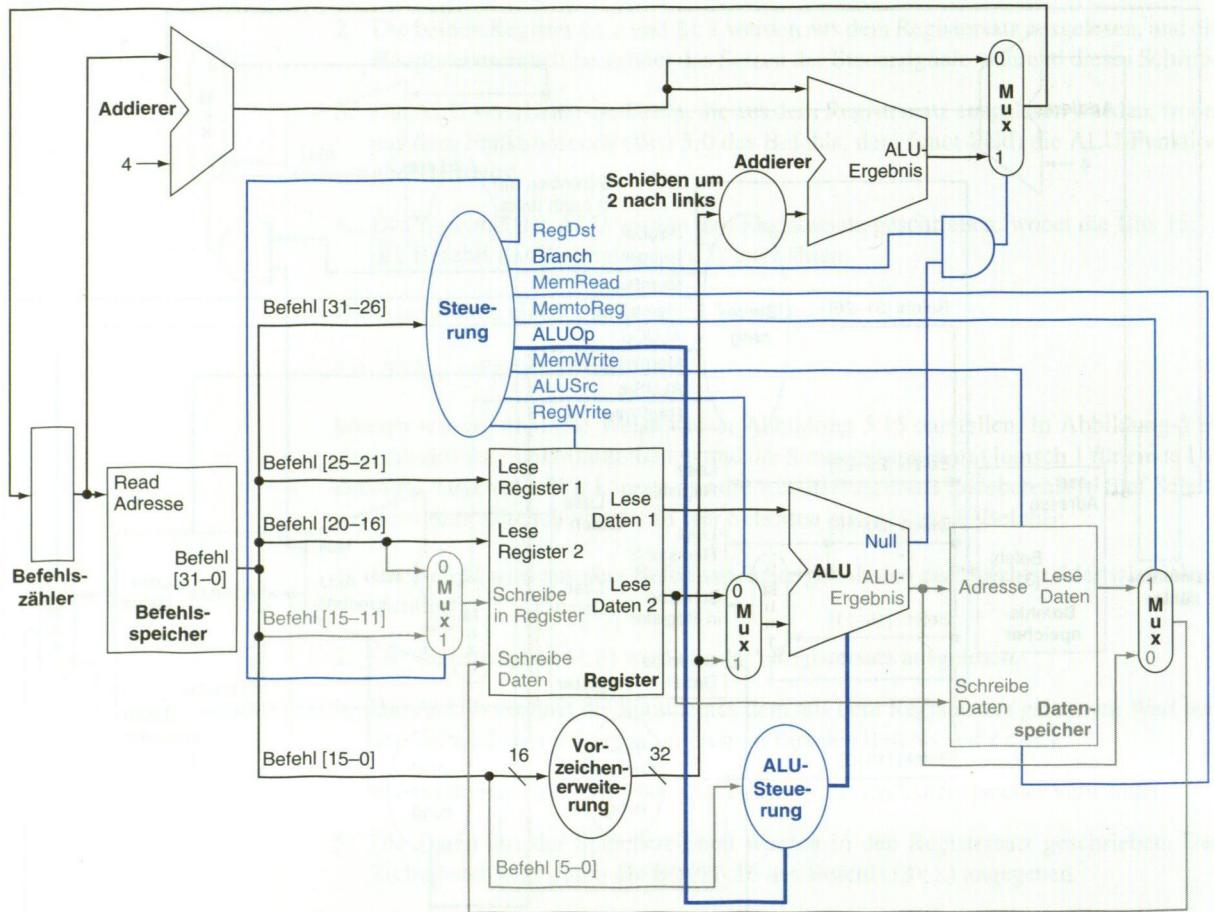


Abb. 5.17 Der Datenpfad bei der Ausführung eines **beq**-Befehls (**branch on equal**). Steuerleitungen, Einheiten im Datenpfad und Verbindungen, die aktiv sind, sind farblich hervorgehoben. Nach dem Durchführen des Vergleichs mithilfe des Registersatzes und der ALU wird der Zero-Ausgang zum Auswählen des nächsten Befehlszählers aus den zwei möglichen Quellen verwendet.

Feld	000010	Adresse
Bit-Position	31:26	25:0

Abb. 5.18 Befehlsformat für den Sprungbefehl (Opcode = 2). Die Zieladresse für einen Sprungbefehl wird durch Konkatenation der oberen 4 Bits des aktuellen Befehlszählerwerts + 4 und dem 26-Bit-Adressfeld im Sprungbefehl und durch Hinzufügen von 00 als den beiden niederwertigsten Bits erstellt.

BEISPIEL

Implementierung von Sprüngen

In Abbildung 5.14 ist die Implementierung vieler Befehle dargestellt, die in Kapitel 2 beschrieben wurden. Eine Befehlsklasse, die bisher noch fehlt, ist der Sprungbefehl. Erweitern Sie den Datenpfad und die Steuerung aus Abbildung 5.14 mit dem Sprungbefehl. Beschreiben Sie, wie neue Steuerleitungen belegt werden müssen.

Der Sprungbefehl ist dem Verzweigungsbefehl ähnlich, berechnet jedoch den Zielbefehlszähler anders und ist ein unbedingter Befehl. Wie bei einer Verzweigung sind die niederwertigen 2 Bits einer Sprungadresse immer 00_B . Die nächst niedrigen 26 Bits dieser 32-Bit-Adresse werden vom 26-Bit-immediate-Feld im Befehl bereitgestellt (siehe Abbildung 5.18). Die oberen 4 Bits der Adresse, die den Befehlszählerwert ersetzen müssen, ergeben sich aus dem Befehlszählerwert des Sprungbefehls + 4. Somit können wir einen Sprung implementieren, indem wir

- die oberen 4 Bits des aktuellen Befehlszählerwerts + 4 (d.h. die Bits 31:28 der sequenziell folgenden Befehlsadresse),
- das 26-Bit-immediate-Feld des Sprungbefehls und
- die Bits 00_B

im Befehlszähler konkatenieren.

In Abbildung 5.19 ist die um die Steuerung für einen Sprung erweiterte Abbildung 5.14 dargestellt. Ein weiterer Multiplexer wird verwendet, um die Quelle für den neuen Befehlszählerwert auszuwählen, der entweder der inkrementierte Befehlszählerwert (Befehlszählerwert + 4), der Befehlszähler des Verzweigungsziels oder der Befehlszähler des Sprungziels ist. Für den zusätzlichen Multiplexer ist ein weiteres Steuersignal erforderlich. Dieses Steuersignal, als *Jump* bezeichnet, wird nur auf logisch 1 gesetzt, wenn es sich bei dem Befehl um einen Sprung handelt, d.h. wenn der Opcode 2 ist.

ANTWORT

Tab. 5.5 Die Steuerfunktion für die einfache Eintaktausführung wird durch diese Wahrheitstabelle vollständig beschrieben. In der oberen Hälfte der Tabelle sind die Kombinationen der Eingangssignale angegeben, die den vier Opcodes entsprechen, mit denen bestimmt wird, wie die Steuersignale gesetzt werden. (Op [5:0] entspricht den Bits 31:26 des Befehls, d.h. dem Op-Feld.) Im unteren Teil der Tabelle sind die Ausgänge angegeben. Der Ausgang RegWrite ist somit für zwei verschiedene Eingangskombinationen logisch 1. Wenn wir nur die vier in dieser Tabelle dargestellten Opcodes betrachten, können wir die Wahrheitstabelle durch Verwenden von Don't-cares im Eingangsbereich vereinfachen. Wir können beispielsweise einen R-Befehl mit dem Ausdruck $\overline{\text{Op}5} \cdot \overline{\text{Op}2}$ erkennen, da dies ausreicht, um R-Befehle von den Befehlen `lw`, `sw` und `beq` zu unterscheiden. Wir nutzen diese Möglichkeit der Vereinfachung nicht, da die restlichen MIPS-Opcodes in einer vollständigen Implementierung verwendet werden.

Eintaktausführung (single-cycle implementation) Auch als Eintaktzyklusausführung bezeichnet. Eine Implementierung, bei der ein Befehl in einem Taktzyklus ausgeführt wird.

Eingang oder Ausgang	Signalname	R-Format	lw	sw	beq
Eingänge	Op5	0	1	1	0
	Op4	0	0	0	0
	Op3	0	0	1	0
	Op2	0	0	0	1
	Op1	0	1	1	0
	Op0	0	1	1	0
Ausgänge	RegDst	1	0	X	X
	ALUSrc	0	1	1	0
	MemtoReg	0	1	X	X
	RegWrite	1	1	0	0
	MemRead	0	1	0	0
	MemWrite	0	0	1	0
	Branch	0	0	0	1
	ALUOp1	1	0	0	0
	ALUOp0	0	0	0	1

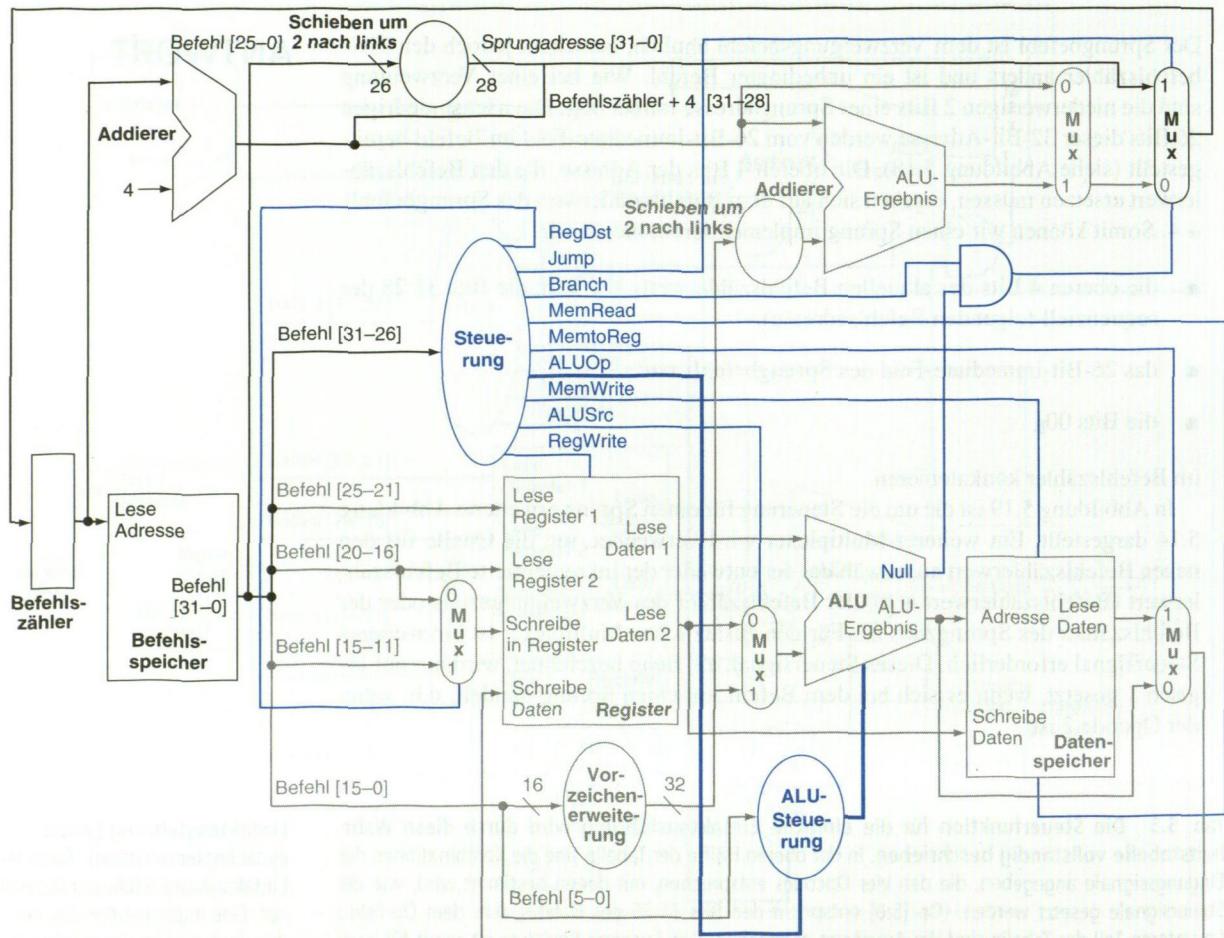


Abb. 5.19 Die einfache Steuerung und der einfache Datenpfad wurden zum Steuern des Sprungbefehls erweitert. Ein weiterer Multiplexer (rechts oben) wird verwendet, um zwischen dem Sprungziel und entweder dem Verzweigungsziel oder dem sequenziell diesem Befehl folgenden Befehl auszuwählen. Dieser Multiplexer wird durch das Sprungsteuersignal gesteuert. Die Sprungzieladresse wird durch Verschieben der unteren 26 Bits des Sprungbefehls um 2 Bit nach links ermittelt, d.h. durch Hinzufügen von 00 als die niedrigwertigsten Bits, und durch anschließendes Verknüpfen der oberen 4 Bits des Befehlszählerwerts + 4 als die hochwertigen Bits, was eine 32-Bit-Adresse ergibt.

Warum eine Eintaktausführung heute nicht verwendet wird

Obwohl der Eintaktentwurf einwandfrei funktioniert, wird er in modernen Entwürfen nicht verwendet, weil er nicht effizient ist. Um zu verstehen, warum das so ist, müssen Sie bedenken, dass der Taktzyklus in diesem Eintaktentwurf für jeden Befehl gleich lang sein muss, und dass der CPI-Wert (siehe Kapitel 4) somit 1 beträgt. Der Taktzyklus wird durch den längsten möglichen Pfad im Rechner bestimmt. Dieser Pfad ist mit Sicherheit ein Ladebefehl, der der Reihe nach fünf Funktionseinheiten nutzt: den Befehlsspeicher, den Registersatz, die ALU, den Datenspeicher und den Registersatz. Obwohl der CPI-Wert 1 beträgt, ist die Gesamtleistung einer Eintaktausführung nicht besonders gut, da einige Befehlsklassen in einen kürzeren Taktzyklus passen würden.

