

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-4752-313

Universitätsbibliothek Bern
Zentralbibliothek

2007



Spektrum
AKADEMISCHER VERLAG

IT 456 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

Never waste time.

Amerikanisches Sprichwort

Pipelining Eine Implementierungstechnik, bei der mehrere Befehle ähnlich wie bei einem Fließband überlappend ausgeführt werden.

6.1

Übersicht über die Technik des Pipelining

Pipelining ist eine Implementierungstechnik, bei der mehrere Befehle überlappend ausgeführt werden. Bei heutigen Rechnern stellt das Pipelining die Schlüsseltechnik zum Beschleunigen von Prozessoren dar.

In diesem Abschnitt werden die zentralen Begriffe und Problemstellungen im Zusammenhang mit dem Pipelining anhand eines Vergleichs erläutert. Wenn Sie lediglich an einer Übersicht interessiert sind, sollten Sie sich auf diesen Abschnitt konzentrieren und anschließend die Abschnitte 6.9 und 6.10 lesen. Dort finden Sie Informationen zu verbesserten Pipelining-Techniken, die in neueren Prozessoren wie dem Pentium III und Pentium 4 eingesetzt werden. Wenn es Sie interessiert, wie ein Computer mit Pipelines intern aufgebaut ist, ist dieser Abschnitt eine gute Einführung in die Abschnitte 6.2 bis 6.8.

Jeder, der viel Wäsche wäschte, wendet intuitiv eine Art Pipelining-Technik an. Wäschewaschen *ohne Pipelining* funktioniert folgendermaßen:

1. Füllen der Waschmaschine mit einer Ladung schmutziger Kleidung.
2. Nach dem Waschen umfüllen der Wäsche aus der Waschmaschine in den Wäschetrockner.
3. Nach dem Trocknen Wäsche zusammenlegen.
4. Nach dem Zusammenlegen einen Mitbewohner bitten, die Wäsche in den Schrank zu räumen.

Erst wenn Ihr Mitbewohner fertig ist, beginnen Sie mit der nächsten Waschladung von vorn.

Das Wäschewaschen *mit Pipelining* braucht viel weniger Zeit (siehe Abbildung 6.1). Sobald die Waschmaschine die erste Ladung gewaschen hat und die nasse Wäsche im Trockner ist, können Sie die Waschmaschine mit der zweiten Ladung Schmutzwäsche befüllen. Wenn die erste Ladung trocken ist, können Sie mit Zusammenlegen beginnen, die nasse Ladung in den Trockner geben und die nächste Ladung Schmutzwäsche in die Waschmaschine füllen. Als Nächstes kann Ihr Mitbewohner die erste Ladung in den Schrank räumen, Sie können die zweite Ladung zusammenlegen, die dritte Ladung wird im Trockner getrocknet und die vierte in der Waschmaschine gewaschen. Zu diesem Zeitpunkt sind alle so genannten *Pipelinstufen* aktiv. Solange für jede Stufe eigene Ressourcen verfügbar sind, können wir die Aufgaben mittels Pipelining durchführen.

Das Paradoxe am Pipelining ist, dass die Zeit vom Befüllen der Waschmaschine mit einer Ladung bis zum Trocknen, Zusammenlegen und Wegräumen dieser Ladung in den Schrank beim Pipelining nicht kürzer ist. Das Pipelining ist für viele Ladungen Wäsche nur deshalb schneller, weil die Ladungen parallel verarbeitet werden und daher mehr Wäscheladungen pro Stunde gewaschen werden können. Mit dem Pipelining lässt sich der Durchsatz unseres Wäschewaschsystems verbessern, ohne die Zeit für die Bearbeitung einer kompletten Ladung zu verkürzen. Mit dem Pipelining wird also die Zeit zum Waschen einer Waschmaschinenladung nicht verkürzt, aber wenn wir viele Wäscheladungen zu waschen haben, wird die Gesamtzeit aufgrund des besseren Durchsatzes verkürzt.

Wenn alle vier Stufen etwa gleich viel Zeit beanspruchen und genügend Arbeit vorhanden ist, entspricht die Beschleunigung aufgrund des Pipelining der Anzahl der Pipelinstufen, in diesem Beispiel also vier: Waschen, Trocknen, Zusammenlegen und Wegräumen. Somit ist Wäschewaschen mit Fließbandprinzip potenziell viermal

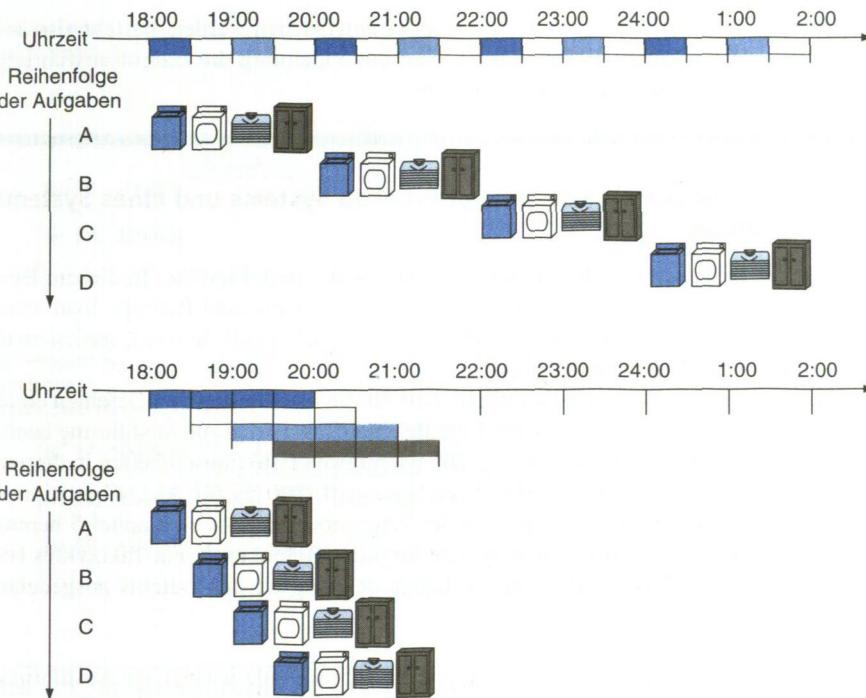


Abb. 6.1 Der Waschsalon mit und ohne Pipelining. Ann, Brian, Cathy und Don haben jeweils schmutzige Wäsche zu waschen, zu trocknen, zusammenzulegen und aufzuräumen. Die Waschmaschine, der Wäschetrockner, der „Zusammenleger“ und „Wegräumer“ brauchen jeweils 30 Minuten für ihre Aufgabe. Wenn die Aufgaben sequenziell ausgeführt werden, werden für vier Ladungen Wäsche acht Stunden benötigt, bei einem Waschsalon mit Fließbandprinzip (Pipelining) dagegen nur 3,5 Stunden. Dargestellt ist jeweils die Pipelinestufe unterschiedlicher Ladungen in Abhängigkeit von der Zeit durch mehrfache Darstellung der vier Ressourcen entlang dieser zweidimensionalen Zeitachse. Natürlich stehen die Ressourcen nur jeweils einmal zur Verfügung.

schneller als Wäschewaschen ohne Fließbandprinzip: 20 Wäscheladungen benötigen etwa fünfmal so viel Zeit wie eine Ladung Wäsche, während 20 Ladungen Wäsche ohne Fließbandprinzip 20-mal so viel Zeit beanspruchen wie eine Ladung. In Abbildung 6.1 ist die Verarbeitung nur 2,3-mal schneller, weil wir nur vier Ladungen haben. Am Anfang und am Ende des Arbeitsvorgangs in der Version mit Pipelining in Abbildung 6.1 ist die Pipeline nicht voll. Dieses „Anlaufen“ und „Auslaufen“ beeinträchtigt die Leistung, wenn die Anzahl der Aufgaben im Vergleich zur Anzahl der Pipelinestufen nicht groß ist. Wenn die Anzahl der Ladungen wesentlich größer als vier ist, sind die Stufen fast die ganze Zeit voll und der Durchsatz kommt sehr nahe an vier heran.

Dasselbe Prinzip gilt für Prozessoren, bei denen Befehle mittels Pipelining ausgeführt werden. Für die Ausführung von MIPS-Befehlen sind herkömmlicherweise fünf Schritte (Befehlsphasen) erforderlich:

1. Holen des Befehls aus dem Speicher.
2. Lesen der Register und gleichzeitiges Entschlüsseln des Befehls. Das Format der MIPS-Befehle ermöglicht das gleichzeitige Lesen und Entschlüsseln.
3. Ausführen der Operation oder Berechnen einer Adresse.
4. Zugreifen auf einen Operanden im Datenspeicher.
5. Schreiben des Ergebnisses in ein Register.

Die MIPS-Pipeline, die wir in diesem Kapitel untersuchen werden, besteht also aus fünf Stufen. Das folgende Beispiel zeigt, dass das Pipelining die Befehlausführung ebenso beschleunigt wie das Wäschewaschen.

BEISPIEL

Vergleich der Leistung eines sequenziellen Systems und eines Systems mit Pipelining

Damit die Diskussion konkreter wird, entwerfen wir eine Pipeline. In diesem Beispiel und im Rest dieses Kapitels beschränken wir uns auf acht Befehle: load word (`lw`), store word (`sw`), add (`add`), subtract (`sub`), and (`and`), or (`or`), set less than (`slt`) und branch on equal (`beq`).

Vergleichen Sie die durchschnittliche Zeit für die Ausführung von Befehlen eines sequenziellen Systems, bei dem alle Befehle einen Taktzyklus zur Ausführung benötigen, mit einer Pipeline-Ausführung. Die wichtigsten Funktionseinheiten in diesem Beispiel benötigen 200 ps für den Speicherzugriff, 200 ps für ALU-Operationen und 100 ps zum Lesen und Schreiben des Registersatzes. Wie in Kapitel 5 bereits erwähnt, wird beim sequenziellen System für jeden Befehl exakt ein Taktzyklus benötigt, so dass der Taktzyklus auf die Länge des langsamsten Befehls ausgedehnt werden muss.