Rechenleistung von Eintaktrechnern

BEISPIEL

Nehmen wir an, die Ausführungszeiten der wichtigsten Funktionseinheiten in dieser Implementierung lauten wie folgt:

- Speichereinheiten: 200 Picosekunden (ps)
- ALU und Addierer: 100 ps
- Registersatz (lesen oder schreiben): 50 ps

Nehmen wir an, Multiplexer, Steuereinheit, Befehlszählerzugriffe, Vorzeichenerweiterungseinheit und Verbindungsleitungen weisen keine Verzögerung auf. Welche der folgenden Implementierungen wäre schneller und um wie viel?

1. Eine Implementierung, bei der jeder Befehl in einem Taktzyklus mit fester Länge ausgeführt wird.
2. Eine Implementierung, bei der jeder Befehl in einem Taktzyklus mit variabler Länge ausgeführt wird, wobei der Taktzyklus für jeden Befehl nur so lange wie erforderlich ist. (Ein Ansatz wie dieser ist nicht besonders einfach zu realisieren, aber er ermöglicht uns zu sehen, was geopfert wird, wenn alle Befehle in einem Taktzyklus mit derselben Länge ausgeführt werden müssen.)

Um die Rechenleistung zu vergleichen, gehen wir von der folgenden Befehlszusammenstellung aus: 25 % Ladebefehle, 10 % Speicherbefehle, 45 % ALU-Befehle, 15 % Verzweigungen und 5 % Sprünge.

Beginnen wir damit, dass wir die CPU-Ausführungszeiten vergleichen. Erinnern Sie sich an Kapitel 4, dass

ANTWORT

$$\text{CPU-Ausführungszeit} = \text{Befehlszahl} \times \text{CPI-Wert} \times \text{Taktzykluszeit}$$

Da der CPI-Wert 1 sein muss, können wir diese Gleichung wie folgt vereinfachen:

$$\text{CPU-Ausführungszeit} = \text{Befehlszahl} \times \text{Taktzykluszeit}$$

Wir müssen lediglich die Taktzykluszeit für die beiden Implementierungen ermitteln, da die Befehlszahl und der CPI-Wert bei beiden Implementierungen derselbe ist. Der für die unterschiedlichen Befehlsklassen kritische Pfad lautet wie folgt:

Befehls-klasse		Von der Befehlsklasse verwendete Funktionseinheiten				
R-Befehl	Befehlsholschritt	Registerzugriff	ALU	Registerzugriff		
load word	Befehlsholschritt	Registerzugriff	ALU	Speicherzugriff	Registerzugriff	
store word	Befehlsholschritt	Registerzugriff	ALU	Speicherzugriff		
branch	Befehlsholschritt	Registerzugriff	ALU			
jump	Befehlsholschritt					

Mit diesen kritischen Pfaden können wir die erforderliche Länge für die einzelnen Befehlsklassen berechnen:

Befehls-klasse	Befehls-speicher	Register-lese-vorgang	ALU-Operation	Daten-speicher	Register-schreib-vorgang	Gesamt
R-Befehl	200	50	100	0	50	400 ps
load word	200	50	100	200	50	600 ps
store word	200	50	100	200		550 ps
branch	200	50	100	0		350 ps
jump	200					200 ps

Der Taktzyklus für einen Rechner mit einem festen Takt für alle Befehle wird durch den längsten Befehl bestimmt und beträgt 600 ps. (Hierbei handelt es sich lediglich um einen Näherungswert da unser Zeitmodell stark vereinfacht ist. In Wirklichkeit ist das Zeitverhalten moderner digitaler Systeme deutlich komplexer.)

Bei einem Rechner mit einem variablen Takt liegt der Taktzyklus zwischen 200 ps und 600 ps. Wir können die durchschnittliche Taktzykluslänge für einen Rechner mit einer variablen Taktlänge mithilfe der oben angegebenen Daten und der Häufigkeitsverteilung der Befehle ermitteln.

Somit ergibt sich für die durchschnittliche Zeit pro Befehl mit einem variablen Taktzyklus

CPU-Taktzyklus

$$\begin{aligned}
 &= 600 \times 25\% + 550 \times 10\% + 400 \times 45\% + 350 \times 15\% + 200 \times 5\% \\
 &= 447,5 \text{ ps}
 \end{aligned}$$

Da die Implementierung mit variablem Taktzyklus einen kürzeren durchschnittlichen Taktzyklus aufweist, ist diese Implementierung eindeutig schneller. Lassen Sie uns nun das Leistungsverhältnis ermitteln:

$$\frac{\text{Prozessorleistung}_{\text{variabler Takt}}}{\text{Prozessorleistung}_{\text{fester Takt}}} = \frac{\text{Prozessorausführungszeit}_{\text{fester Takt}}}{\text{Prozessorausführungszeit}_{\text{variabler Takt}}} = \left(\frac{\text{Befehlszahl} \times \text{Prozessortaktzyklen}_{\text{fester Takt}}}{\text{Befehlszahl} \times \text{Prozessortaktzyklen}_{\text{variabler Takt}}} \right) = \frac{\text{Prozessortaktzyklen}_{\text{fester Takt}}}{\text{Prozessortaktzyklen}_{\text{variabler Takt}}} = \frac{600}{447,5} = 1,34$$

Die Implementierung mit variablem Taktzyklus wäre 1,34-mal schneller. Leider ist es äußerst schwierig, für jede Befehlsklasse einen variablen Taktzyklus zu implementieren. Und der Aufwand für einen Ansatz wie diesen steht möglicherweise in keinem Verhältnis zum Nutzen. Wie wir im nächsten Abschnitt sehen werden, gibt es eine Alternative, die darin besteht, einen kürzeren Taktzyklus, in dem weniger Arbeit bewältigt wird, zu verwenden und die Anzahl der Taktzyklen für die unterschiedlichen Befehlsklassen zu variieren.

Der Eintaktzyklusentwurf mit einem festen Taktzyklus bringt erhebliche Einbußen mit sich, die für diesen kleinen Befehlssatz jedoch in Kauf genommen werden können. In der Vergangenheit wurde diese Implementierungsmethode tatsächlich für Rechner mit sehr einfachen Befehlssätzen verwendet. Wenn wir jedoch versuchten, die Gleitkommaeinheit oder einen Befehlssatz mit komplexeren Befehlen zu implementieren, würde dieser Eintaktentwurf nicht funktionieren (Beispiel auf CD in Aufgabe 5.4 im Abschnitt **For More Practice**).

Da wir annehmen müssen, dass der Taktzyklus für alle Befehle der Verzögerung im ungünstigsten Fall (worst case) entspricht, können wir keine Implementierungstechnik



niken verwenden, die die Verzögerung des häufig vorkommenden Falls reduzieren, die Worst-Case-Zykluszeit jedoch nicht verbessern. Eine Eintaktausführung verletzt somit unser wichtigstes Entwurfsprinzip, nach dem der häufig vorkommende Fall optimiert werden soll. Darüber hinaus kann bei dieser Eintaktausführung jede Funktionseinheit nur einmal pro Takt eingesetzt werden. Daher müssen einige Funktionseinheiten mehrfach vorhanden sein, wodurch die Implementierung teurer wird. Ein Eintaktzyklusentwurf ist also sowohl hinsichtlich der Leistung als auch hinsichtlich der Hardwarekosten ineffizient!

Diese Schwierigkeiten können wir vermeiden, indem wir Implementierungstechniken mit einem kürzeren Taktzyklus (abgeleitet von der Verzögerung der Hauptfunktionseinheit) verwenden, die für jeden Befehl mehrere Taktzyklen erfordern. Im nächsten Abschnitt wird diese alternative Implementierungsmethode untersucht. In Kapitel 6 werden wir eine weitere Implementierungstechnik betrachten. Hierbei handelt es sich um das Pipelining, bei dem ein Datenpfad verwendet wird, der dem Eintaktdatenpfad sehr ähnlich, jedoch wesentlich effizienter ist. Das Pipelining ist effizienter, weil sich die Ausführung mehrerer Befehle überschneidet. Dadurch wird die Hardware besser genutzt und die Leistung verbessert. Für Leser, die sich in erster Linie für die in Prozessoren verwendeten anspruchsvollen Konzepte interessieren, reichen die Informationen in diesem Abschnitt aus, um die einführenden Abschnitte von Kapitel 6 zu lesen und die grundlegende Funktion eines Pipeline-Prozessors zu verstehen. Diejenigen Leser, die wissen möchten, wie die Hardware die Steuerung wirklich implementiert, lesen einfach weiter.

Sehen Sie sich das Steuersignal in Tabelle 5.5 an. Kann eines der Steuersignale in der Tabelle durch die Invertierung eines anderen ersetzt werden? (Hinweis: Berücksichtigen Sie die Don't-cares.) Wenn ja, lässt sich ein Signal ohne Einsatz eines Inverters für das andere verwenden?



5.5

Eine Mehrzyklenimplementierung

In einem Beispiel weiter oben haben wir die einzelnen Befehle entsprechend der erforderlichen Operationen von Funktionseinheiten in Schritte zerlegt. Mithilfe dieser Schritte können wir eine **Mehrzyklenimplementierung** (*multicycle implementation*) erstellen. In einer Mehrzyklenimplementierung benötigt jeder *Schritt* in der Ausführung einen Taktzyklus. Bei der Mehrzyklenimplementierung kann eine Funktionseinheit mehrere Male pro Befehl verwendet werden, sofern sie in unterschiedlichen Taktzyklen eingesetzt wird. Diese mehrmalige Nutzung kann dazu beitragen, dass weniger Hardware erforderlich ist. Die Eigenschaft, dass Befehle unterschiedlich viele Taktzyklen beanspruchen können, und die Eigenschaft, dass Funktionseinheiten während der Ausführung eines einzigen Befehls mehrere Male eingesetzt werden können, sind die Hauptvorteile einer Mehrzyklenimplementierung. In Abbildung 5.20 ist die abstrakte Version des Mehrzyklendatenpfads dargestellt. Beim Vergleich von Abbildung 5.20 mit dem Datenpfad für die Eintaktversion in Abbildung 5.11 ergeben sich die folgenden Unterschiede:

- Für Befehle und Daten wird nur eine gemeinsame Speichereinheit verwendet.
- Anstelle einer ALU und zweier Addierer gibt es nur eine ALU.

Mehrzyklenimplementierung
(multicycle implementation)
 Auch als Mehrtaktzyklenimplementierung bezeichnet. Eine Implementierung, bei der ein Befehl in mehreren Taktzyklen ausgeführt wird.

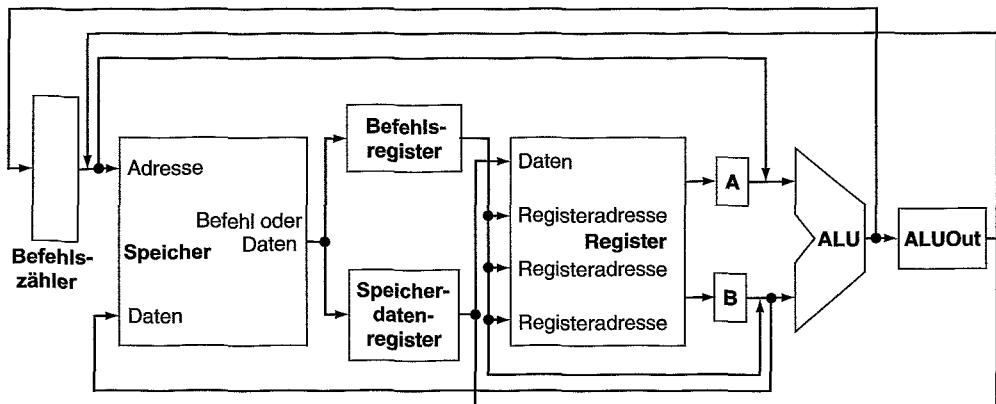


Abb. 5.20 Die abstrakte Darstellung eines Mehrzyklendatenpfads. In dieser Abbildung sind die wichtigsten Elemente des Datenpfads dargestellt: eine gemeinsame Speichereinheit, eine von mehreren Befehlen gemeinsam genutzte ALU sowie die Verbindungen zwischen diesen gemeinsamen Einheiten. Damit gemeinsame Funktionseinheiten genutzt werden können, müssen Multiplexer ergänzt oder erweitert und neue temporäre Register eingefügt werden, in denen Daten zwischen Taktzyklen für denselben Befehl gespeichert werden können. Bei den zusätzlichen Registern handelt es sich um das Befehlsregister (IR), das Speicherdatenregister (MDR), A, B und ALUOut.

- Nach jeder größeren Funktionseinheit werden ein oder mehrere Register eingefügt, um den Ausgabewert dieser Einheit zu speichern, bis der Wert in einem nachfolgenden Taktzyklus verwendet wird.

Am Ende eines Taktzyklus müssen alle in nachfolgenden Taktzyklen verwendete Daten in einem Speicherelement gespeichert werden. Daten, die in einem späteren Taktzyklus von *nachfolgenden Befehlen* verwendet werden, werden in einem der für den Programmierer sichtbaren Speicherelemente gespeichert: im Registersatz, im Befehlszähler oder im Speicher. Daten, die in einem späteren Taktzyklus von *demselben Befehl* verwendet werden, müssen dagegen in einem dieser zusätzlichen Register gespeichert werden.

Die Position der zusätzlichen Register wird somit von zwei Faktoren bestimmt: Welche kombinatorischen Einheiten passen in einen Taktzyklus und welche Daten werden in späteren Taktzyklen zum Implementieren des Befehls benötigt. Bei diesem Mehrzyklentwurf gehen wir davon aus, dass in einem Taktzyklus maximal eine der folgenden Operationen durchgeführt werden kann: ein Speicherzugriff, ein Zugriff auf den Registersatz (zwei Lesevorgänge oder ein Schreibvorgang) oder eine ALU-Operation. Sämtliche Daten, die durch diese drei Funktionseinheiten (Speicher, Registersatz oder ALU) generiert werden, müssen also in einem temporären Register für die Verwendung in einem späteren Taktzyklus gespeichert werden. Wenn die Daten nicht gespeichert würden, bestünde die Gefahr einer Wettrennbedingung, die die Verwendung eines falschen Werts zur Folge haben könnte.