ANTWORT

In Tabelle 6.1 ist angegeben, wie viel Zeit die acht Befehle jeweils zur Ausführung benötigen. Beim sequenziellen System muss der Takt so lang sein, wie der langsamste Befehl (in Tabelle 6.1 ist das der Befehl `lw`), so dass die für jeden Befehl beanspruchte Zeit 800 ps beträgt. Ähnlich wie in Abbildung 6.1 wird in Abbildung 6.2 die Ausführung von drei Load-word-Befehlen ohne und mit Pipelining verglichen. Der Zeitabstand zwischen dem ersten und vierten Befehl im Entwurf ohne Pipelining beträgt 3×800 ps oder 2400 ps.

Alle Pipelinestufen beanspruchen einen einzigen Taktzyklus. Daher muss der Taktzyklus so lang sein wie die langsamste Stufe. So wie beim sequenziellen System der längste Taktzyklus mit 800 ps verwendet werden muss, auch wenn einige Befehle lediglich 500 ps benötigen, so muss auch bei der Ausführung mit Pipelining der langsamste Taktzyklus mit 200 ps verwendet werden, auch wenn einige Stufen nur 100 ps benötigen. Das Pipelining ermöglicht dennoch eine vierfache Leistungssteigerung: Der Zeitabstand zwischen dem ersten und vierten Befehl beträgt 3×200 ps oder 600 ps.

Wir können die weiter oben formulierten Beobachtungen der Beschleunigung mittels Pipelining in eine Formel fassen. Wenn die Stufen alle genau gleich lange Ausfüh-

Tab. 6.1 Gesamtzeit für jeden Befehl, ermittelt aus der Summe der Ausführungszeiten für die einzelnen Komponenten. Bei dieser Berechnung wird vorausgesetzt, dass es bei den Multiplexern, bei der Steuereinheit, bei den Befehlszählerzugriffen und bei der Vorzeichenerweiterungseinheit zu keiner Verzögerung kommt.

Befehlsklasse	Befehl holen	Register lesen	ALU-Operation	Datenzugriff	In Register schreiben	Gesamtzeit
load word (<code>lw</code>)	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps
store word (<code>sw</code>)	200 ps	100 ps	200 ps	200 ps		700 ps
R-format (<code>add</code> , <code>sub</code> , <code>and</code> , <code>or</code> , <code>slt</code>)	200 ps	100 ps	200 ps		100 ps	600 ps
branch (<code>beq</code>)	200 ps	100 ps	200 ps			500 ps

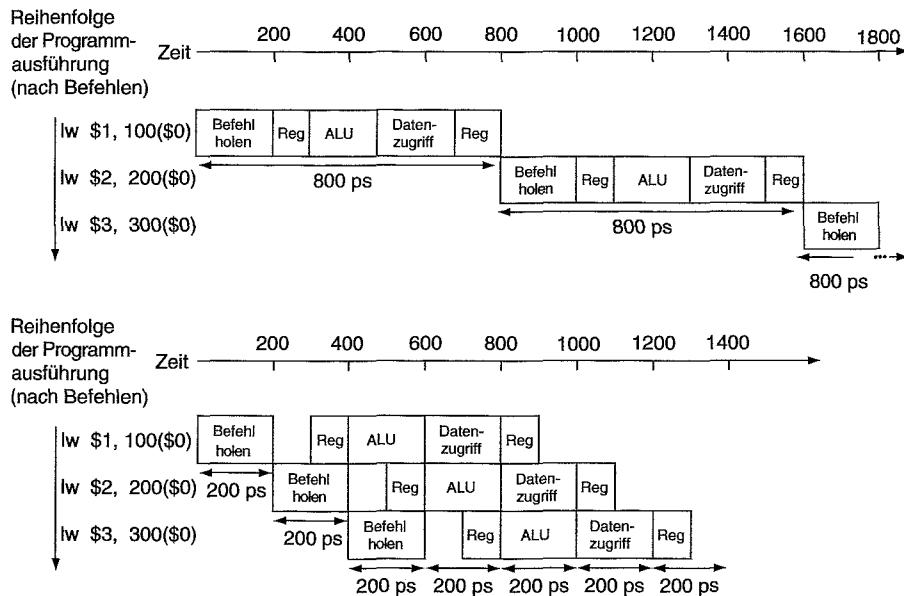


Abb. 6.2 Befehlsausführung bei einem sequenziellen System ohne Pipelining (oben) und mit Pipelining (unten) im Vergleich. Beide Male werden dieselben Hardwarekomponenten verwendet, deren Ausführungszeiten in Tabelle 6.1 aufgeführt sind. Bei diesem Beispiel ist eine Verkürzung der Durchschnittszeit für die Ausführung von Befehlen von 800 ps auf 200 ps zu beobachten. Vergleichen Sie diese Abbildung mit Abbildung 6.1. Beim Waschen sind wir davon ausgegangen, dass alle Stufen gleich lang sind. Wenn der Trockner die langsamste Einheit wäre, würde die Trocknerstufe die Stufenzzeit bestimmen. Die Ausführungszeiten der Pipelinestufen beim Computer werden durch die langsamste Ressource bestimmt, also entweder durch die ALU-Operation oder durch den Speicherzugriff. Wir gehen davon aus, dass das Schreiben des Registersatzes in der ersten Hälfte des Taktzyklus erfolgt und das Lesen aus dem Registersatz in der zweiten Hälfte. Von dieser Voraussetzung gehen wir während des gesamten Kapitels aus.

rungzeiten haben, beträgt die Zeit zwischen Befehlen (unter idealen Bedingungen) bei einem Prozessor mit Pipelining

$$\text{Zeit zwischen Befehlen}_{\text{mit Pipelining}} = \frac{\text{Zeit zwischen Befehlen}_{\text{ohne Pipelining}}}{\text{Anzahl der Pipelinestufen}}$$

Unter idealen Bedingungen und mit einer großen Anzahl von Befehlen entspricht die Beschleunigung aufgrund des Pipelining etwa der Anzahl der Pipelinestufen. Somit ist eine fünfstufige Pipeline nahezu fünfmal schneller als das System mit sequenzieller Befehlsausführung.

Gemäß der Formel müsste eine fünfstufige Pipeline nahezu fünfmal leistungsfähiger sein als die Ausführung im System mit sequenziellem Zyklus mit 800 ps Takt und also einen 160 ps Takt aufweisen. Anhand des Beispiels wird jedoch deutlich, dass die Stufen zeitlich nicht ausbalanciert sind. Zudem beinhaltet das Pipelining einen gewissen Mehraufwand, dessen Ursache in Kürze verständlich wird. Somit überschreitet die Befehlsausführungszeit im Prozessor mit Pipelining die minimal mögliche Zeit und die Beschleunigung ist geringer als die Anzahl der Pipelinestufen.

Hinzu kommt, dass sich nicht einmal unsere Annahme einer vierfachen Beschleunigung für unser Beispiel in der Gesamtausführungszeit für die drei Befehle bestätigt: 1400 ps zu 2400 ps. Natürlich lässt sich dies auf die geringe Anzahl an ausgeführten Befehlen zurückführen. Was würde geschehen, wenn wir die Anzahl der Befehle

erhöhten? Wir könnten die vorherige Zahl auf 1 000 003 Befehle erhöhen. Wir würden die Anzahl der Befehle im Beispiel mit Pipelining um 1 000 000 erhöhen, wobei jeder Befehl 200 ps zu der Gesamtausführungszeit betragen würde. Die Gesamtausführungszeit würde $1\ 000\ 000 \times 200\ \text{ps} + 1400\ \text{ps}$ oder 200 001 400 ps betragen. Im Beispiel ohne Pipeline würden wir 1 000 000 Befehle hinzufügen, wobei jeder die Gesamtausführungszeit um 800 ps erhöht, so dass die Gesamtausführungszeit $1\ 000\ 000 \times 800\ \text{ps} + 2400\ \text{ps}$ oder 800 002 400 ps beträgt. Unter diesen idealen Bedingungen entspricht das Verhältnis der Gesamtausführungszeiten für reale Programme auf Prozessoren ohne Pipelining zu Prozessoren mit Pipelining nahezu dem Verhältnis der Zeiten zwischen Befehlen:

$$\frac{800\ 002\ 400\ \text{ps}}{200\ 001\ 400\ \text{ps}} \approx 4,00 = \frac{800\ \text{ps}}{200\ \text{ps}}$$

Das Pipelining verbessert die Leistung durch einen *erhöhten Befehlsdurchsatz*. Die Ausführungszeit der einzelnen Befehle wird dagegen nicht reduziert. Der Befehlsdurchsatz ist jedoch die wichtigste Metrik, da echte Programme Milliarden von Befehlen ausführen.

Entwurf von Befehlssätzen für das Pipelining

Bereits mit dieser einfachen Beschreibung des Pipelining können wir Erkenntnisse zum Entwurf des MIPS-Befehlssatzes gewinnen, der für die Ausführung mit Pipelining konzipiert wurde.

Erstens sind alle MIPS-Befehlsformate gleich, d.h. die Befehle sind gleich lang. Aufgrund dieser Tatsache ist es wesentlich einfacher, Befehle in der ersten Pipelinestufe zu holen und in der zweiten Pipelinestufe zu entschlüsseln. Bei einem Befehlssatz wie dem IA-32-Befehlssatz, bei dem Befehle eine unterschiedliche Länge zwischen einem und 17 Byte aufweisen, ist ein Pipelinesystem erheblich schwieriger zu realisieren. Wie wir in Kapitel 5 sehen konnten, übersetzen alle neueren Implementierungen der IA-32-Architektur IA-32-Befehle in einfache Befehle (Herstellerbezeichnung „micro-operations“, jedoch nicht zu verwechseln mit Mikrooperationen!), die den MIPS-Befehlen ähnlich sind. Wie wir in Abschnitt 6.10 sehen werden, verarbeitet der Pentium 4 anstelle der systemeigenen IA-32-Befehle diese „micro-operations“ mittels Pipeline!