Um diese Anforderungen zu erfüllen, werden die folgenden temporären Register ergänzt:

- Das Befehlsregister (IR, Instruction Register) und das Speicherdatenregister (MDR, Memory Data Register) werden ergänzt, um den Ausgangswert der Speichereinheit jeweils für einen Befehlslesevorgang und einen Datenlesevorgang zu speichern. Hierfür werden zwei getrennte Register verwendet, da beide Werte während desselben Taktzyklus benötigt werden, wie wir in Kürze sehen werden.
- Die Register A und B werden zum Speichern der vom Registersatz bereitgestellten Werte der Registeroperanden verwendet.
- Im Register ALUOut wird der Ausgangswert der ALU gespeichert.

Mit Ausnahme des IR-Registers werden in allen Registern Daten nur für die Zeitdauer zwischen zwei aufeinander folgenden Taktzyklen gespeichert, so dass hier kein Schreibsteuersignal erforderlich ist. Im IR-Register muss der Befehl bis zum Ende der Ausführung dieses Befehls gespeichert werden. Daher erfordert dieses Register ein Schreibsteuersignal. Diese Unterscheidung wird bei der Beschreibung der einzelnen Taktzyklen für die jeweiligen Befehle deutlicher.

Da einige Funktionseinheiten aus unterschiedlichen Gründen gemeinsam genutzt werden, müssen wir einerseits weitere Multiplexer einsetzen und andererseits vorhandene Multiplexer erweitern. Beispiel: Da die Speichereinheit sowohl für Befehle als auch für Daten verwendet wird, benötigen wir einen Multiplexer, um zwischen den beiden Quellen für eine Speicheradresse, also zwischen dem Befehlszähler (bei Befehlszugriff) und ALUOut (für Datenzugriff) auszuwählen.

Wenn wir die drei ALUs aus dem Eintaktdatenpfad durch eine ALU ersetzen, bedeutet das, dass diese eine ALU alle Eingänge aufnehmen muss, die bei der Eintaktausführung von drei verschiedenen ALUs bereitgestellt werden. Der Umgang mit den zusätzlichen Eingängen erfordert zwei Änderungen am Datenpfad:

1. Für den ersten ALU-Eingang wird ein weiterer Multiplexer hinzugefügt. Der Multiplexer wählt zwischen dem A-Register und dem Befehlszähler.
2. Aus dem 2:1-Multiplexer am zweiten ALU-Eingang wird ein 4:1-Multiplexer. Die beiden zusätzlichen Eingänge am Multiplexer sind die Konstante 4 (zum Inkrementieren des Befehlszählers) und das vorzeichenerweiterte und verschobene Offset-Feld (zum Berechnen der Sprungadresse).

In Abbildung 5.21 sind die einzelnen Elemente des Datenpfads mit diesen zusätzlichen Multiplexern dargestellt. Durch die Einführung weniger Register und Multiplexer benötigen wir statt zwei nur noch eine Speichereinheit und können auf zwei Addierer verzichten. Da Register und Multiplexer im Vergleich zu einer Speichereinheit oder ALU recht klein sind, kann dies eine spürbare Reduzierung der Hardwarekosten bedeuten.

Da bei dem in Abbildung 5.21 dargestellten Datenpfad pro Befehl mehrere Taktzyklen verwendet werden, sind andere Steuersignale erforderlich. Die für den Programmierer sichtbaren Einheiten (Befehlszähler, Speichereinheit und Registersatz) sowie das IR-Register benötigen Schreibsteuersignale. Die Speichereinheit benötigt darüber hinaus ein Lesesignal. Zum Steuern der ALU können wir die ALU-Steuereinheit aus dem Eintaktdatenpfad verwenden (siehe Tabelle 5.2 und **Appendix C** auf CD). Und schließlich benötigt jeder der Multiplexer mit zwei Eingängen eine Steuerleitung, während der Multiplexer mit vier Eingängen zwei Steuerleitungen beansprucht. In Abbildung 5.22 ist der Datenpfad aus Abbildung 5.21 mit diesen zusätzlichen Steuerleitungen dargestellt.



Der Mehrzyklendatenpfad muss noch weiter ergänzt werden, um Verzweigungen und Sprünge unterstützen zu können. Nach diesen Ergänzungen werden wir die Ablauffolge der Befehle sehen und anschließend die Datenpfadsteuerung entwerfen.

Beim Sprungbefehl und beim Verzweigungsbefehl gibt es drei mögliche Quellen für den Wert, der in den Befehlszähler geschrieben wird:

1. Der Ausgangswert der ALU, der dem Befehlszählerwert + 4 während der Befehlsolphase entspricht. Dieser Wert muss direkt im Befehlszähler gespeichert werden.
2. Das Register ALUOut, in dem die Adresse des Sprungziels, nachdem diese berechnet wurde, gespeichert wird.
3. Die unteren 26 Bits des Befehlsregisters (IR), um zwei Bits nach links verschoben und mit den oberen 4 Bits des inkrementierten Befehlszählers (der bei einem Sprungbefehl die Quelle darstellt) verknüpft.

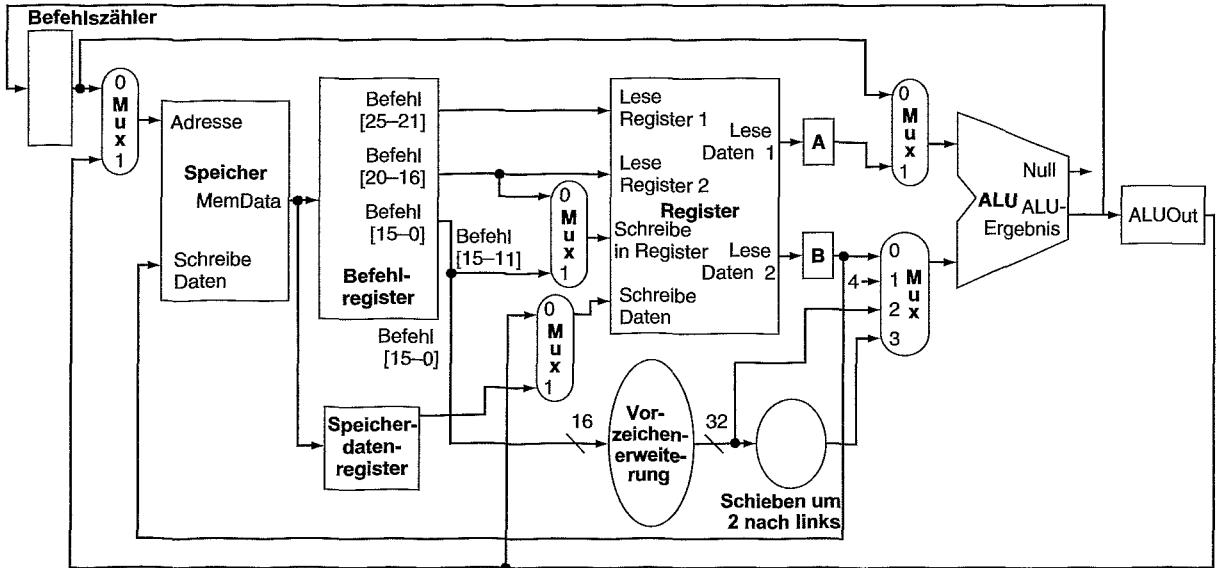


Abb. 5.21 Der Mehrzyklendatenpfad für MIPS bearbeitet die grundlegenden Befehle. Dieser Datenpfad unterstützt zwar die normale Inkrementierung des Befehlszählers, dennoch werden einige zusätzliche Verbindungen und ein Multiplexer für Verzweigungen und Sprünge benötigt. Wir werden diese in Kürze ergänzen. Diese Ergänzungen umfassen im Vergleich zum Eintaktdatenpfad mehrere Register (IR, MDR, A, B, ALUOut), einen Multiplexer für die Speicheradresse, einen Multiplexer für den oberen ALU-Eingang und eine Erweiterung des Multiplexers am unteren ALU-Eingang zu einem 4:1-Selektor. Aufgrund dieser Ergänzungen ist es uns möglich, auf zwei Addierer und eine Speiereinheit zu verzichten.

Wie wir bereits beim Implementieren der Eintaktsteuerung beobachten konnten, werden Werte bedingt und unbedingt in den Befehlszähler geschrieben. Während einer normalen Inkrementierung und bei Sprüngen wird ein Wert unbedingt in den Befehlszähler geschrieben. Wenn es sich bei dem Befehl um einen bedingten Sprung bzw. um eine Verzweigung handelt, wird der inkrementierte Befehlszähler nur durch den Wert im ALUOut-Register ersetzt, sofern die beiden Zielregister gleich sind. Somit benötigen wir für unsere Implementierung zwei getrennte Steuersignale: PCWrite, das einen Wert unbedingt in den Befehlszähler schreibt, und PCWriteCond, das einen Wert in den Befehlszähler schreibt, sofern die Sprungbedingung erfüllt ist.

Wir müssen diese beiden Steuersignalleitungen mit der Schreibsteuerung des Befehlszählers verbinden. Wie beim Eintaktdatenpfad werden wir einige Gatter verwenden, um das Signal für die Schreibsteuerung des Befehlszählers aus dem PCWrite-, PCWriteCond- und Zero-Signal der ALU zu gewinnen, das verwendet wird, um festzustellen, ob die beiden Registeroperanden eines `beq`-Befehls gleich sind. Um zu bestimmen, ob der Befehlszähler während eines bedingten Sprungs beschrieben werden muss, verknüpfen wir die Zero-Signalleitung der ALU mit der PCWriteCond-Signalleitung mithilfe eines UND-Gatters. Die Ausgangsleitung dieses UND-Gatters wird mithilfe eines ODER-Gatters mit der Leitung für das unbedingte Schreibsteuersignal PCWrite des Befehlszählers verknüpft. Die Ausgangsleitung dieses ODER-Gatters wird mit der Signalleitung für die Schreibsteuerung des Befehlszählers verbunden.

In Abbildung 5.23 ist der vollständige Mehrzyklendatenpfad und die Steuereinheit mit den zusätzlichen Steuersignalen und dem Multiplexer für die Implementierung der Befehlszähleraktualisierung dargestellt.

Bevor wir die Schritte zum Ausführen der einzelnen Befehle untersuchen, möchten wir die Wirkungsweise all dieser Steuersignale (wie beim Eintaktentwurf in Tabelle 5.3) näher betrachten. In Tabelle 5.6 ist die Wirkungsweise der einzelnen Steuersignale im Zustand „logisch 1“ und „logisch 0“ dargestellt.

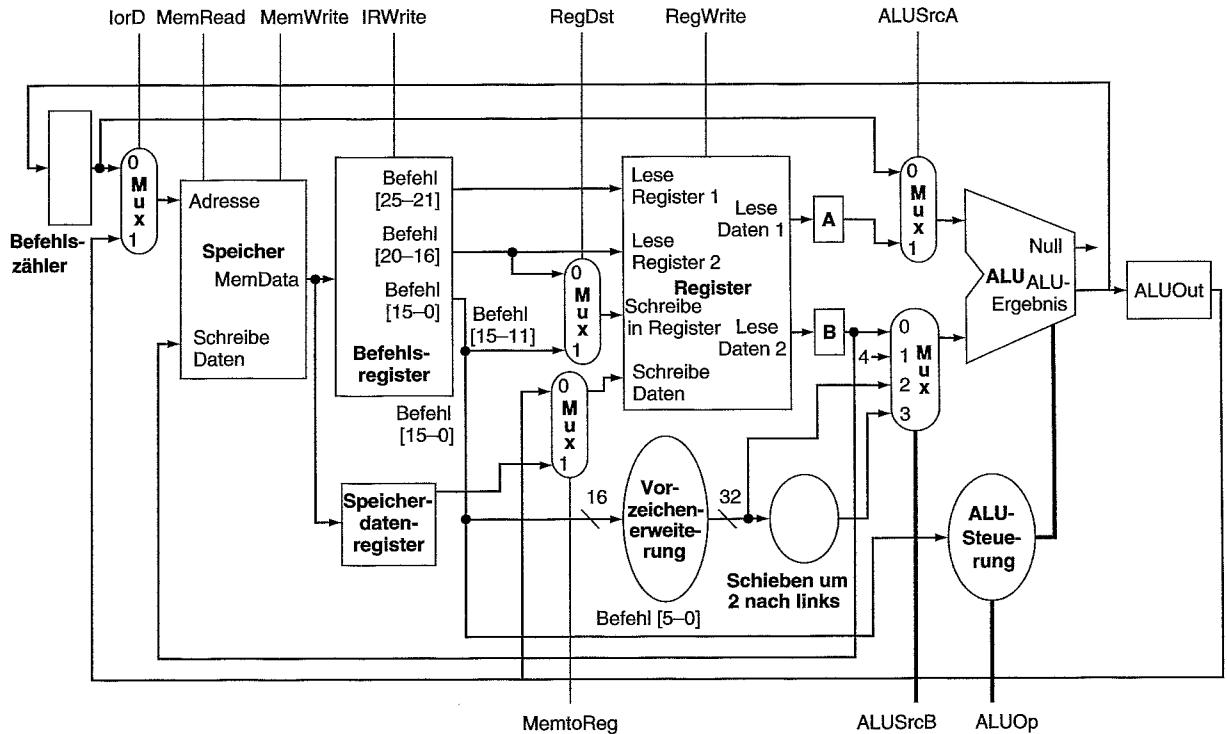


Abb. 5.22 Der Mehrzyklendatenpfad aus Abbildung 5.21 mit den Steuerleitungen. Bei den Signalleitungen ALUOp und ALUSrcB handelt es sich um 2 Bit breite Leitungen, während alle anderen Steuerleitungen 1 Bit breit sind. Weder für Register A noch für Register B ist ein Schreibsignal erforderlich, da deren Inhalt nur bei dem Zyklus gelesen wird, der dem Schreibvorgang direkt folgt. Das Speicherdatenregister wurde neu eingefügt, um die Daten aus einem Ladevorgang zu speichern, wenn die Daten aus dem Speicher kommen. Daten, die bei einem Ladevorgang aus dem Speicher kommen, können nicht direkt in den Registersatz geschrieben werden, da der Taktzyklus nicht genügend Zeit bereitstellt, um sowohl auf den Speicher zuzugreifen als auch in den Registersatz zu schreiben. Die MemRead-Signalleitung wurde oben in die Speichereinheit verlegt, um die Darstellung zu vereinfachen. Der vollständige Satz Datenpfade und Steuerleitungen für Sprünge wird in Kürze ergänzt.

Vertiefung: Um die Anzahl der Signalleitungen für die Verbindung der Funktionseinheiten untereinander zu reduzieren, können Entwickler gemeinsame Busse verwenden. Bei einem gemeinsamen Bus handelt es sich um mehrere Leitungen, die mehrere Einheiten miteinander verbinden. In der Regel bestehen diese aus mehreren Quellen, die Daten auf den Bus legen können, und aus mehreren Zielen, die diese Werte lesen können. So, wie wir die Anzahl der Funktionseinheiten für den Datenpfad reduziert haben, so können wir die Anzahl der Busse reduzieren, die diese Einheiten miteinander verbinden, indem wir die Busse gemeinsam nutzen. Mit der ALU sind beispielsweise sechs Quellen verbunden. Es werden aber immer jeweils nur zwei davon benötigt. Somit kann ein Buspaar zum Speichern von Werten verwendet werden, die an die ALU gesendet werden. Statt nun einen großen Multiplexer vor die ALU zu schalten, kann ein Entwickler einen gemeinsamen Bus verwenden und sicherstellen, dass immer jeweils nur eine der Quellen den Bus ansteuert. Damit werden zwar Signalleitungen gespart, es werden jedoch genauso viele Steuerleitungen benötigt, um zu steuern, welche Werte auf den Bus geschaltet werden dürfen. Eine Busstruktur wie diese hat den großen Nachteil möglicher Leistungseinbußen, da ein Bus aller Wahrscheinlichkeit nach nicht so schnell ist wie eine Punkt-zu-Punkt-Verbindung.



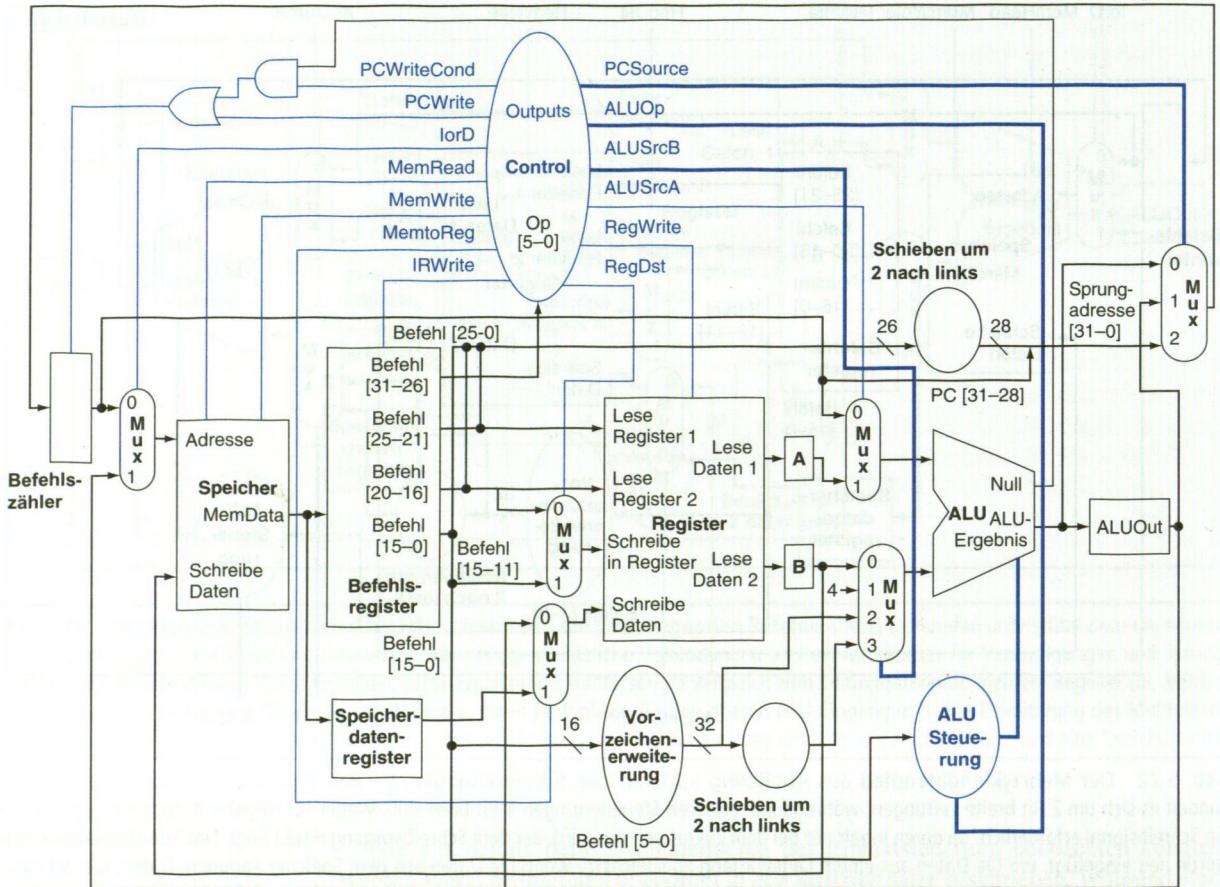


Abb. 5.23 Der vollständige Datenpfad für die Mehrzyklenausführung mit den erforderlichen Steuerleitungen. Die Steuerleitungen aus Abbildung 5.22 wurden mit der Steuereinheit verbunden, und die zum Ändern des Befehlszählers erforderlichen Steuer- und Datenpfad-elemente wurden ergänzt. Zu den wichtigsten Ergänzungen zu Abbildung 5.22 zählen der Multiplexer, der zum Auswählen der Quelle eines neuen Befehlszählerwerts verwendet wird, Gatter, die zum Verknüpfen der Schreibsignale für den Befehlszähler verwendet werden, und die Steuersignale PCSource, PCwrite und PCwriteCond. Das PCWriteCohd-Signal wird verwendet, um zu entscheiden, ob ein bedingter Sprung ausgeführt werden muss. Die Unterstützung von Sprüngen wurde ergänzt.

Zerlegen der Befehlausführung in Taktzyklen

Den Datenpfad in Abbildung 5.23 vorausgesetzt, müssen wir nun untersuchen, was in den einzelnen Taktzyklen der Mehrzyklenausführung geschieht, da dadurch bestimmt wird, welche zusätzlichen Steuersignale möglicherweise benötigt werden, und wie die Steuersignale gesetzt werden. Mit dem Zerlegen der Ausführung in Taktzyklen möchten wir eine Maximierung der Ausführungsleistung erzielen. Wir können damit beginnen, die Ausführung eines Befehls in mehrere Schritte zu zerlegen, die jeweils einen Taktzyklus beanspruchen, und zu versuchen, die Menge an Arbeit pro Zyklus in etwa gleich zu halten. So werden wir beispielsweise die Einschränkung vornehmen, dass jeder Schritt maximal eine ALU-Operation oder einen Registersatzzugriff oder einen Speicherzugriff enthält. Mit dieser Einschränkung ist es möglich, dass der Taktzyklus so kurz sein kann wie die längste dieser Operationen.

Denken Sie daran, dass am Ende eines Taktzyklus Datenwerte, die in einem nachfolgenden Zyklus benötigt werden, in einem Register gespeichert werden müssen, bei dem es sich entweder um eines der zentralen Speicherelemente (z.B. Befehlszähler, Registersatz oder Speicher), um ein temporäres Register, in das bei jedem Taktzyklus

Tab. 5.6 Die Aktion, die jeweils durch das Setzen der einzelnen Steuersignale in Abbildung 5.23 verursacht wird. In der oberen Tabelle werden die 1-Bit-Steuersignale beschrieben, während in der unteren Tabelle die 2-Bit-Signale beschrieben werden. Nur die Steuerleitungen, die Multiplexer ansteuern, bewirken eine Aktion, wenn sie auf logisch 0 gesetzt sind. Die Angaben in dieser Tabelle ähneln denen in Tabelle 5.3 zum Eintaktdatenpfad, enthalten jedoch einige neue Steuerleitungen (IRWrite, PCWrite, PCWriteCond, ALUSrcB und PCSource). Zudem sind die Steuerleitungen, die nicht mehr verwendet werden oder ersetzt wurden (PCSrc, Branch und Jump), nicht mehr enthalten.

Aktionen der 1-Bit-Steuersignale

Signalname	Wirkung bei logisch 0	Wirkung bei logisch 1
RegDst	Die Adresse des Zielregisters für „Schreibe in Register“ wird vom rt-Feld bereitgestellt.	Die Adresse des Zielregisters für „Schreibe in Register“ wird vom rd-Feld bereitgestellt.
RegWrite	Keine.	In das durch „Schreibe in Register Adresse“ ausgewählte Allzweckregister wird der Wert des Schreibe-Daten-Eingangs geschrieben.
ALUSrcA	Der erste ALU-Operand ist der Befehlszähler.	Der erste ALU-Operand wird von Register A bereitgestellt.
MemRead	Keine.	Der Speicherinhalt an der vom Adresseneingang angegebenen Position wird an den Speicherdatenausgang gelegt.
MemWrite	Keine.	Der Speicherinhalt an der vom Adresseneingang angegebenen Position wird durch den Wert am Schreibe-Daten-Eingang ersetzt.
MemtoReg	Der dem Schreibe-Daten-Eingang des Registersatzes bereitgestellte Wert wird von ALUOut bereitgestellt.	Der dem Schreibe-Daten-Eingang des Registersatzes bereitgestellte Wert wird vom MDR bereitgestellt.
lOrD	Der Befehlszähler wird verwendet, um der Speichereinheit die Adresse bereitzustellen.	ALUOut wird verwendet, um der Speichereinheit die Adresse bereitzustellen.
IRWrite	Keine.	Der Ausgabewert des Speichers wird in das IR geschrieben.
PCWrite	Keine.	Der Befehlszählerwert wird geschrieben. Die Quelle wird durch PCSource angesteuert.
PCWriteCond	Keine.	Der Befehlszählerwert wird geschrieben, wenn der Zero-Ausgang aus der ALU ebenfalls aktiv ist.

Aktionen der 2-Bit-Steuersignale

Signalname	Wert (binär)	Wirkungsweise
ALUOp	00	Die ALU führt eine Addition durch.
	01	Die ALU führt eine Subtraktion durch.
	10	Das funct-Feld des Befehls legt fest, welche ALU-Operation durchgeführt wird.
ALUSrcB	00	Der zweite Eingangswert an der ALU wird von Register B bereitgestellt.
	01	Der zweite Eingangswert an der ALU ist die Konstante 4.
	10	Der zweite Eingangswert an der ALU besteht aus den vorzeichenverweiterten, unteren 16 Bits des IR.
	11	Der zweite Eingang an der ALU besteht aus den um zwei Bits nach links verschobenen, vorzeichenverweiterten, unteren 16 Bits des IR.
PCSource	00	Der Ausgangswert der ALU (Befehlszählerwert + 4) wird an den Befehlszähler gesendet, um in den Befehlszähler geschrieben zu werden.
	01	Der Inhalt von ALUOut (Sprungzieladresse) wird an den Befehlszähler gesendet, um in den Befehlszähler geschrieben zu werden.
	10	Die Sprungzieladresse (IR[25:0] um zwei Bits nach links verschoben und mit Befehlszählerwert + 4 [31:28] konkateniert) wird an den Befehlszähler gesendet, um in den Befehlszähler geschrieben zu werden.

geschrieben wird (z.B. A, B, MDR oder ALUOut) oder um ein temporäres Register mit Schreibsteuerung (z.B. IR) handelt. Denken Sie außerdem daran, dass wir aufgrund der Tatsache, dass unser Entwurf flankengesteuert ist, den aktuellen Wert eines Registers fortlaufend lesen können. Der neue Wert wird erst beim nächsten Taktzyklus angezeigt.

Beim Eintaktdatenpfad verwendet jeder Befehl für seine Ausführung eine Reihe von Elementen im Datenpfad. Viele der Elemente im Datenpfad arbeiten sequenziell und

verwenden den Ausgangswert eines anderen Elements als Eingangswert. Einige Elemente im Datenpfad arbeiten parallel. So wird beispielsweise der Befehlszähler inkrementiert und gleichzeitig der Befehl gelesen. Eine ähnliche Situation finden wir in einem Mehrzyklendatenpfad vor. Alle in einem Schritt genannten Operationen werden in einem Taktzyklus parallel ausgeführt, während aufeinander folgende Schritte sequenziell in unterschiedlichen Taktzyklen ausgeführt werden. Die Einschränkung auf eine ALU-Operation, einen Speicherzugriff oder einen Registersatzzugriff bestimmt, was in einem Schritt passt.

Wir unterscheiden zwischen dem Lesen aus und dem Schreiben in den Befehlszähler oder eines der eigenständigen Register und dem Lesen aus oder Schreiben in den Registersatz. Beim ersten Fall ist der Lese- oder Schreibvorgang Teil eines Taktzyklus, während das Lesen oder Schreiben eines Ergebnisses in den Registersatz einen weiteren Taktzyklus erfordert. Dies wird unterschieden, weil beim Registersatz im Vergleich zu den einzelnen eigenständigen Registern zusätzliche Steuerungsvorgänge und Zugriffe durchgeführt werden, die zusätzliche Zeit benötigen. Wenn wir also den Taktzyklus kurz halten, benötigen wir für Registersatzzugriffe getrennte Taktzyklen.