Zweitens gibt es bei MIPS nur einige wenige Befehlsformate. Bei diesen befinden sich die Felder für die Angabe der Quellregisteradressen jeweils an derselben Stelle. Diese Symmetrie bedeutet, dass die zweite Pipelinestufe damit beginnen kann, den Registersatz zu lesen, während die Hardware zeitgleich bestimmt, welcher Befehl geholt wurde. Dekodierstufe und Registerlesestufe des Maschinenbefehlszyklus fallen also bei der MIPS-Pipeline zu einer gemeinsamen, der zweiten Stufe zusammen. Wenn MIPS-Befehlsformate nicht symmetrisch aufgebaut wären, müsste Stufe 2 geteilt werden, was eine Pipeline mit sechs Stufen zur Folge hätte. Wir werden später sehen, welche Nachteile längere Pipelines aufweisen.

Drittens kommen Speicheroperanden bei MIPS nur in Lade- oder Speicherbefehlen vor. Diese Beschränkung bedeutet, dass wir die Ausführungsstufe zum Berechnen der Speicheradresse verwenden können und den eigentlichen Speicherzugriff erst in der Folgestufe ausführen. Wenn wir mit Speicheroperanden wie in der IA-32-Architektur arbeiten könnten, würden sich die Stufen 3 und 4 der MIPS-Pipeline auf je eine Adressstufe, Speicherstufe und eine Ausführungsstufe erweitern.

Viertens müssen die Operanden, wie in Kapitel 2 beschrieben, im Speicher ausgerichtet sein. Somit benötigen wir keinen einzigen Datentransfer-Befehl, der zweimal auf den Datenspeicher zugreifen muss. Die angeforderten Daten können in einer einzigen Pipelinestufe zwischen Prozessor und Speicher übertragen werden.

Pipelinehemmnisse

Beim Pipelining gibt es Situationen, in denen der nächste Befehl nicht im nachfolgenden Taktzyklus ausgeführt werden kann. Diese Ereignisse werden als *Hemmnisse* oder auch *Konflikte* bezeichnet, von denen es drei verschiedene Typen gibt.

Strukturkonflikt

Der erste Konflikt wird als **Strukturkonflikt (structural hazard)** bezeichnet. Hierbei kann die Hardware die Befehlskombination, die in einem Taktzyklus ausgeführt werden soll, nicht unterstützen. Im Waschsalon würde ein Strukturkonflikt auftreten, wenn wir anstelle einer Waschmaschine und eines Trockners ein Waschmaschinen-Trockner-Kombigerät verwendeten oder wenn unser Mitbewohner mit etwas Anderem beschäftigt wäre und die Wäsche nicht in den Schrank räumen würde. Unsere sorgfältig ausgedachten Pipelinestrukturen wären damit nicht ausführbar.

Wie wir bereits weiter oben erwähnt haben, wurde der MIPS-Befehlssatz für das Pipelining konzipiert, so dass es Entwicklern leichter gemacht wurde, beim Entwerfen einer Pipeline Strukturkonflikte zu vermeiden. Nehmen wir jedoch an, wir hätten anstelle zweier Speicher nur einen. Wenn die Pipeline in Abbildung 6.2 einen vierten Befehl enthielte, würde im selben Taktzyklus, in dem der erste Befehl auf Daten im Speicher zugreift, der vierte Befehl gleichzeitig einen Befehl aus demselben Speicher holen. Mit nur einem Speicher hätte unsere Pipeline einen Strukturkonflikt.

Strukturkonflikt (structural hazard) Ein Ereignis, bei dem ein Befehl nicht im vorgesehenen Taktzyklus ausgeführt werden kann, da die Hardware die Befehlskombination nicht unterstützt, die zum Ausführen im angegebenen Taktzyklus festgelegt wurde.

Datenkonflikte

Datenkonflikte (data hazard) treten auf, wenn die Pipeline anhalten muss, da ein Schritt auf den Abschluss eines anderen wartet. Nehmen wir an, Sie finden beim Zusammenlegen der Wäsche eine Socke, zu der die zweite fehlt. Eine mögliche Strategie besteht darin, in Ihr Zimmer hinunterzulaufen und in der Kommode nach der passenden zweiten Socke zu suchen. Während Sie suchen, müssen getrocknete Ladungen, die zusammengelegt werden könnten, und gewaschene Ladungen, die getrocknet werden könnten, liegen bleiben.

Bei einer Pipeline im Rechner treten Datenkonflikte aufgrund der Abhängigkeit eines Befehls von einem zu einem früheren Zeitpunkt begonnenen Befehl auf, der sich noch in der Pipeline befindet. (Eine Abhängigkeit, die es beim Wäschewaschen nicht wirklich gibt.) Nehmen wir beispielsweise an, wir hätten einen add-Befehl direkt gefolgt von einem subtract-Befehl, der die Summe (\$s0) verwendet:

```
add    $s0, $t0, $t1
sub    $t2, $s0, $t3
```

Ohne Eingreifen kann ein Datenkonflikt die Pipelineverarbeitung erheblich verzögern. Der add-Befehl schreibt sein Ergebnis erst in der fünften Stufe, was bedeutet, dass wir drei so genannte *Bubbles* (Pipelineleerlauf, Leeroperationen oder Wartetakte) in die Pipeline einfügen müssen.

Wir könnten Compiler verwenden, die alle derartigen Konflikte eliminieren. Das Ergebnis wäre jedoch nicht befriedigend. Diese Abhängigkeiten kommen einfach zu häufig vor, und die Verzögerung ist zu lang, als dass wir erwarten können, dass der Compiler dieses Problem löst.

Die wichtigste Gegenmaßnahme beruht auf der Beobachtung, dass wir mit dem Beheben des Datenkonflikts nicht warten müssen, bis der Befehl ausgeführt ist. Bei der obigen Codesequenz können wir die Summe aus der Addition als Eingangswert für die Subtraktion bereitstellen, sobald die ALU die Summe berechnet hat. Das Verwenden zusätzlicher Hardware zum frühzeitigen Abrufen des fehlenden Elements aus den internen Ressourcen wird als **Forwarding** oder **Bypassing** (Daten-Bypasstechnik) bezeichnet.

Datenkonflikt (data hazard) Auch als Pipelinehemmnis durch Datenabhängigkeit bezeichnet. Ein Ereignis, bei dem ein Befehl nicht im vorgesehenen Taktzyklus ausgeführt werden kann, weil Daten zum Ausführen des Befehls noch nicht verfügbar sind.

Forwarding Auch als **Bypassing** bezeichnet. Eine Methode zum Lösen eines Datenkonflikts, bei der das fehlende Datenelement aus internen Pufferspeichern abgerufen und nicht darauf gewartet wird, bis dieses aus den für den Programmierer sichtbaren Registern oder aus dem Speicher kommt.

BEISPIEL**ANTWORT****Forwarding mit zwei Befehlen**

Zeigen Sie für die beiden obigen Beispielbefehle, welche Pipelinestufen durch Forwarding miteinander verbunden werden müssen. Verwenden Sie die Zeichnung in Abbildung 6.3, um den Datenpfad während der fünf Pipelinestufen darzustellen. Richten Sie eine Kopie des Datenpfads für jeden Befehl ähnlich der Waschsalonpipeline in Abbildung 6.1 aus.

In Abbildung 6.4 ist die Verbindung zum Weiterleiten des Werts in \$s0 nach der Ausführungsstufe des add-Befehls als Eingangswert an die Ausführungsstufe des sub-Befehls dargestellt.

Bei dieser grafischen Darstellung von Ereignissen sind Forwarding-Pfade nur zulässig, wenn die Zielstufe zu einem späteren Zeitpunkt ausgeführt wird als die Quellstufe. Es kann beispielsweise keinen zulässigen Forwarding-Pfad vom Ausgang der Speicherzugriffsstufe im ersten Befehl zum Eingang der Ausführungsstufe des nachfolgenden Befehls geben, da dies einen Zeitsprung zurück bedeuteten würde.

Forwarding funktioniert sehr gut und wird in Abschnitt 6.4 ausführlich beschrieben. Damit lassen sich jedoch nicht alle Pipelineverzögerungen verhindern. Nehmen wir beispielsweise an, mit dem ersten Befehl werde kein add-Befehl durchgeführt, sondern Register \$s0 aus dem Speicher geladen. Wie wir anhand von Abbildung 6.4 erkennen können, werden die gewünschten Daten erst *nach* der vierten Stufe des ersten Befehls in der Abhängigkeit bereitgestellt, also zu spät für den *Eingang* der dritten Stufe des sub-Befehls. Also kann es auch mit Forwarding vorkommen, dass wir, wie in Abbildung 6.5 zu sehen, eine Stufe aufgrund eines **Load-use-Konflikts (load-use data hazard)** anhalten müssen. Diese Abbildung zeigt ein wichtiges Pipelinekonzept, das formal als **Pipelineleerlauf (pipeline stall)**, weniger formal jedoch häufig als **Bubble** bezeichnet wird. Wir werden Pipelineleerläufe noch an anderen Stellen in der Pipeline kennenlernen. In Abschnitt 6.5 wird beschrieben, wie wir schwierigen Fällen wie diesem entweder mit Hardwareerkennung und Hardwareverzögerungen oder mit Software begegnen können, die die Ladeverzögerung wie eine Sprungverzögerung behandelt.