Die möglichen Ausführungsschritte und deren Aktionen sind unten aufgeführt. Jeder MIPS-Befehl beansprucht zwischen drei und fünf dieser Schritte:

1. Befehlholschritt

Lade den Befehl aus dem Speicher und berechne die Adresse des nächsten sequenziellen Befehls:

```
IR <= Memory[PC] ;
PC <= PC + 4;
```

Operation: Sende den Befehlszählerwert als Adresse an den Speicher, führe einen Lesevorgang aus und schreibe den Befehl in das Befehlsregister (IR), in dem der Befehl gespeichert wird. Erhöhe außerdem den Befehlszählerwert um vier. Wir verwenden das Verilog-Symbol „`<=`“, das angibt, dass alle Werte auf der rechten Seite ausgewertet und anschließend alle Zuweisungen durchgeführt werden. Dies spiegelt wider, wie die Hardware Befehle während des Taktzyklus ausführt.

Um diesen Schritt zu implementieren, müssen wir die Steuersignale MemRead und IRWrite auf logisch 1 und IorD auf 0 setzen, um den Befehlszähler als Quelle der Adresse auszuwählen. Wir erhöhen außerdem den Befehlszähler um vier. Dazu müssen wir das ALUSrcA-Signal auf 0 (senden des Befehlszählers an die ALU), das ALUSrcB-Signal auf 01 (senden der Konstante 4 an die ALU) und ALUOp auf 00 (veranlassen, dass die ALU addiert) setzen. Schließlich werden wir die inkrementierte Befehlsadresse wieder im Befehlszähler speichern wollen. Dazu muss die Befehlszählerquelle auf 00 und das PCWrite-Signal gesetzt werden. Die Befehlszählerinkrementierung und der Befehlsspeicherzugriff können parallel erfolgen. Der neue Wert des Befehlszählers wird erst beim nächsten Taktzyklus sichtbar. (Der inkrementierte Befehlszähler wird ebenfalls in ALUOut gespeichert. Diese Aktion ist jedoch unkritisch.)

2. Befehlsentschlüsselungs- und Registerholschritt

Im vorhergehenden und in diesem Schritt kennen wir den Befehl noch nicht. Daher können wir nur Aktionen ausführen, die sich entweder auf alle Befehle anwenden lassen (wie das Holen des Befehls in Schritt 1) oder die für den Fall, dass der Befehl ein anderer als erwartet ist, nicht gefährlich sind. Somit können wir in diesem Schritt die beiden in den Befehlsfeldern rs und rt angegebenen Register lesen, da es nicht gefährlich ist, diese Register zu lesen, auch wenn dies möglicherweise nicht erforderlich ist.

Die aus dem Registersatz ausgelesenen Werte werden möglicherweise in späteren Phasen benötigt. Also lesen wir diese aus der Registerdatei aus und speichern sie in den temporären Registern A und B.

Darüber hinaus berechnen wir auch die Sprungzieladresse. Auch das ist ungefährlich, da wir den Wert ignorieren können, wenn sich herausstellt, dass es sich bei dem Befehl nicht um eine Verzweigung handelt. Das potenzielle Sprungziel wird in ALU-Out gespeichert.

Die frühe Durchführung dieser „optimistischen“ Aktionen hat den Vorteil, dass die Anzahl der für die Ausführung eines Befehls erforderlichen Taktzyklen reduziert wird. Wir können diese optimistischen Aktionen aufgrund der Regelmäßigkeit der Befehlsformate früh ausführen. Wenn der Befehl beispielsweise zwei Registereingänge enthält, befinden sich diese immer im rs- und im rt-Feld, und wenn es sich bei dem Befehl um eine Verzweigung handelt, umfasst der Offset immer die niederwertigen 16 Bits:

```
A <= Reg [IR [25:21]] ;
B <= Reg [IR [20:16]] ;
ALUOut <= PC + (sign-extend (IR [15:0]) << 2) ;
```

Operation: Greife auf den Registersatz zu, um die Register rs und rt zu lesen und das Ergebnis in die Register A und B zu schreiben. Da A und B bei jedem Zyklus überschrieben werden, kann der Registersatz bei jedem Zyklus gelesen und die Werte in A und B gespeichert werden. In diesem Schritt wird außerdem die Sprungzieladresse berechnet und in ALUOut gespeichert, wo sie beim nächsten Zyklus verwendet wird, sofern es sich bei dem Befehl um einen Sprung handelt. Dazu muss ALUSrcA auf 0 (so dass der Befehlszählerwert an die ALU gesendet wird), ALUSrcB auf 11 (so dass das vorzeichenerweiterte und verschobene Offset-Feld an die ALU gesendet wird) und ALUOp auf 00 (so dass die ALU eine Addition durchführt) gesetzt werden. Die Registersatzzugriffe und die Berechnung des Sprungziels werden parallel durchgeführt.

Die nach diesem Taktzyklus durchzuführende Aktion kann vom Befehlsinhalt abhängen.

3. Ausführung, Berechnung der Speicheradresse oder Sprungausführung

Dies ist der erste Zyklus, bei dem die Datenpfadoperation durch die Befehlsklasse bestimmt wird. In allen Fällen arbeitet die ALU mit den Operanden, die im vorhergehenden Schritt vorbereitet wurden, und führt abhängig von der Befehlsklasse eine von vier Funktionen aus. Wir geben die durchzuführende Aktion abhängig von der Befehlsklasse an:

Speicherzugriff:

```
ALUOut <= A + sign-extend (IR [15:0]) ;
```

Operation: Die ALU addiert die Operanden und bildet so die Speicheradresse. Dazu muss ALUSrcA auf 1 (so dass der erste ALU-Eingang an Register A gelegt ist) und ALUSrcB auf 10 gesetzt werden (so dass der Ausgang der Vorzeichenerweiterungseinheit für den zweiten ALU-Eingang verwendet wird). Die ALUOp-Signale müssen auf 00 gesetzt werden (wodurch die ALU zu einer Addition veranlasst wird).

Arithmetisch-logischer Befehl (R-Befehl):

```
ALUOut <= A op B;
```

Operation: Die ALU führt die vom Funktionscode angegebene Operation auf den beiden Werten aus, die im vorhergehenden Zyklus aus dem Registersatz ausgelesen wurden. Dazu muss ALUSrcA auf 1 und ALUSrcB auf 00 gesetzt werden, wodurch veranlasst wird, dass die Register A und B als ALU-Eingänge verwendet werden. Die ALUOp-Signale müssen auf 10 gesetzt werden (so dass das funct-Feld zum Bestimmen des ALU-Steuersignales verwendet wird).

Verzweigung:

```
if (A == B) PC <= ALUOut;
```

Operation: Die ALU wird verwendet, um zu prüfen, ob die beiden im vorhergehenden Schritt gelesenen Registerwerte gleich sind. Das Zero-Ausgangssignal der ALU bestimmt, ob ein Sprung durchgeführt wird oder nicht. Dazu muss ALUSrcA auf 1 und ALUSrcB auf 00 gesetzt werden (so dass die Registersatzausgänge an den ALU-Eingängen anliegen). Die ALUOp-Signale müssen für den Gleichheitstest auf 01 gesetzt werden (damit die ALU eine Subtraktion durchführt). Das PCWriteCond-Signal muss auf logisch 1 gesetzt werden, um den Befehlszähler zu aktualisieren, wenn der Zero-Ausgang der ALU auf logisch 1 gesetzt wird. Wenn PCSource auf 01 gesetzt wird, wird der in den Befehlszähler geschriebene Wert vom ALUOut-Register bereitgestellt, in dem die im vorhergehenden Zyklus berechnete Sprungzieladresse gespeichert ist. Bei ausgeführten bedingten Sprüngen bzw. bei ausgeführten Verzweigungen werden zweimal Werte in den Befehlszähler geschrieben: Einmal wird der Ausgangswert der ALU (während der Befehlentschlüsselungs- und Registerholphase) und einmal wird der ALUOut-Wert (während der Sprungausführungsphase) in den Befehlszähler geschrieben. Der zuletzt in den Befehlszähler geschriebene Wert wird für die nächste Befehlsholphase verwendet.

Sprung:

```
# {x, y} ist die Verilog-Notation für die Konkatenation der Bit-Felder x und y  
PC <= {PC [31:28], (IR [25:0], 2'b00)};
```

Operation: Der Befehlszähler wird durch die Sprungadresse ersetzt. PCSource wird so gesetzt, dass die Sprungadresse an den Befehlszähler geleitet wird, und PCWrite wird auf logisch 1 gesetzt, damit die Sprungadresse in den Befehlszähler geschrieben wird.

4. Speicherzugriffs- oder R-Befehlausführungsschritt

Während dieses Schritts greift ein Lade- oder Speicherbefehl auf den Speicher zu und ein arithmetisch-logischer Befehl schreibt das Ergebnis. Wenn vom Speicher ein Wert empfangen wird, wird dieser im Speicherdatenregister (MDR) gespeichert und muss im nächsten Taktzyklus verwendet werden.

Speicherzugriff:

```
MDR <= Memory [ALUOut];
```

oder

```
Memory [ALUOut] <= B;
```

Operation: Wenn es sich bei dem Befehl um einen Ladebefehl handelt, wird ein Datenwort vom Speicher empfangen und in das MDR geschrieben. Wenn es sich bei dem Befehl um einen Speicherbefehl handelt, werden die Daten in den Speicher geschrieben. In beiden Fällen handelt es sich bei der verwendeten Adresse um die Adresse.

die im vorhergehenden Schritt berechnet und im ALUOut-Register gespeichert wurde. Bei einem Speicherbefehl wird der Quelloperand in B gespeichert. (B wird zweimal gelesen: einmal in Schritt 2 und einmal in Schritt 3. Glücklicherweise wird beide Male derselbe Wert gelesen, da sich die Registeradresse, die in IR gespeichert und zum Lesen aus dem Registersatz verwendet wird, nicht ändert.) Das MemRead-Signal (bei einem Ladebefehl) oder das MemWrite-Signal (bei einem Speicherbefehl) muss auf logisch 1 gesetzt werden. Zudem wird bei Lade- und Speicherbefehlen das Signal IorD auf 1 gesetzt, damit nicht die vom Befehlszähler, sondern die von der ALU bereitgestellte Speicheradresse verwendet wird. Da das MDR bei jedem Taktzyklus geschrieben wird, muss kein Steuersignal explizit auf logisch 1 gesetzt werden.

Arithmetisch-logischer Befehl (R-Befehl):

```
Reg [IR [15:11]] <= ALUOut;
```

Operation: Speichere den Inhalt von ALUOut, der dem Ausgangswert der ALU-Operation im vorhergehenden Zyklus entspricht, im Ergebnisregister. Das RegDst-Signal muss auf 1 gesetzt werden, damit das rd-Feld (Bits 15:11) zum Auswählen des Registers zum Schreiben verwendet wird. RegWrite muss auf logisch 1 gesetzt werden, und MemtoReg muss auf 0 gesetzt werden, so dass nicht der Speicherdatenausgangswert, sondern der Ausgangswert der ALU in das Register geschrieben wird.

5. Speicherleseabschlusschritt

Während dieses Schritts werden Ladebefehle abgeschlossen, indem der Wert aus dem Speicher zurückgeschrieben wird.

Ladebefehl:

```
Reg [IR [20:16]] <= MDR;
```

Operation: Schreibe die geladenen Daten, die im vorhergehenden Zyklus im MDR gespeichert wurden, in den Registersatz. Dazu setzen wir MemtoReg auf 1 (um das Ergebnis aus dem Speicher zu schreiben), RegWrite auf logisch 1 (um einen Schreibvorgang zu veranlassen) und RegDst auf 0, um das rt-Feld (Bits 20:16) als Registeradresse auszuwählen.

Diese fünf Schritte sind in Tabelle 5.7 zusammengefasst. Anhand dieser Schritte können wir bestimmen, was die Steuerung bei den einzelnen Taktzyklen zu tun hat.

Definition der Steuerung

Nun, da wir festgelegt haben, welche Steuersignale vorhanden sein und wann diese auf logisch 1 gesetzt werden müssen, können wir die Steuereinheit implementieren. Für den Entwurf der Steuereinheit für den Eintaktdatenpfad haben wir mehrere Wahrheitstabellen verwendet, die auf der Grundlage der Befehlsklasse angegeben haben, wie die Steuersignale gesetzt werden. Beim Mehrzyklendatenpfad ist die Steuerung komplexer, da der Befehl in mehreren Schritten ausgeführt wird. Die Steuerung für den Mehrzyklendatenpfad muss sowohl die Signale angeben, die in einem Schritt gesetzt werden, als auch die Signale, die im nachfolgenden Schritt gesetzt werden.

In diesem Abschnitt und in [Abschnitt 5.7](#) auf CD werden wir zwei unterschiedliche Methoden zum Definieren der Steuerung untersuchen. Die erste Methode beruht auf endlichen Automaten, die für gewöhnlich grafisch dargestellt werden. Die zweite Methode, die als **Mikroprogrammierung (microprogramming)** bezeichnet wird, verwendet für die Steuerung eine Programmdarstellung. Mit beiden Methoden wird die Steuerung in einer Form dargestellt, die es ermöglicht, dass die detaillierte Implementierung mit Gattern, Festwertspeichern oder PLAs durch ein CAD-System synthetisiert



Mikroprogramm (micro-program) Eine symbolische Darstellung der Steuerung in Form von Befehlen, die als **Mikrobefehle** bezeichnet und in einem einfachen Mikroprogrammwerk ausgeführt werden.

Tab. 5.7 Übersicht über die für die Ausführung aller Befehlsklassen erforderlichen Schritte bzw. Befehlausführungsphasen. Die Befehle beanspruchen zwischen drei und fünf Ausführungsphasen. Die ersten beiden Phasen sind von der Befehlsklasse unabhängig. Nach diesen Phasen muss der Befehl je nach Befehlsklasse zwischen einem und drei weiteren Zyklen ausführen. Die leeren Einträge für die Speicherzugriffsphase oder die Speicherleseabschlussphase weisen darauf hin, dass die jeweilige Befehlsklasse weniger Zyklen beansprucht. In einer Mehrzyklenimplementierung wird ein neuer Befehl gestartet, sobald der aktuelle Befehl abgeschlossen ist, so dass sich keine Zyklen im Leerlauf befinden oder vergeudet werden. Wie bereits erwähnt, wird der Registersatz bei jedem Zyklus gelesen. Solange das IR unverändert bleibt, sind die aus dem Registersatz gelesenen Werte jedoch identisch. Insbesondere der während der Befehlsentschlüsselungsphase aus Register B gelesene Wert für eine Verzweigung oder einen R-Befehl ist mit dem Wert identisch, der während der Ausführungsphase in Register B gespeichert und anschließend in der Speicherzugriffsphase für eine store-word-Befehl verwendet wurde.

Befehlausführungsphase	Aktion bei R-Befehlen	Aktion bei Speicherreferenzbefehlen	Aktion bei Verzweigungen	Aktion bei Sprüngen	
Befehlsholphase		IR <= Memory[PC] PC <= PC + 4			
Befehlsentschlüsselungs-/ Registerholphase		A <= Reg [IR[25:21]] B <= Reg [IR[20:16]] ALUOut <= PC +(sign-extend (IR[15:0]) << 2)			
Ausführung, Adressberechnung, Verzweigungs-/ Sprungausführung	ALUOut <= A op B	ALUOut <= A + sign-extend (IR[15:0])	if (A == B) PC <= ALUOut	PC <= PC [31:28], (IR[25:0]),2'b00	
Speicherzugriff oder R-Befehlausführung	Reg [IR[15:11]] <= ALUOut	Load: MDR <= Memory[ALUOut] or Store: Memory [ALUOut] <= B			
Speicherleseabschluss		Load: Reg[IR[20:16]] <= MDR			

wird. In diesem Kapitel konzentrieren wir uns auf den Entwurf der Steuerung und die Darstellung in diesen beiden Formen.



In **Abschnitt 5.8** (auf CD) ist anhand von Beispielen für den Mehrzyklendatenpfad sowie für die Steuerung eines endlichen Automaten dargestellt, wie Hardwarebeschreibungssprachen zum Entwerfen moderner Prozessoren verwendet werden. Beim Entwurf moderner Digitalsysteme wird der letzte Schritt, bei dem eine Hardwarebeschreibung in konkrete Gatter umgesetzt wird, mithilfe von Synthesewerkzeugen für die Logik und den Datenpfad realisiert. In **Appendix C** ist beschrieben, wie dieser Prozess durch Übersetzen der Mehrzyklusteuereinheit in eine detaillierte Hardwareimplementierung funktioniert. Die wichtigsten Konzepte der Steuerung werden in diesem Kapitel vermittelt und sind ohne die Informationen in **Abschnitt 5.8** bzw. in **Appendix C** (auf CD) verständlich. Wenn Sie jedoch Hardware entwerfen möchten, ist Abschnitt 5.9 hilfreich, und in **Appendix C** finden Sie Informationen dazu, wie die Implementierungen auf der Gatterebene aussehen werden.



Anhand dieser Implementierung und dem Wissen, dass jeder Zustand einen Taktzyklus beansprucht, können wir den CPI-Wert für einen typischen Befehlsmix ermitteln.

BEISPIEL

CPI-Wert in einer Mehrzyklen-CPU

Wie lautet der CPI-Wert, wenn wir den SPECINT2000-Befehlsmix aus Tabelle 3.14 verwenden und voraussetzen, dass jeder Zustand in einer Mehrzyklen-CPU einen Taktzyklus erfordert?

ANTWORT

Der Mix besteht aus 25 % Ladebefehlen (1 % load-byte-Befehle + 24 % load-word-Befehle), 10 % Speicherbefehlen (1 % store-byte-Befehle + 9 % store-word-Befehle), 11 % Verzweigungen (6 % beq-Befehle, 5 % bne-

Befehle), 2 % Sprünge (1 % ja1-Befehle + 1 % jr-Befehle) und 52 % ALU-Befehlen (der gesamte Rest des Befehlsmix, von dem wir annehmen, dass er aus ALU-Befehlen besteht). Aus Tabelle 5.7 ergibt sich, dass sich die Anzahl der Taktzyklen für die einzelnen Befehlsklassen wie folgt zusammensetzt:

- Ladebefehle: 5
- Speicherbefehle: 4
- ALU-Befehle: 4
- Verzweigungen: 3
- Sprünge: 3

Für den CPI-Wert ergibt sich Folgendes:

$$\begin{aligned} \text{CPI} &= \frac{\text{CPU-Taktzyklen}}{\text{Befehlszahl}} = \frac{\sum \text{Befehlszahl}_i \times \text{CPI}_i}{\text{Befehlszahl}} \\ &= \sum \frac{\text{Befehlszahl}_i}{\text{Befehlszahl}} \times \text{CPI}_i \end{aligned}$$

Das Verhältnis

$$\frac{\text{Befehlszahl}_i}{\text{Befehlszahl}}$$

ist einfach die Befehlhäufigkeit für die Befehlsklasse i . Wir können somit ersetzen und erhalten Folgendes

$$\text{CPI} = 0,25 \times 5 + 0,10 \times 4 + 0,52 \times 4 + 0,11 \times 3 + 0,02 \times 3 = 4,12$$

Dieser CPI-Wert ist besser als der Worst-Case-CPI-Wert 5,0, bei dem alle Befehle dieselbe Anzahl an Taktzyklen beanspruchen. Dieser Unterschied kann jedoch durch Overheads bei beiden Entwürfen verkleinert oder vergrößert werden. Der Mehrzyklenentwurf ist wahrscheinlich auch kostengünstiger, da im Datenpfad weniger Einzelkomponenten verwendet werden.

Bei der ersten Methode zum Definieren der Mehrzyklensteuerung verwenden wir einen **endlichen Automaten (finite state machine)**. Ein endlicher Automat besteht aus mehreren Zuständen und Angaben zum Ändern der Zustände. Diese Angaben werden durch eine **Zustandsabbildungsfunktion (next-state function)** definiert, die den aktuellen Zustand und die Eingänge auf einen neuen Zustand abbildet. Wenn wir für die Steuerung einen endlichen Automaten verwenden, gibt jeder Zustand zudem eine Reihe von Ausgängen an, die auf logisch 1 gesetzt (aktiviert) werden, wenn sich der Automat in diesem Zustand befindet. Die Implementierung eines endlichen Automaten setzt für gewöhnlich voraus, dass alle Ausgänge, die nicht explizit auf logisch 1 gesetzt sind, automatisch auf logisch 0 gesetzt sind. Entsprechend hängt die richtige Arbeitsweise des Datenpfads von der Tatsache ab, dass ein Signal, das nicht ausdrücklich auf logisch 1 gesetzt ist, auf logisch 0 und nicht auf Don't-care gesetzt ist. So darf beispielsweise das RegWrite-Signal nur auf logisch 1 gesetzt sein, wenn ein Registersatzeintrag geschrieben werden muss. Wenn es nicht ausdrücklich auf logisch 1 gesetzt ist, muss es auf logisch 0 gesetzt sein.

Multiplexersteuerungen sind etwas anders, da diese einen der Eingänge auswählen, gleichgültig ob diese auf 0 oder 1 gesetzt sind. Daher legen wir für alle Multiplexer-Steuersignale in einem endlichen Automaten, die uns wichtig sind, immer fest, ob diese gesetzt werden. Wenn wir den endlichen Automaten mit Logikbausteinen implementieren, kann das Steuersignal standardmäßig auf 0 gesetzt sein und es werden

Endlicher Automat (finite state machine) Eine sequentielle logische Funktion, die aus einer Menge von Eingängen und Ausgängen, aus einer Zustandsabbildungsfunktion, die den aktuellen Zustand und die Eingänge auf einen neuen Zustand abbildet, und aus einer Ausgangsfunktion besteht, die den aktuellen Zustand und möglicherweise die Eingänge auf mehrere auf logisch 1 gesetzte Ausgänge abbildet.

Zustandsabbildungsfunktion (next-state function) Eine kombinatorische Funktion, die anhand der Eingänge und des aktuellen Zustands den nächsten Zustand eines endlichen Automaten bestimmt.

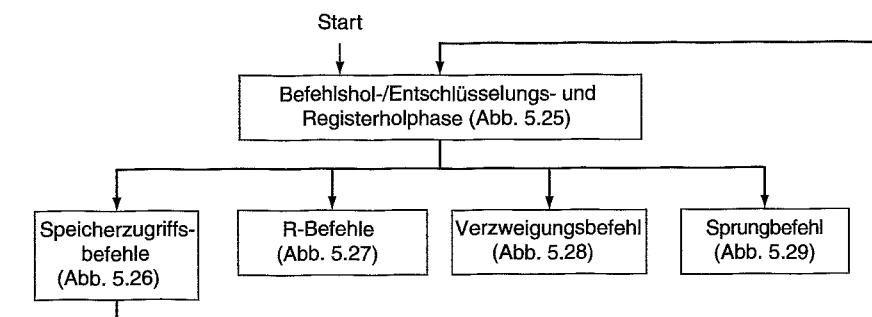


Abb. 5.24 Die abstrakte Darstellung der Steuerung durch einen endlichen Automaten. Die ersten Schritte sind von der Befehlsklasse unabhängig. Danach werden mehrere, vom Opcode des Befehls abhängige Sequenzen zum Ausführen der jeweiligen Befehlsklasse verwendet. Nach dem Ausführen der für diese Befehlsklassen erforderlichen Aktionen kehrt die Steuerung zurück, um einen neuen Befehl zu holen. Die einzelnen Kästchen in dieser Abbildung können mehrere Zustände darstellen. Der mit *Start* bezeichnete Pfeil kennzeichnet den Zustand, in dem mit dem Holen des ersten Befehls begonnen wird.

somit keine Gatter benötigt. Ein einfaches Beispiel für einen endlichen Automaten ist in **Appendix B** (auf CD) dargestellt. Und wenn Sie mit dem Prinzip des endlichen Automaten nicht vertraut sind, sollten Sie vor dem Weiterlesen **Appendix B** lesen.

Die Steuerung eines endlichen Automaten entspricht im Wesentlichen den fünf auf den Seiten 268–271 beschriebenen Ausführungsschritten. Jeder Zustand in einem endlichen Automaten beansprucht einen Taktzyklus. Der endliche Automat besteht aus mehreren Teilen. Da die ersten beiden Ausführungsschritte bei allen Befehlen gleich sind, sind auch die ersten beiden Zustände des endlichen Automaten für alle Befehle gleich. Die Schritte 3 bis 5 hängen vom Opcode ab. Nach der Ausführung des letzten Schritts für eine bestimmte Befehlsklasse kehrt der endliche Automat zum ersten Zustand zurück und beginnt wieder damit, den nächsten Befehl zu holen.

In Abbildung 5.24 ist diese abstrakte Darstellung des endlichen Automaten zu sehen. Um die Details des endlichen Automaten einzufügen, werden wir zunächst den Befehlshol- und Befehlsentschlüsselungsteil erweitern und dann die Zustände (und Aktionen) für die unterschiedlichen Befehlsklassen darstellen.

Die ersten beiden Zustände des endlichen Automaten sind in Abbildung 5.25 in Form einer herkömmlichen grafischen Darstellung zu sehen. Um die Abbildung leichter erläutern zu können, nummerieren wir die Zustände, wobei die Nummern willkürlich gewählt sind. Zustand 0, der Schritt 1 entspricht, ist der Anfangszustand des Automaten.

Die in den einzelnen Zuständen auf logisch 1 gesetzten Signale sind in dem Kreis (bzw. Knoten) abgebildet, der den Zustand darstellt. Die Pfeile zwischen den Zuständen definieren den nächsten Zustand und sind mit den Bedingungen markiert, anhand derer ein bestimmter nächster Zustand ausgewählt wird, wenn mehrere nächste Zustände möglich sind. Nach Zustand 1 hängt es von der Befehlsklasse ab, welche Signale auf logisch 1 gesetzt sind. Daher gehen von Zustand 1 des endlichen Automaten vier Pfeile (bzw. gerichtete Kanten) weg, die den vier Befehlsklassen entsprechen: Speicherreferenz, R-Befehl, branch-on-equal-Befehl und jump-Befehl. Dieser Vorgang, bei dem abhängig vom Befehl in unterschiedliche Zustände verzweigt wird, wird als *Entschlüsselung* bezeichnet, da die Auswahl des nächsten Zustands und damit der Aktionen, die folgen, von der Befehlsklasse abhängen.