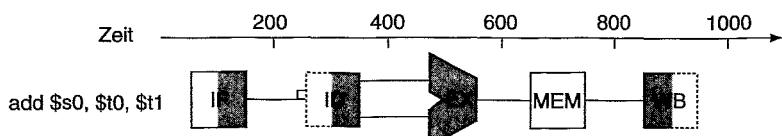


Abb. 6.3 Grafische Darstellung der Befehlspipeline im Sinne der Waschsalonpipeline in Abbildung 6.1. Hier verwenden wir für die Darstellung der Hardwareressourcen Symbole mit Abkürzungen der Pipelinestufen, wie wir sie im ganzen Kapitel beschreiben. Die Symbole für die fünf Stufen sind: **IF** für die Befehlsholstufe (Instruction Fetch), wobei das Kästchen den Befehlsspeicher darstellt. **ID** für die Befehlsentschlüsselungs-/Registerlesestufe (Instruction Decode), wobei das Kästchen das gelesene Register darstellt. **EX** für die Ausführungsstufe (Execution), wobei das Symbol die ALU darstellt. **MEM** für die Speicherzugriffsstufe (Memory Access), wobei das Kästchen den Datenspeicher darstellt. Und **WB** für die Rückschreibstufe (Write Back), wobei das Symbol das geschriebene Register darstellt. Durch die Grauschattierung wird angegeben, dass das Element vom Befehl verwendet wird. **MEM** hat daher einen weißen Hintergrund, weil **add** nicht auf den Datenspeicher zugreift. Eine Grauschattierung der rechten Hälfte des Registersatzes oder des Speichers bedeutet, dass das Element in dieser Stufe gelesen wird, und eine Grauschattierung der linken Hälfte bedeutet, dass das Element in dieser Stufe geschrieben wird. Daher ist die rechte Hälfte des **ID**-Symbols in der zweiten Stufe schattiert, weil der Registersatz gelesen wird, und die linke Hälfte des **WB**-Symbols ist in der fünften Stufe schattiert, weil in den Registersatz geschrieben wird.

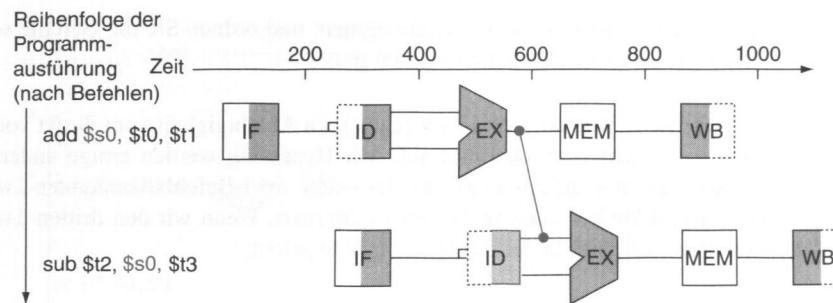


Abb. 6.4 **Grafische Darstellung des Forwarding.** Die Verbindung stellt den Forwarding-Pfad vom Ausgang der EX-Stufe des add-Befehls zum Eingang der EX-Stufe des sub-Befehls dar, wobei der in der zweiten Stufe des sub-Befehls gelesene Wert aus Register \$s0 ersetzt wird.

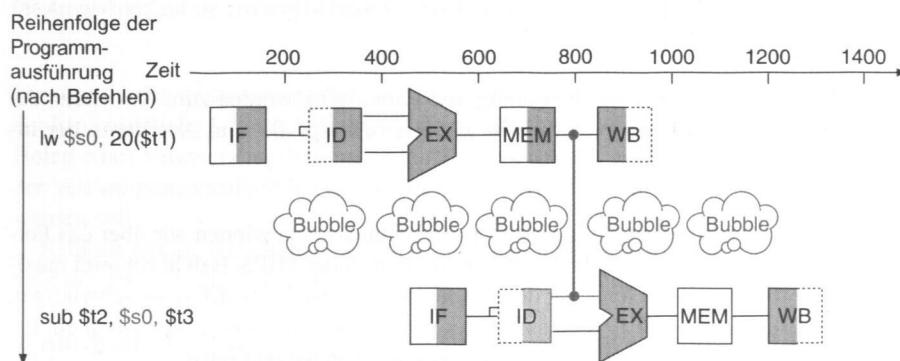


Abb. 6.5 **Wir müssen selbst die Pipeline mit Forwarding anhalten, wenn ein R-Befehl nach einem Ladebefehl versucht, die geladenen Daten zu verwenden.** Ohne Wartetakt wäre der Pfad vom Ausgang der Speicherzugriffsstufe zum Eingang der Ausführungsstufe ein zeitlicher Rücksprung, was nicht möglich ist. Diese Abbildung ist eigentlich eine Vereinfachung, da wir erst nach dem Holen und Entschlüsseln des Subtraktionsbefehls wissen, ob die Pipeline angehalten werden muss. In Abschnitt 6.5 wird erläutert, was im Falle eines Konflikts im Einzelnen geschieht.

Umordnen von Code zum Vermeiden von Pipelineleerläufen

Gehen wir von folgendem Codesegment in C aus:

```
A = B + E;
C = B + F;
```

Der MIPS-Code für dieses Segment lautet unter der Voraussetzung, dass sich alle Variablen im Speicher befinden und als Offsets von Register \$t0 adressierbar sind, wie folgt:

```
lw $t1, 0($t0)
lw $t2, 4($t0)
add $t3, $t1,$t2
sw $t3, 12($t0)
lw $t4, 8($t0)
add $t5, $t1,$t4
sw $t5, 16($t0)
```

BEISPIEL

Suchen Sie die Konflikte in diesem Codesegment und ordnen Sie die Befehle so um, dass die Pipeline nicht angehalten werden muss.

ANTWORT

Beide add-Befehle weisen aufgrund ihrer jeweiligen Abhängigkeit vom direkt vorausgehenden lw-Befehl einen Konflikt auf. Mit Bypassing werden einige andere potenzielle Konflikte wie die Anhängigkeit des ersten add-Befehls vom ersten lw-Befehl sowie Konflikte bei Speicherbefehlen eliminiert. Wenn wir den dritten lw-Befehl nach oben verschieben, sind beide Konflikte gelöst:

```
lw $t1, 0($t0)
lw $t2, 4($t1)
lw $t4, 8($01)
add $t3, $t1,$t2
sw $t3, 12($t0)
add $t5, $t1,$t4
sw $t5, 16($t0)
```

Bei einem Prozessor mit Pipelining und Forwarding werden zum Ausführen der umgeordneten Sequenz zwei Zyklen weniger benötigt als zum Ausführen der ursprünglichen Sequenz.

Neben den vier auf Seite 302 genannten Erkenntnissen gewinnen wir über das Forwarding weitere Einblicke in die MIPS-Architektur. Jeder MIPS-Befehl schreibt maximal ein Ergebnis und das kurz vor dem Ende der Pipeline. Forwarding ist schwieriger, wenn pro Befehl mehrere Ergebnisse weitergeleitet werden müssen oder wenn ein Ergebnis frühzeitig in der Befehlausführung geschrieben werden muss.



Vertiefung: Die Bezeichnung „Forwarding“ (Weiterleiten) bezieht sich auf die Idee, dass das Ergebnis von einem früheren Befehl an einen späteren Befehl weitergeleitet wird. „Bypassing“ (Umleiten) dagegen verweist darauf, dass das Ergebnis um den Registersatz herum direkt an die gewünschte Einheit geleitet wird.

Steuerkonflikte

Steuerkonflikt (*control hazard*)

Auch Verzweigungskonflikt (*branch hazard*) genannt. Ein Ereignis, bei dem der gewünschte Befehl nicht im gewünschten Taktzyklus ausgeführt werden kann, weil der Befehl, der geholt wurde, nicht der ist, der benötigt wird. Das bedeutet, dass die Abfolge von Befehlsadressen anders als von der Pipeline erwartet ist.

Der dritte Konflikttyp wird als **Steuerkonflikt (*control hazard*)** bezeichnet. Er kann entstehen, wenn aufgrund der Ergebnisse eines Befehls eine Entscheidung getroffen werden muss, während andere Befehle ausgeführt werden.

Nehmen wir an, unser Waschsalonteam wird mit der schönen Aufgabe betraut, die Trikots einer Fußballmannschaft zu waschen. Wir müssen, je nachdem wie stark verschmutzt die Wäsche ist, entscheiden, ob die gewählte Waschmittelmenge und Wassertemperatur ausreicht, um die Trikots sauber zu bekommen, aber nicht so stark ist, dass sich die Trikots vorzeitig abtragen. In unserer Waschsalonpipeline müssen wir bis zur zweiten Stufe warten, um prüfen zu können, ob wir Waschmittelmenge oder Wassertemperatur ändern müssen. Was tun?

Eine der beiden Lösungen zum Auflösen von Steuerkonflikten im Waschsalon und das entsprechende Pendant beim Computer wird im Folgenden beschrieben.

Leerlauf: Arbeiten Sie einfach eine Stufe nach der anderen ab, bis die erste Ladung trocken ist, und wiederholen Sie diesen Vorgang, bis Sie die richtige Waschmittelmenge und Wassertemperatur herausgefunden haben. Diese konservative Arbeitsweise funktioniert zweifelsohne, ist jedoch langsam.

Die entsprechende Entscheidungsaufgabe bei einem Computer ist der Verzweigungsbefehl. Wir müssen direkt nach der Verzweigung beim nächsten Taktzyklus mit dem Holen des Befehls beginnen. Aber die Pipeline kann den nächsten Befehl nicht

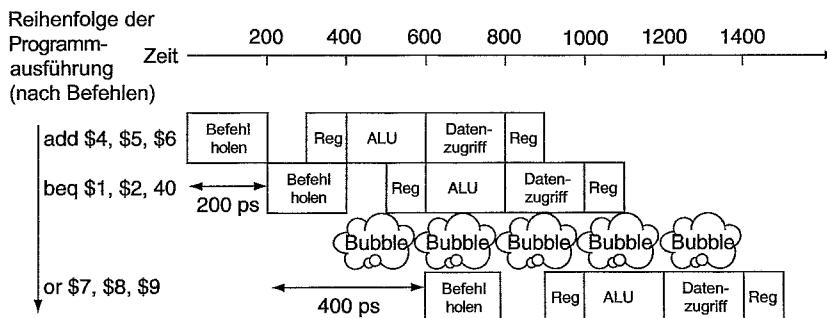


Abb. 6.6 Pipeline, die bei jedem bedingten Sprung angehalten wird, um Steuerkonflikte zu lösen. Nach der Verzweigung kommt es zu einem einstufigen Pipelineleerlauf oder Bubble. In der Realität ist das Generieren eines Leerlaufs etwas schwieriger, wie wir in Abschnitt 6.6 sehen werden. Die Auswirkung auf die Leistung ist jedoch dieselbe als würde ein Leerlaufakt eingefügt.

kennen, da der Verzweigungsbefehl *gerade erst* aus dem Speicher geholt wurde! Eine mögliche Lösung besteht wie beim Waschsalon darin, die Pipeline nach dem Holen eines Verzweigungsbefehls anzuhalten, zu warten, bis die Pipeline das Ergebnis der Verzweigung ermittelt hat und weiß, von welcher Befehlsadresse der Befehl geholt werden soll.

Nehmen wir an, wir verfügen über genügend zusätzliche Hardware, so dass wir während der zweiten Pipelinestufe Register testen, die Sprungadresse berechnen und den Befehlszähler aktualisieren können. (Ausführlichere Informationen hierzu finden Sie in Abschnitt 6.6.) Auch mit dieser zusätzlichen Hardware würde die Pipeline mit bedingten Sprüngen wie in Abbildung 6.6 aussehen. Der *or*-Befehl, der ausgeführt wird, wenn der Sprung fehlschlägt, wird für die Dauer eines zusätzlichen 200-ps-Taktzyklus angehalten, bevor er ausgeführt wird.

Leistung des Sprungbefehls mit Leerlauf

Schätzen Sie die Auswirkungen des Leerlaufs bei Sprüngen auf den CPI-Wert (Clock Cycles Per Instruction) ein. Gehen Sie davon aus, dass alle anderen Befehle einen CPI-Wert von 1 aufweisen.

In Tabelle 3.14, Seite 194 in Kapitel 3 wird deutlich, dass Verzweigungen 13% der Befehle ausmachen, die bei SPECint2000-Benchmarks ausgeführt werden. Da alle anderen ausgeführten Befehle einen CPI-Wert von 1 aufweisen und Verzweigungen für den Leerlauf einen zusätzlichen Taktzyklus benötigen, ergibt sich ein CPI-Wert von 1,13 und somit eine Verzögerung um 1,13 im Vergleich zum Idealfall. Dies bezieht sich jedoch nur auf Verzweigungen. Unbedingte Sprünge können ebenfalls einen Leerlauf verursachen.

Wenn wir die Verzweigung in der zweiten Stufe nicht auflösen können, wie das bei längeren Pipelines häufig der Fall ist, kommt es zu noch größeren Verzögerungen, wenn wir die Pipeline bei Verzweigungen anhalten. Der Nachteil dieser Lösung ist für die meisten Computer zu groß und Grund für eine zweite Lösung des Steuerkonflikts:

Vorhersage: Wenn Sie sich ziemlich sicher sind, dass Sie die für das Wäschens der Trikots richtige Waschmittelmenge und Wassertemperatur kennen, dann sagen Sie einfach vorher, dass es funktionieren wird, und waschen die zweite Ladung, während Sie darauf warten, bis die erste trocken

BEISPIEL

ANTWORT

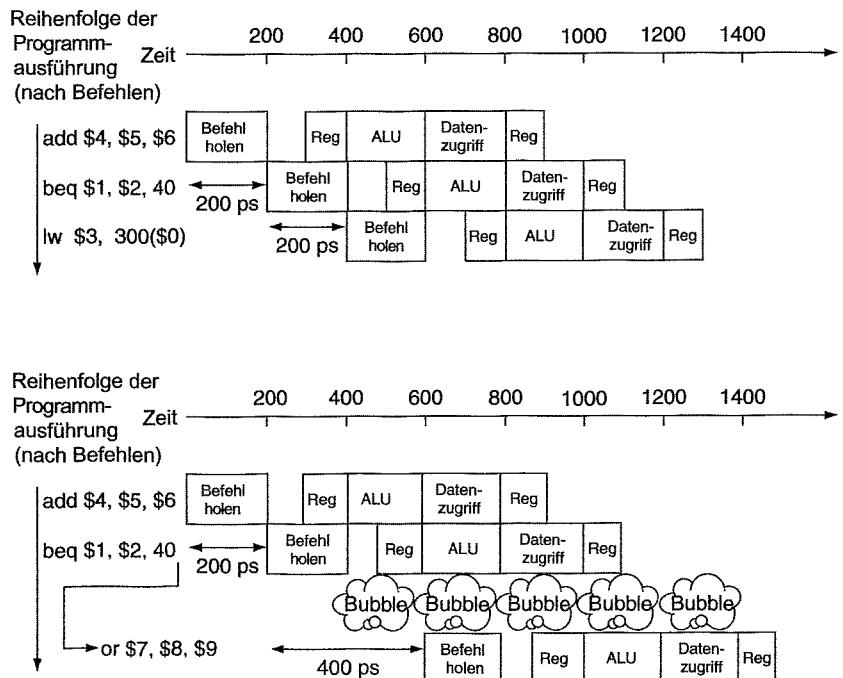


Abb. 6.7 Vorhersagen, dass Verzweigungen nicht ausgeführt werden, zur Auflösung von Steuerkonflikten. Im oberen Teil der Abbildung ist die Pipeline dargestellt, wenn der Sprung nicht ausgeführt wird. Im unteren Teil der Abbildung ist die Pipeline dargestellt, wenn der Sprung ausgeführt wird. Wie wir bereits in Abbildung 6.6 feststellen konnten, wird durch Einfügen eines Leertaktes auf diese Art und Weise die Darstellung dessen vereinfacht, was zumindest während des ersten Taktzyklus direkt nach dem Sprung wirklich geschieht. In Abschnitt 6.6 finden Sie eine ausführlichere Beschreibung hierzu.

ist. Wenn Sie Recht haben, wird mit dieser Option die Pipeline nicht verzögert. Wenn Sie nicht Recht haben, müssen Sie die Ladung, die gewaschen wurde, während Sie die Entscheidung trafen, noch einmal waschen.

Computer verwenden im Umgang mit Sprüngen tatsächlich **Vorhersagen**. Ein einfacher Ansatz besteht darin, immer vorherzusagen, dass Sprünge **nicht ausgeführt** (*untaken branch*) werden. Wenn Sie Recht haben, arbeitet die Pipeline mit voller Geschwindigkeit. Die Pipeline wird nur angehalten, wenn Sprünge ausgeführt werden. In Abbildung 6.7 ist ein Beispiel hierfür dargestellt.

Bei einer anspruchsvoller Version der **Sprungvorhersage (branch prediction)** wird angenommen, dass gewisse Sprünge ausgeführt werden, andere nicht. Bei unserem Vergleich wird für die dunklen oder Heimtrikots eine bestimmte Waschmittelmenge und Wassertemperatur verwendet, während für die hellen oder Straßentrikots eine andere Waschmittelmenge und Wassertemperatur verwendet wird. Ein Beispiel für den Bereich der Rechner sind Sprünge am Ende von Schleifen, die an den Anfang der Schleife verzweigen. Die Wahrscheinlichkeit, dass der Sprung ausgeführt wird ist hoch (solange die Schleife ausgeführt wird). Da es sich bei Schleifen um Rücksprünge handelt, könnte die Sprungvorhersage festlegen, dass Sprünge auf zurückliegende Adressen immer als auszuführend anzunehmen sind.

Starre Ansätze wie diese beruhen auf der Annahme stereotypen Verhaltens und berücksichtigen die Individualität einzelner Sprungbefehle nicht. In krassem Gegensatz dazu, ziehen **dynamische Hardware-Prädiktoren** ihre Schlüsse aus dem Verhalten jedes einzelnen Sprungs und können die Vorhersage für eine Verzweigung während der

Nicht ausgeführter Sprung (untaken branch) Ein Sprung, dessen Bedingung fehlschlägt und der mit dem unmittelbar nachfolgenden Befehl fortfährt. Ein ausgeführter Sprung ist ein Sprung, der zum Sprungziel verzweigt.

Sprungvorhersage (branch prediction) Eine Methode zum Auflösen eines Steuerkonflikts, bei der für den Sprung ein bestimmtes Ergebnis angenommen wird und bei der unter dieser Voraussetzung fortgefahrt wird, statt auf die Bestätigung des tatsächlichen Ergebnisses zu warten.

Ausführung eines Programms ändern. In unserem Beispiel würde eine Person bei der dynamischen Vorhersage prüfen, wie stark die Wäsche verschmutzt ist, die Waschmittelmenge und Waschtemperatur entsprechend einschätzen und die nächste Einschätzung vom Erfolg der letzten Einschätzung abhängig machen. Ein beliebter Ansatz bei der dynamischen Vorhersage von Verzweigungen besteht darin, ausgeführte und nicht ausgeführte Sprünge zu protokollieren, und dann anhand der letzten Sprünge die nächsten vorherzusagen. Wie wir noch sehen werden, werden in großem Umfang Art und Anzahl von Sprüngen protokolliert mit dem Ergebnis, dass dynamische Sprungprädiktoren Sprünge mit einer Genauigkeit von über 90% vorhersagen können (siehe Abschnitt 6.6). Wenn die Einschätzung falsch ist, muss die Pipelinesteuerung sicherstellen, dass die Befehle nach dem falsch eingeschätzten Sprung keine Auswirkung haben, und sie muss die Pipeline von der richtigen Sprungadresse aus neu starten. Bei unserem Waschsalonvergleich dürfen wir keine neuen Waschlösungen mehr in die Waschmaschine füllen, so dass wir mit der falsch vorhergesagten Ladung von vorn beginnen können.

Wie bei allen anderen Möglichkeiten zur Auflösung von Steuerkonflikten verschärft sich das Problem bei längeren Pipelines, in diesem Fall durch eine Zunahme der Kosten durch falsche Vorhersagen. Möglichkeiten zur Auflösung von Steuerkonflikten werden in Abschnitt 6.6 ausführlicher beschrieben.