In Abbildung 5.26 ist der Teil des endlichen Automaten dargestellt, der zum Implementieren der Speicherreferenzbefehle benötigt wird. Bei den Speicherreferenzbefehlen wird im ersten Zustand nach dem Holen des Befehls und der Register die Spei-

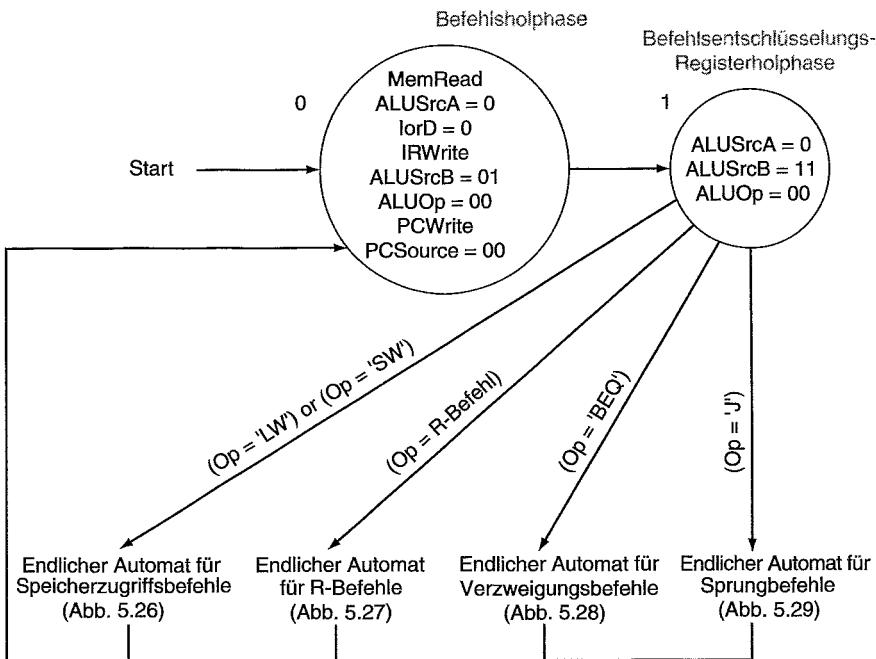


Abb. 5.25 Der Befehlshol- und Befehlsentschlüsselungsteil ist bei allen Befehlen gleich. Die Zustände entsprechen dem oberen Kästchen in der Darstellung des endlichen Automaten in Abbildung 5.24. Im ersten Zustand setzen wir zwei Signale auf logisch 1, damit der Speicher einen Befehl liest und in das Befehlsregister schreibt (MemRead und IRWrite), und wir setzen IorD auf 0, um den Befehlszähler als die Adressquelle auszuwählen. Die Signale ALUSrcA, ALUSrcB, ALUOp, PCWrite und PCSource werden so gesetzt, dass der Wert Befehlszählerwert + 4 berechnet und im Befehlszähler gespeichert wird. (Dieser Wert wird auch in ALUOut gespeichert, aber von dort aus nie verwendet.) Im nächsten Zustand berechnen wir die Sprungzieladresse, indem wir ALUSrcB auf 11 (dadurch werden die verschobenen und vorzeichenerweiterten unteren 16 Bits des IR an die ALU gesendet), ALUSrcA auf 0 und ALUOp auf 00 setzen. Wir speichern das Ergebnis im ALUOut-Register, in das bei jedem Zyklus geschrieben wird. Es folgen vier weitere Zustände, die von der Klasse des Befehls abhängen, der während dieses Zustands bekannt wird. Mit dem als *Op* bezeichneten Eingang der Steuereinheit wird bestimmt, welche dieser Pfeile verfolgt werden soll. Denken Sie daran, dass alle Signale, die nicht ausdrücklich auf logisch 1 gesetzt sind, automatisch auf logisch 0 gesetzt sind. Das ist bei Signalen, die Schreibbefehle steuern, besonders wichtig. Wenn bei Multiplexer-Steuersignalen nicht angegeben ist, ob diese gesetzt sind, zeigt das, dass die Einstellung des Multiplexers nicht wichtig ist.

cheradresse berechnet (Zustand 2). Zum Berechnen der Speicheradresse müssen die Multiplexer am ALU-Eingang so gesetzt sein, dass am ersten Eingang das Register A anliegt, während am zweiten Eingang das vorzeichenerweiterte Displacement-Feld anliegt. Das Ergebnis wird in das ALUOut-Register geschrieben. Nach dem Berechnen der Speicheradresse muss der Speicher gelesen oder beschrieben werden. Dazu sind zwei verschiedene Zustände erforderlich. Wenn der Opcode des Befehls *lw* lautet, wird der Speicher in Zustand 3 (entsprechend dem Speicherzugriffsschritt) gelesen (MemRead ist auf logisch 1 gesetzt). Der Ausgangswert des Speichers wird immer in MDR geschrieben. Wenn der Opcode des Befehls *sw* lautet, wird in Zustand 5 in den Speicher geschrieben (MemWrite ist auf logisch 0 gesetzt). In den Zuständen 3 und 5 ist das Signal IorD auf 1 gesetzt, um zu erzwingen, dass die von der ALU bereitgestellte Speicheradresse verwendet wird. Nach dem Ausführen eines Schreibbefehls ist die Ausführung des Befehls *sw* abgeschlossen und der nächste Zustand ist nun Zustand 0. Wenn es sich um einen Ladebefehl handelt, ist jedoch ein weiterer Zustand (Zustand 4) erforderlich, um das Ergebnis aus dem Speicher in den Registersatz zu schreiben. Wenn das

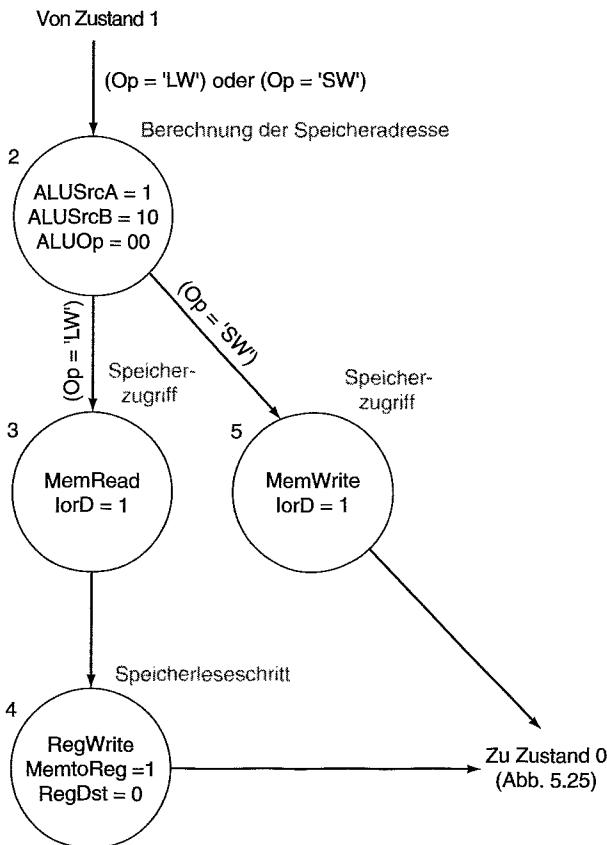


Abb. 5.26 Der endliche Automat zum Steuern von Speicherreferenzbefehlen besitzt vier Zustände. Diese Zustände entsprechen dem in Abbildung 5.24 mit „Speicherzugriffsbefehle“ beschrifteten Kästchen. Nach dem Berechnen der Speicheradresse wird für den Ladebefehl und für den Speicherbefehl eine getrennte Sequenz benötigt. Durch Setzen der Steuersignale ALUSrcA, ALUSrcB und ALUOp wird die Berechnung der Speicheradresse in Zustand 2 veranlasst. Ladebefehle erfordern einen zusätzlichen Zustand, um das Ergebnis aus dem MDR (dort wird das Ergebnis in Zustand 3 geschrieben) in den Registersatz zu schreiben.

Multiplexer-Steuersignal MemtoReg auf 1 und das Multiplexer-Steuersignal RegDst auf 0 gesetzt wird, wird der in MDR geladene Wert in den Registersatz geschrieben, wobei rt als Registeradresse verwendet wird. Nach diesem Zustand, der dem letzten Speicherleseschritt entspricht, kommt wieder Zustand 0.

Um den R-Befehl zu implementieren, sind zwei Zustände erforderlich, die den Schritten 3 (Ausführung) und 4 (Ausführung von R-Befehlen) entsprechen. In Abbildung 5.27 ist dieser aus zwei Zuständen bestehende Teil des endlichen Automaten dargestellt. In Schritt 6 werden ALUSrcA auf logisch 1 und die ALUSrcB-Signale auf 00 gesetzt. Dadurch wird erzwungen, dass die beiden Registerwerte, die aus dem Registersatz ausgelesen wurden, als Eingangswerte für die ALU verwendet werden. Wenn ALUOp auf 10 gesetzt wird, verwendet die ALU-Steuereinheit das funct-Feld zum Setzen der ALU-Steuersignale. In Zustand 7 wird RegWrite auf logisch 1 gesetzt, um Werte in den Registersatz zu schreiben, RegDst wird auf logisch 1 gesetzt, damit das rd-Feld als Registeradresse für das Ziel verwendet wird, und MemtoReg wird auf logisch 0 gesetzt, damit ALUOut als Quelle für den Wert verwendet wird, der in den Registersatz geschrieben wird.

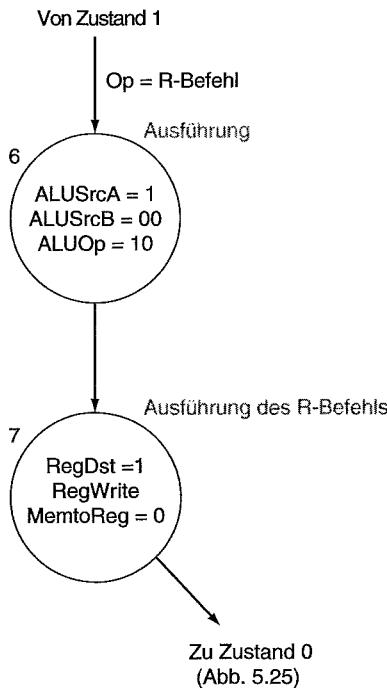


Abb. 5.27 R-Befehle können mit einem einfachen endlichen Automaten mit zwei Zuständen implementiert werden. Diese Zustände entsprechen dem in Abbildung 5.24 mit „R-Befehle“ beschrifteten Kästchen. Im ersten Zustand wird eine ALU-Operation durchgeführt, während im zweiten Zustand das Ergebnis der ALU-Operation (das sich in ALUOut befindet) in den Registersatz geschrieben wird. Durch die drei Signale, die in Zustand 7 auf logisch 1 gesetzt werden, wird der Inhalt von ALUOut in den Registersatz in das durch das rd-Feld des Befehlsregisters angegebene Register geschrieben.

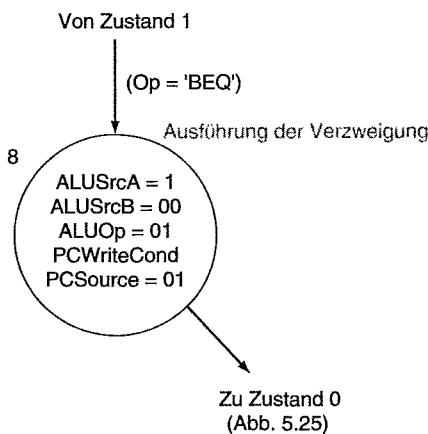


Abb. 5.28 Der Verzweigungsbefehl erfordert nur einen Zustand. Die ersten drei Ausgangssignale, die auf logisch 1 gesetzt sind, verlassen die ALU, die Register (ALUSrcA, ALUSrcB und ALUOp) zu vergleichen, während die Signale PCSource und PCWriteCond einen Schreibbefehl ausführen, sofern die Sprungbedingung erfüllt ist. Der in ALUOut geschriebene Wert wird nicht verwendet. Stattdessen wird nur das Zero-Ausgangssignal der ALU verwendet. Die Sprungzieladresse wird aus ALUOut ausgelesen, wo sie am Ende von Zustand 1 gespeichert wurde.

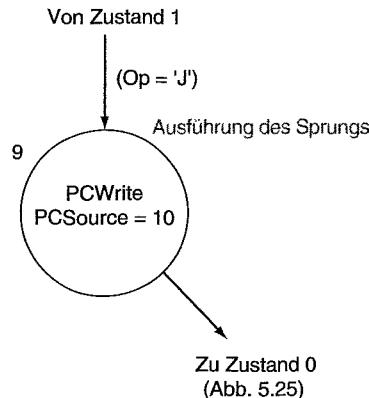


Abb. 5.29 Der Sprungbefehl erfordert einen Zustand, in dem zwei Steuersignale auf logisch 1 gesetzt werden, damit die um zwei Bit nach links verschobenen und mit den oberen vier Bits des Befehlsregisters dieses Befehls verknüpften unteren 26 Bits des Befehlsregisters in das Befehlsregister geschrieben werden.

Bei Verzweigungen bzw. bedingten Sprüngen ist nur ein weiterer Zustand erforderlich, da diese die Ausführung im dritten Befehlausführungsschritt abschließen. In diesem Zustand müssen die Steuersignale gesetzt werden, die veranlassen, dass die ALU die Inhalte der Register A und B miteinander vergleicht. Darüber hinaus werden auch die Signale gesetzt, die dafür sorgen, dass die Adresse im ALUOut-Register in den Befehlszähler geschrieben wird, sofern die Schreibbedingung erfüllt ist. Um den Vergleich durchführen zu können, müssen wir ALUSrcA auf logisch 1 und ALUSrcB auf 00 und den ALUOp-Wert auf 01 (erzwingt eine Subtraktion) setzen. (Wir verwenden nur den Zero-Ausgangswert der ALU, nicht das Ergebnis der Subtraktion.) Um den Schreibvorgang beim Befehlszähler zu steuern, setzen wir PCWriteCond auf logisch 1 und PCSource auf 01, wodurch der im ALUOut-Register (enthält die in Zustand 1, Abbildung 5.25 berechnete Sprungadresse) befindliche Wert in den Befehlszähler geschrieben wird, wenn das Zero-Bit der ALU auf logisch 1 gesetzt ist. In Abbildung 5.28 ist dieser Zustand dargestellt.

Die letzte Befehlsklasse ist der Sprung. Wie beim bedingten Sprung bzw. bei der Verzweigung ist für dessen Ausführung ebenfalls nur ein weiterer Zustand erforderlich (in Abbildung 5.29 dargestellt). In diesem Zustand wird das PCWrite-Signal auf logisch 1 gesetzt, damit ein Wert in den Befehlszähler geschrieben wird. Wenn PC-Source auf 10 gesetzt wird, wird zum Schreiben der Wert bereitgestellt, der die unteren 26 Bits des Befehlsregisters umfasst, wobei 00_B als die mit den oberen vier Bits des Befehlszählers verknüpften niederwertigen Bits addiert werden.

Nun können wir diese Teile des endlichen Automaten in einer Beschreibung der Steuereinheit wie in Abbildung 5.31 zusammenfügen. Hier sind die in jedem Zustand auf logisch 1 gesetzten Signale dargestellt. Der nächste Zustand hängt von den Opcode-Bits des Befehls ab. Daher markieren wir die Kanten mit einem Abgleich für die entsprechenden Opcodes des Befehls.