Vertiefung: Es gibt einen dritten Ansatz zur Auflösung des Steuerkonflikts. Dieser Ansatz wird als *verzögerte Entscheidung* bezeichnet. Bei unserem Beispiel würden Sie immer, wenn Sie eine Entscheidung dieser Art bezüglich einem Typ Wäsche treffen, einfach eine Ladung Wäsche eines anderen Typs in die Waschmaschine geben, während Sie darauf warten, dass die Wäsche des ersten Typs trocknet. Solange Sie genügend Wäsche zum Waschen haben, die von diesem Test nicht betroffen ist, funktioniert diese Lösung gut.

Die von der MIPS-Architektur verwendete Lösung für Rechner wird als *verzögerter Sprung* bezeichnet. Der verzögerte Sprung führt immer den nächsten Befehl in Folge aus, wobei der Sprung *nach* dieser Verzögerung ausgeführt wird. Dies bleibt dem in MIPS-Assemblersprache Programmierenden verborgen, da der Assembler die Befehle automatisch so anordnet, um das vom Programmierer gewünschte Sprungverhalten zu erzielen. Die MIPS-Software fügt einen Befehl direkt nach dem verzögerten Sprungbefehl ein, der vom Sprung nicht abhängig ist, und ein ausgeführter Sprung ändert die Adresse des Befehls *nach* diesem sicheren Befehl. In unserem Beispiel hat der `add`-Befehl vor dem Sprung in Abbildung 6.6 keine Auswirkung auf den Sprung und kann hinter den Sprung verschoben werden, um die Sprungverzögerung komplett zu verbergen. Da verzögerte Sprünge bei kurzen Sprüngen hilfreich sind, verwendet kein Prozessor einen um mehr als einen Zyklus verzögerten Sprung. Für längere Sprungverzögerungen wird in der Regel eine hardwareunterstützte Sprungvorhersage verwendet.



Zusammenfassung: Die Technik des Pipelining

Pipelining ist eine Technik, die die Parallelität zwischen den Befehlen in einer Befehlssequenz nutzt. Sie hat den bedeutenden Vorteil, dass sie im Gegensatz zu einigen anderen Beschleunigungstechniken (siehe Kapitel 9) für den Programmierer im Wesentlichen unsichtbar ist.

In den nächsten Abschnitten dieses Kapitels stellen wir das Konzept des Pipelining anhand der MIPS-Befehle `lw`, `sw`, `add`, `sub`, `and`, `or`, `slt` und `beq` (dieselben wie in Kapitel 5) und einer vereinfachten Version der dazu gehörenden Pipeline vor. Anschließend werden wir uns mit den Problemen, die durch das Pipelining entstehen sowie mit der unter typischen Situationen erzielbaren Leistung befassen.

Wenn Sie eine detailliertere Betrachtung der Software und der durch Pipelining erzielbaren Leistung wünschen, verfügen Sie nun über genügend Hintergrundwissen, um zum Abschnitt 6.9 zu springen. Dort werden erweiterte Pipelining-Konzepte wie superskalares und dynamisches Scheduling vorgestellt. In Abschnitt 6.10 wird die Pipeline des Pentium-4-Mikroprozessors untersucht.

Wenn Sie wissen möchten, wie Pipelining implementiert wird und wie Konflikte aufgelöst werden, können Sie mit der Beschreibung des Pipelining-Entwurfs für einen Datenpfad in Abschnitt 6.2 und der zugrunde liegenden Steuerung in Abschnitt 6.3 fortfahren. Mithilfe dieses neu erworbenen Wissens können Sie in Abschnitt 6.4 nachlesen, wie Forwarding implementiert wird, und in Abschnitt 6.5, wie Verzögerungen implementiert werden. Anschließend lernen Sie in Abschnitt 6.6 weitere Möglichkeiten zur Auflösung von Steuerkonflikten kennen, und in Abschnitt 6.8 erfahren Sie, wie Unterbrechungen behandelt werden.

Durch das Pipelining werden die Anzahl der gleichzeitig ausgeführten Befehle und der Takt, mit dem Befehle gestartet und abgeschlossen werden, erhöht. Mit dem Pipelining wird nicht die zum Ausführen eines einzelnen Befehls erforderliche Zeit verkürzt, die auch als **Latenz (pipeline)** bezeichnet wird. So benötigt beispielsweise die Ausführung eines Befehls in einer fünfstufigen Pipeline nach wie vor 5 Taktzyklen. Mit den in Kapitel 4 verwendeten Bezeichnungen ausgedrückt, verbessert das Pipelining anstelle der *Ausführungszeit* oder der *Latenz* einzelner Befehle den *Befehlsdurchsatz*.

Befehlssätze können Entfernen von Pipelines, die bereits mit Struktur-, Steuer- und Datenkonflikten zureckkommen müssen, das Leben erleichtern oder erschweren. Sprungvorhersage, Forwarding und Verzögerungen tragen dazu bei, dass ein Computer schneller wird und dennoch die richtigen Ergebnisse liefert.

Latenz (pipeline) Die Anzahl der Stufen in einer Pipeline oder die Anzahl der Stufen zwischen zwei Befehlen in der Ausführung.

Grundlegendes zur Leistungsfähigkeit von Programmen

Abgesehen vom Speichersystem ist die effiziente Arbeitsweise der Pipeline in der Regel der wichtigste Faktor zum Bestimmen des CPI-Werts und somit der Leistung des Prozessors. Wie wir in Abschnitt 6.9 sehen werden, ist das Verständnis der Leistung eines modernen Prozessors mit Mehrfachzuordnung und Pipelining nicht einfach und erfordert Kenntnisse, die über die Fragestellungen hinausgehen, die sich bei einem Prozessor mit einfacherem Pipelining ergeben. Dennoch bleiben Struktur-, Daten- und Steuerkonflikte sowohl bei einfachen als auch bei anspruchsvollerden Pipelines wichtig.

Bei modernen Pipelines entwickeln sich Strukturkonflikte in der Regel um die Gleitkommaeinheit, die möglicherweise nicht vollständig als Pipeline aufgebaut ist, während Steuerkonflikte eher bei Ganzzahlprogrammen ein Problem darstellen, die höhere Sprunghäufigkeit und weniger vorhersagbare Sprünge aufweisen. Datenkonflikte können sowohl bei Ganzzahl- als auch bei Gleitkommaprogrammen Leistungsengpässe darstellen. Häufig ist der Umgang mit Datenkonflikten bei Gleitkommaprogrammen einfacher, da der Compiler aufgrund der geringeren Sprunghäufigkeit und des regelmäßigeren Zugriffsmusters Befehle so anordnen kann, dass Konflikte vermieden werden. Optimierungen dieser Art bei Ganzzahlprogrammen zu realisieren, die weniger regelmäßige Zugriffsmuster aufweisen und häufiger Zeiger verwenden, ist entsprechend schwieriger. Wie wir in Abschnitt 6.9 sehen werden, gibt es ehrgeizigere Compiler- und Hardwaretechniken zum Reduzieren von Datenabhängigkeiten durch Scheduling.

Geben Sie für jede der folgenden Codesequenzen an, ob die Pipeline angehalten werden muss, ob Verzögerungen nur mit Forwarding vermieden werden können oder ob die Codesequenzen ohne Verzögerung oder Forwarding ausgeführt werden können:



Sequenz 1	Sequenz 2	Sequenz 3
lw \$t0,0 (\$t0) add \$t1,\$t0,\$t0	add \$t1,\$t0,\$t0 addi \$t2,\$t0,#5 addi \$t4,\$t1,#5	addi \$t1,\$t0,#1 addi \$t2,\$t0,#2 addi \$t3,\$t0,#2 addi \$t3,\$t0,#4 addi \$t5,\$t0,#5

There is less in this than meets the eye.

Tallulah Bankhead, *Bemerkung an Alexander Wollcott*, 1922

6.2 Pipelining des Datenpfads

In Abbildung 6.8 ist der Eintaktdatenpfad aus Kapitel 5 dargestellt. Die Aufteilung eines Befehls in fünf Stufen erfordert eine fünfstufige Pipeline, was wiederum bedeutet, dass sich während eines Taktzyklus bis zu fünf Befehle in der Ausführung befinden können. Somit müssen wir den Datenpfad in fünf Teile unterteilen, wobei jeder Teil nach der jeweiligen Befehlausführungsstufe benannt wird:

1. IF: Instruction Fetch (Befehl holen)
2. ID: Instruction Decode and register file read (Befehl entschlüsseln und Registersatz lesen)
3. EX: EXecution or address calculation (Befehl ausführen oder Adresse berechnen)
4. MEM: data MEMory access (auf Datenspeicher zugreifen)
5. WB: Write Back (Ergebnis rückschreiben)

In Abbildung 6.8 entsprechen diese fünf Komponenten im Wesentlichen der Darstellung des Datenpfads. Befehle und Daten durchlaufen bei der Ausführung die fünf Stufen im Allgemeinen von links nach rechts. Bei unserem Beispiel mit dem Wäsche-waschen wird die Wäsche immer sauberer, trockener und geordneter je weiter sie sich in der Pipeline befindet und keines der Kleidungsstücke wandert rückwärts.

Zu diesem Befehlsfluss von links nach rechts gibt es jedoch zwei Ausnahmen:

- Die Rückschreibstufe, in der das Ergebnis an den Registersatz in der Mitte des Datenpfads zurückgesendet wird
- Die Auswahl des nächsten Werts des Befehlszählers, wobei zwischen dem inkrementierten Befehlszähler und der Sprungadresse aus der MEM-Stufe ausgewählt wird

Daten, die von rechts nach links fließen, haben keine Auswirkungen auf den aktuellen Befehl. Nur Befehle, die die Pipeline später durchlaufen, sind von diesen Rückwärtsbewegungen von Daten betroffen. Beachten Sie, dass der erste Pfeil von rechts nach links zu einem Datenkonflikt führen kann und dass der zweite Pfeil Steuerkonflikte hervorruft.