Ein endlicher Automat kann mit einem temporären Register implementiert werden, in dem der aktuelle Zustand gespeichert wird und einem kombinatorischen Logikblock, mit dem sowohl die auf logisch 1 zu setzenden Datenpfadsignale als auch der nächste Zustand bestimmt werden. In Abbildung 5.30 ist dargestellt, wie eine Implementierung dieser Art aussehen kann. In Appendix C.3 (auf CD) wird ausführlich beschrieben, wie der endliche Automat mithilfe dieser Struktur implementiert wird. Es wird die kombinatorische Steuerlogik für den endlichen Automaten aus Abbildung 5.31 sowohl mit einem Festwertspeicher (ROM, Read Only Memory) als auch mit einem PLA



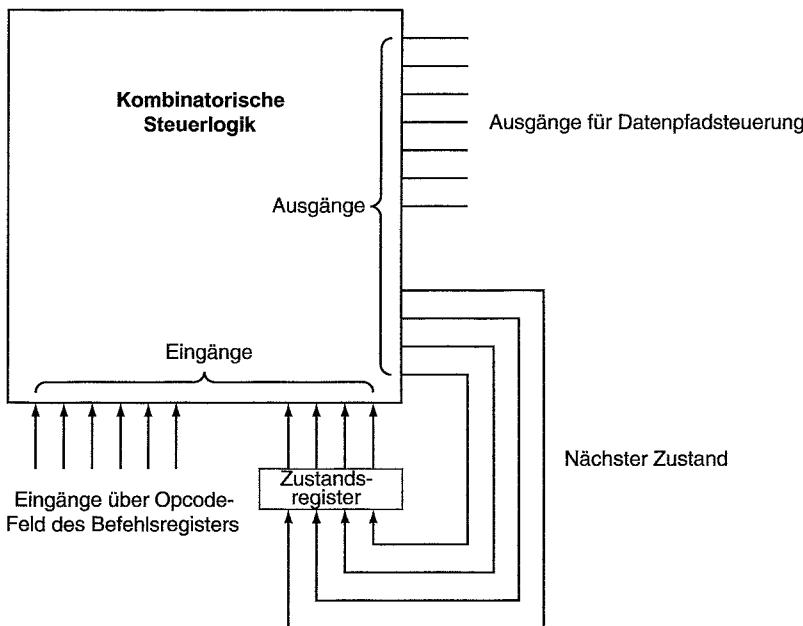


Abb. 5.30 Steuerungen mit endlichen Automaten werden für gewöhnlich mit einem kombinatorischen Logikblock und einem Register zum Speichern des aktuellen Zustands implementiert. Die Ausgangswerte der kombinatorischen Logik umfassen die Bezeichnung des nächsten Zustands und die Steuersignale, die für den aktuellen Zustand auf logisch 1 gesetzt werden müssen. Die Eingangswerte der kombinatorischen Logik umfassen den aktuellen Zustand sowie sämtliche Eingangswerte, die zum Bestimmen des nächsten Zustands verwendet werden. In diesem Beispiel sind die Opcode-Bits des Befehlsregisters die Eingangswerte. Die Ausgangswerte des in diesem Kapitel verwendeten endlichen Automaten hängen nicht von den Eingangswerten, sondern nur vom aktuellen Zustand ab. Im Abschnitt „Vertiefung“ wird dies ausführlicher erläutert.

(Programmable Logic Array) implementiert. (Diese Logikelemente werden in **Appendix B** beschrieben.) Im nächsten Abschnitt in diesem Kapitel betrachten wir eine weitere Methode für die Darstellung der Steuerung. Bei den beiden Methoden handelt es sich lediglich um unterschiedliche Darstellungen derselben Steuerungsinformationen.

Pipelining, das Thema von Kapitel 6, wird nahezu immer zum Beschleunigen der Ausführung von Befehlen verwendet. Bei einfachen Befehlen kann mit Pipelining die höhere Taktrate eines Mehrzyklenentwurfs und ein Eintakt-CPI-Wert eines Eintaktentwurfs erzielt werden. Bei den meisten Prozessoren mit Pipelining beanspruchen einige Befehle jedoch mehr als einen Zyklus und benötigen eine Mehrzyklensteuerung. Gleitkommabefehle sind ein allgemeines Beispiel hierfür. Es gibt viele Beispiele in der IA-32-Architektur, die eine Mehrzyklensteuerung erfordern.

Vertiefung: Der in Abbildung 5.30 dargestellte endliche Automat wird nach Edward Moore als *Moore-Automat* bezeichnet. Dieser Automat ist dadurch gekennzeichnet, dass die Ausgabe ausschließlich vom aktuellen Zustand abhängt. Bei einem Moore-Automaten kann das mit „kombinatorische Steuerlogik“ beschriftete Kästchen in zwei Teile geteilt werden. Der eine Teil umfasst das Ausgangssignal der Steuerung und nur das Eingangssignal für den aktuellen Zustand, während der andere Teil nur das Ausgangssignal für den nächsten Zustand enthält.

Eine andere Art Automat ist der nach George Mealy benannte *Mealy-Automat*. Beim Mealy-Automaten können sowohl das Eingangssignal als auch der aktuelle Zustand zum Bestimmen der Ausgabe verwendet werden. Moore-Automaten weisen hinsichtlich der Geschwindigkeit und der Größe der Steuereinheit potenzi-

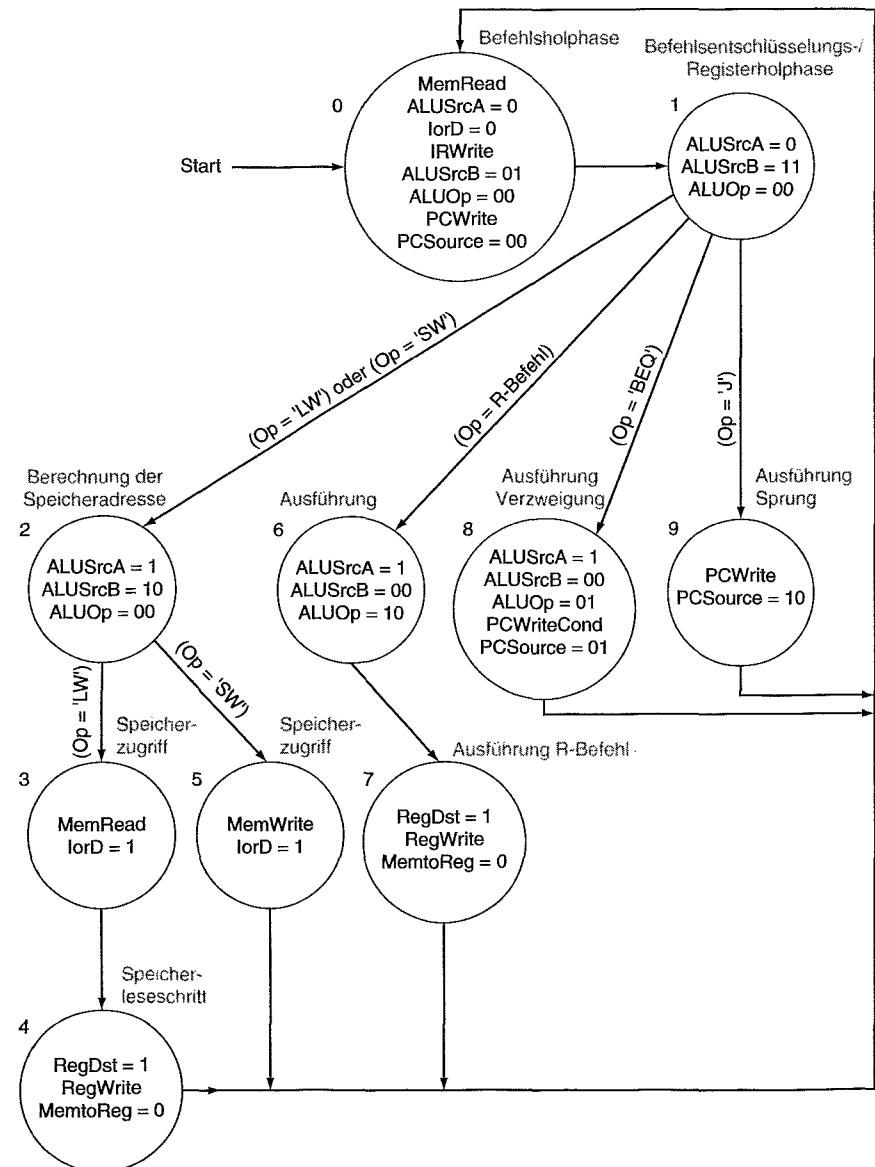


Abb. 5.31 Die vollständige Steuerung mit endlichen Automaten für den in Abbildung 5.23 dargestellten Datenpfad. Die Markierungen der Kanten geben die Bedingungen an, die überprüft werden, um zu bestimmen, welcher Zustand als nächster folgt. Wenn der nächste Zustand ein unbedingter Zustand ist, ist die Kante nicht markiert. Die Markierungen in den Knoten kennzeichnen die Ausgangssignale, die während des jeweiligen Zustands auf logisch 1 gesetzt werden. Das Setzen eines Multiplexer-Steuersignals geben wir immer an, wenn dies für den ordnungsgemäßen Betrieb erforderlich ist. In einigen Zuständen wird ein Multiplexer-Steuersignal auf 0 gesetzt.

elle Vorteile bei der Implementierung auf. Der Vorteil hinsichtlich der Geschwindigkeit ergibt sich aus der Tatsache, dass die Ausgangssignale der Steuerung, die früh im Taktzyklus benötigt werden, nicht von den Eingangssignalen, sondern ausschließlich vom aktuellen Zustand abhängen. In Appendix C, in dem die Implementierung dieses endlichen Automaten mithilfe von logischen Gattern beschrieben wird, macht sich der Vorteil hinsichtlich der Größe deutlich bemerkbar. Ein Moore-Automat hat möglicherweise den Nachteil, dass zusätzliche Zustände erforderlich



sind. In Situationen, in denen sich zwei Zustandssequenzen um einen Zustand unterscheiden, kann der Mealy-Automat die Zustände beispielsweise zusammenfassen, indem er dafür sorgt, dass die Ausgangssignale von den Eingangssignalen abhängen.

Bei einem Prozessor mit einer gegebenen Taktrate wird die relative Leistung zweier Codesegmente durch das Produkt aus dem CPI-Wert und der zum Ausführen des jeweiligen Segments erforderlichen Befehlszahl bestimmt. Wie wir hier gesehen haben, können Befehle auch bei einem einfachen Prozessor unterschiedliche CPI-Werte aufweisen. In den nachfolgenden beiden Kapiteln werden wir sehen, dass die Einführung des Pipelining und die Nutzung von Cache-Speichern noch mehr Möglichkeiten für unterschiedliche CPI-Werte bieten. Viele Faktoren, die Auswirkungen auf den CPI-Wert haben, werden zwar vom Hardwareentwickler gesteuert, aber der Programmierer, der Compiler und das Softwaresystem bestimmen, welche Befehle ausgeführt werden. Und gerade dieser Prozess bestimmt den effektiven CPI-Wert des Programms. Programmierer, die die Leistungsfähigkeit verbessern möchten, müssen wissen, welche Rolle der CPI-Wert spielt, und sie müssen die Faktoren kennen, die sich auf den CPI-Wert auswirken.

Hardware-Software-Schnittstelle

1. Richtig oder falsch: Da der Sprungbefehl weder von den Registerwerten noch von der Berechnung der Sprungzieladresse abhängt, kann er statt im dritten bereits im zweiten Zustand ausgeführt werden.
2. Richtig, falsch oder eventuell: Das Steuersignal PCWriteCond kann durch das Signal PCSOURCE[0] ersetzt werden.



5.6 Ausnahmebehandlung

Die Steuerung stellt beim Prozessordesign die größte Herausforderung dar: Es ist nicht nur schwierig, diesen Teil richtig zu entwerfen, sondern auch, ihn hinsichtlich der Geschwindigkeit zu optimieren. Einer der schwierigsten Bereiche der Steuerung ist die Implementierung von **Ausnahmen (exceptions)** und **Unterbrechungen (interrupts)**, Ereignisse, die sich von Sprüngen unterscheiden und den normalen Fluss der Befehlausführung ändern. Bei einer Ausnahme handelt es sich um ein unerwartetes Ereignis im Prozessor. Der arithmetische Überlauf ist ein Beispiel für eine Ausnahme. Eine Unterbrechung ist ein Ereignis, das ebenfalls eine unerwartete Änderung des Steuerflusses verursacht, jedoch außerhalb des Prozessors verursacht wird. Wie wir in Kapitel 8 sehen werden, werden Unterbrechungen z.B. von Ein-/Ausgabegeräten zum Kommunizieren mit dem Prozessor verwendet.

Viele Architekturen und viele Autoren unterscheiden nicht zwischen Unterbrechungen und Ausnahmen und verwenden häufig für beide Ereignistypen die ältere Bezeichnung *Unterbrechung*. Wir halten uns an die MIPS-Konvention und verwenden den Ausdruck *Ausnahme* für jede unerwartete Änderung des Steuerflusses, ohne zu unterscheiden, ob die Ursache dafür intern oder extern zu suchen ist. Den Ausdruck *Unterbrechung* verwenden wir nur, wenn das Ereignis extern verursacht wird. Bei der Intel IA-32-Architektur wird die Bezeichnung *Unterbrechung* für alle Ereignisse dieser Art verwendet.

Ausnahme (exception) Auch als *Unterbrechung* bezeichnet. Ein nicht geplantes Ereignis, das die Programmausführung unterbricht. Wird zum Erkennen von Überläufen verwendet.

Unterbrechung (interrupt) Eine Ausnahme, die außerhalb des Prozessors verursacht wird. (Bei manchen Architekturen wird der Ausdruck *Unterbrechung* für alle Ausnahmen verwendet.)