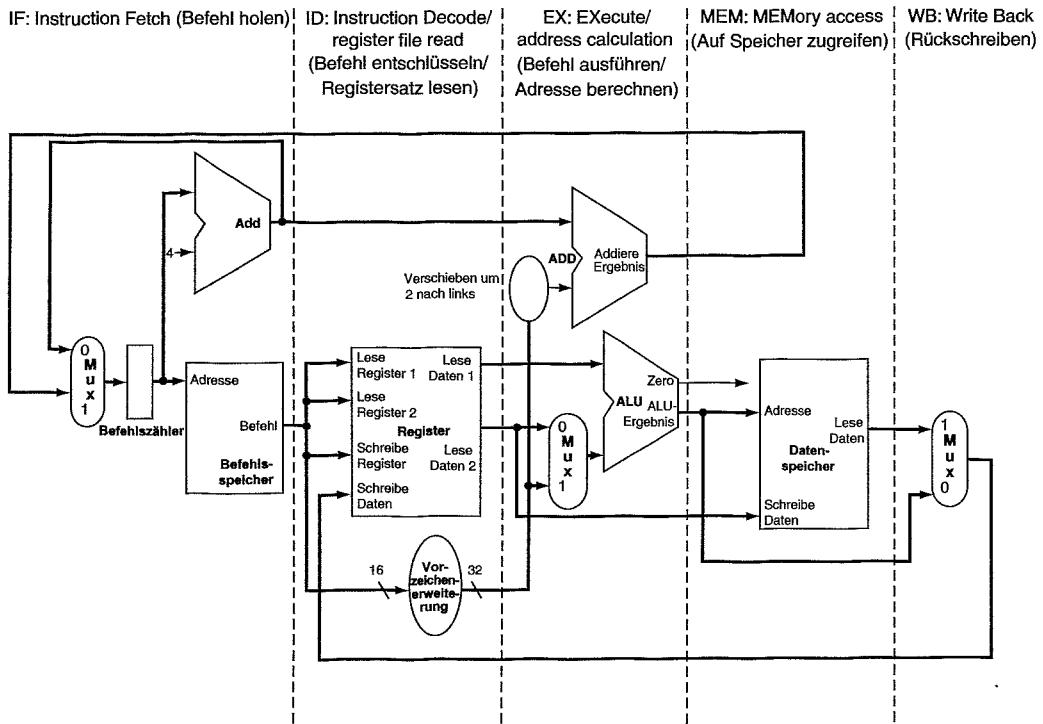


Abb. 6.8 Der Eintaktdatenpfad aus Kapitel 5 (vgl. Abbildung 5.14). Alle Befehlsschritte können dem Datenpfad von links nach rechts zugeordnet werden. Die einzigen Ausnahmen bilden die Aktualisierung des Befehlszählers und der Rückschreibschritt, die farblich hervorgehoben sind und bei denen entweder das ALU-Ergebnis oder die Daten aus dem Speicher zum Schreiben in den Registersatz nach links gesendet werden. (Normalerweise heben wir Steuerleitungen farblich hervor. In diesem Fall handelt es sich jedoch um Datenleitungen.)

Eine Möglichkeit darzustellen, was bei der Befehlausführung mittels Pipeline geschieht, besteht darin, so zu tun, als hätte jeder Befehl einen eigenen Datenpfad, und die daraus resultierenden Datenpfade entlang einer Zeitachse anzutragen, um deren Beziehung zueinander darzustellen. In Abbildung 6.9 ist die Ausführung der Befehle aus Abbildung 6.2 dargestellt, wobei die einzelnen Datenpfade auf einer gemeinsamen Zeitachse dargestellt sind. Wir haben eine vereinfachte Version des Datenpfads aus Abbildung 6.8 gewählt, um die Beziehung der Datenpfade zueinander in Abbildung 6.9 zu verdeutlichen.

Nach Abbildung 6.9 scheint es so, als benötigten drei Befehle drei Datenpfade. In Kapitel 5 haben wir zusätzliche Register zum Speichern von Daten verwendet, so dass Teile des Datenpfads während der Befehlausführung von mehreren Befehlen genutzt werden konnten. Wir wenden hier dieselbe Technik an und nutzen mehrere Datenpfade für mehrere Befehle. Wie Abbildung 6.9 zu entnehmen ist, wird der Befehlsspeicher für einen Befehl nur in einer der fünf Stufen verwendet, so dass er von anderen Befehlen während der anderen vier Stufen verwendet werden kann.

Damit der Wert eines einzelnen Befehls für seine anderen vier Stufen nicht verloren geht, muss der aus dem Befehlsspeicher gelesene Wert in einem Register gespeichert werden. Ähnliche Argumente gelten für alle anderen Pipelinestufen. Also müssen wir an allen Stellen Register einfügen, wo sich in Abbildung 6.8 Grenzen zwischen den Stufen der Pipeline befinden. Diese Änderung erinnert an die Register, die in Kapitel 5 beim Wechsel von einem Eintaktdatenpfad zu einem Mehrzyklendatenpfad eingefügt wurden. In unserem Vergleich mit dem Wäschewaschen können wir einen Korb zwischen je zwei Stufen aufstellen, in den wir die Wäsche für die nächste Stufe legen können.

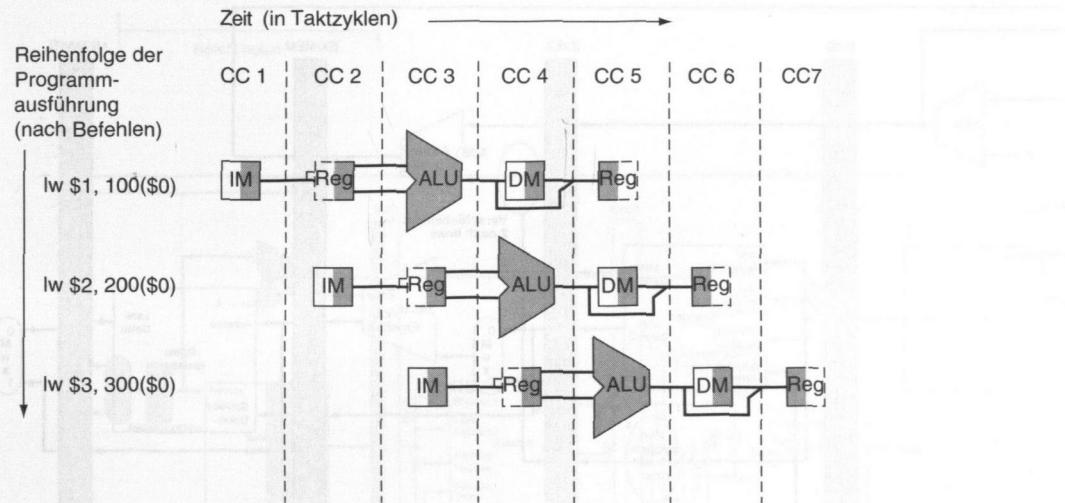


Abb. 6.9 Befehle, die mit dem Eintaktdatenpfad aus Abbildung 6.8 ausgeführt werden, wobei von einer Ausführung mittels Pipeline ausgegangen wird. Ähnlich wie bei den Abbildungen 6.3 bis 6.5 wird bei dieser Abbildung so getan, als hätte jeder Befehl einen eigenen Datenpfad, und jeder Teil ist entsprechend der Nutzung schattiert. Im Gegensatz zu diesen Abbildungen sind die einzelnen Stufen nach der in der jeweiligen Stufe verwendeten Hardwareressource entsprechend den Teilen im Datenpfad in Abbildung 6.8 benannt. IM steht für Instruction Memory (Befehlsspeicher) und Befehlszähler in der Befehlsholstufe, Reg steht für Registersatz und Vorzeichenerweiterungseinheit in der Befehlsentschlüsselungs-/Registerlesestufe (ID) usw. Um die richtige zeitliche Reihenfolge aufrechtzuerhalten, wird bei diesem vereinfachten Datenpfad der Registersatz in zwei logische Teile aufgeteilt: Register, die während der Registerholstufe (ID) gelesen werden, und Register, in die während der Rückschreibstufe (WB) geschrieben wird. Diese doppelte Nutzung wird dargestellt, indem die nicht schattierte linke Hälfte des Registersatzes in der ID-Stufe mit gestrichelter Linie dargestellt wird, wenn der Registersatz nicht beschrieben wird, und indem die nicht schattierte rechte Hälfte in der WB-Stufe mit gestrichelter Linie dargestellt wird, wenn der Registersatz nicht gelesen wird. Wie zuvor gehen wir auch hier davon aus, dass der Registersatz in der ersten Hälfte des Taktzyklus beschrieben und während der zweiten Hälfte gelesen wird.

In Abbildung 6.10 ist das Pipelining des Datenpfads dargestellt, wobei die Pipelineregister farblich hervorgehoben sind. Alle Befehle rücken während der einzelnen Taktzyklen von einem zum nächsten Pipelineregister vor. Die Register werden nach den beiden Stufen benannt, die durch sie getrennt werden. So heißt das Pipelineregister zwischen der IF-Stufe und der ID-Stufe beispielsweise IF/ID-Register.

Am Ende der Rückschreibstufe gibt es kein Pipelineregister. Alle Befehle müssen einen Zustand im Prozessor (den Registersatz, den Speicher oder den Befehlszähler) aktualisieren, so dass für den aktualisierten Zustand kein eigenes Pipelineregister erforderlich ist. Ein Ladebefehl speichert beispielsweise sein Ergebnis in einem der 32 Register und jeder spätere Befehl, der diese Daten benötigt, liest einfach das entsprechende Register.

Jeder Befehl aktualisiert den Befehlszähler entweder durch Inkrementieren oder dadurch, dass er ihn mit einer Sprungzieladresse beschreibt. Sie können sich den Befehlszähler als ein Pipelineregister vorstellen, das der IF-Stufe der Pipeline die gewünschten Daten bereitstellt. Im Gegensatz zu den schattierten Pipelineregistern in Abbildung 6.10, ist der Befehlszähler jedoch Teil der sichtbaren Architektur. Sein Inhalt muss gerettet werden, wenn eine Unterbrechung auftritt, während der Inhalt der Pipelineregister nicht berücksichtigt werden muss. In dem Beispiel mit dem Wäschewaschen wäre der Befehlszähler der Korb, in den Sie die Ladung schmutziger Wäsche vor dem Waschschrifft legen!

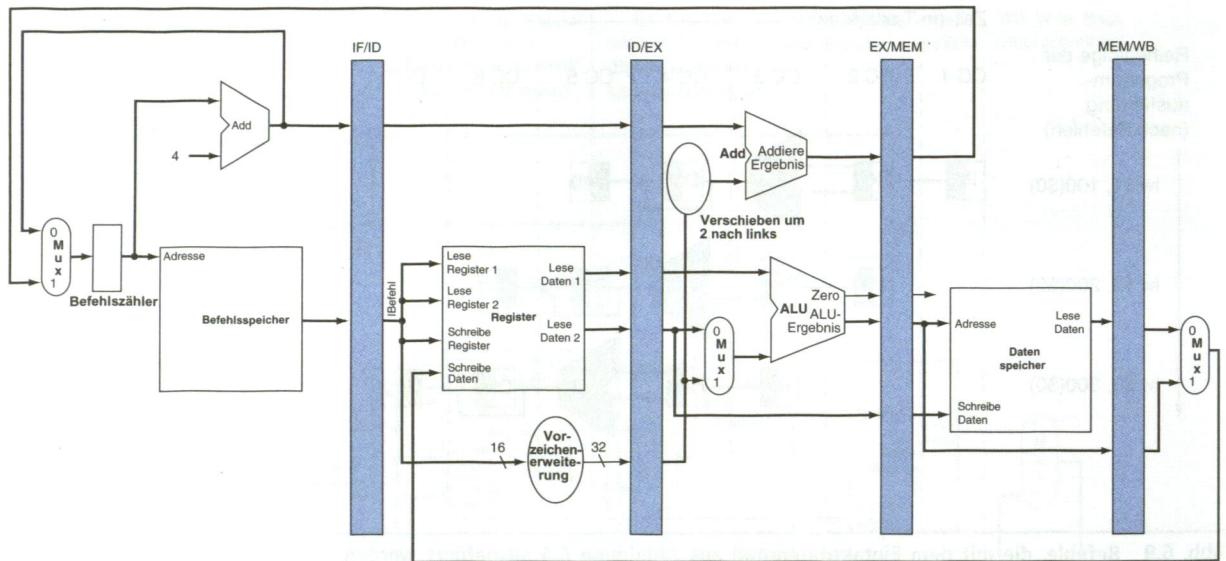


Abb. 6.10 Die Version des Datenpfads in Abbildung 6.8 mit Pipelining. Die farblich hervorgehobenen Pipelineregister trennen die einzelnen Pipelinestufen voneinander. Sie sind nach den Stufen benannt, die sie trennen. So heißt beispielsweise das erste Register IF/ID-Register, da es der Befehlshol- von der Befehlsentschlüsselungsstufe trennt. Die Register müssen breit genug sein, um all die Daten entsprechend der Leitungen, die durch sie hindurchführen, speichern zu können. So muss das IF/ID-Register beispielsweise 64 Bit breit sein, da in diesem Register sowohl der aus dem Speicher geholte 32-Bit-Befehl als auch die inkrementierte 32-Bit-Befehlszähleradresse gespeichert werden muss. Diese Register werden wir im Laufe dieses Kapitels noch erweitern. Im Moment umfassen die anderen drei Pipelineregister jedoch jeweils 128 Bit, 97 Bit und 64 Bit.

Um zu zeigen, wie das Pipelining funktioniert, werden wir in diesem Kapitel ihre Funktionsweise in Abhängigkeit der Zeit anhand von einzelnen Sequenzen der Abbildungen erläutern. Es scheint, als wäre zum Verstehen dieser zusätzlichen Seiten eine Menge Zeit erforderlich. Aber keine Angst. Für die Sequenzen benötigen Sie viel weniger Zeit als es zunächst scheinen mag, da Sie die Sequenzen miteinander vergleichen und so feststellen können, was sich in den einzelnen Taktzyklen ändert. In den Abschnitten 6.4 und 6.5 wird beschrieben, was geschieht, wenn zwischen Befehlen in Pipelines Datenkonflikte auftreten. Im Moment können Sie diese noch ignorieren.

In den Abbildungen 6.11 bis 6.13, unserer ersten Sequenz, sind die aktiven Teile der fünf Pipelinestufen des Datenpfads farblich hervorgehoben, die ein Ladebefehl durchläuft. Wir stellen einen Ladebefehl als Erstes dar, weil er in allen fünf Stufen aktiv ist. Wie in den Abbildungen 6.3 bis 6.10 ist die *rechte Hälfte* der Register oder des Speichers hervorgehoben, wenn *gelesen* wird, und die *linke Hälfte*, wenn *geschrieben* wird. In jeder Abbildung ist die Abkürzung des Befehls 1w und der Name der Pipelinestufe angegeben, die jeweils aktiv ist. Die folgenden fünf Stufen sind dargestellt:

1. *Instruction fetch (Befehl holen)*: Im oberen Teil der Abbildung 6.11 ist der Befehl dargestellt, der mithilfe der im Befehlszähler enthaltenen Adresse aus dem Speicher gelesen und im IF/ID-Pipelinerregister gespeichert wird. Das IF/ID-Pipelinerregister ist dem Befehlsregister in Abbildung 5.21, Seite 264 ähnlich. Die Adresse im Befehlszähler wird um vier inkrementiert, anschließend wieder in den Befehlszähler geschrieben und steht so für den nächsten Taktzyklus bereit. Diese inkrementierte Adresse wird außerdem im IF/ID-Pipelinerregister gespeichert, falls sie zu einem späteren Zeitpunkt für einen Befehl wie etwa den *beq*-Befehl benötigt wird. Der Computer kann nicht wissen, welcher Befehlstyp geholt wird. Also muss er auf jeden Befehl vorbereitet sein und möglicherweise benötigte Informationen für die Pipeline weiterleiten.

6.2 Pipelining des Datenpfads

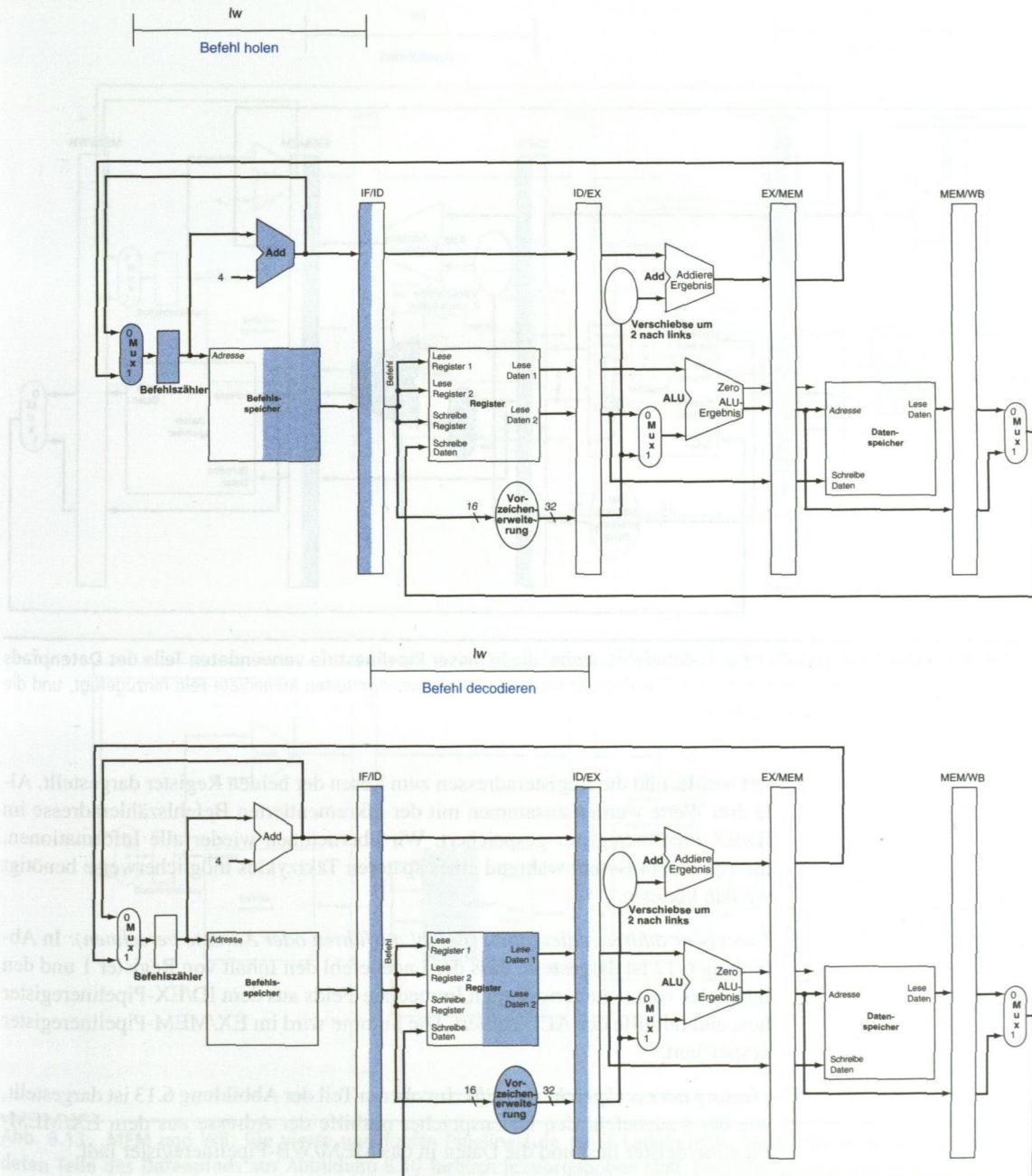


Abb. 6.11 IF und ID: Die erste und zweite Pipelinestufe eines Befehls, wobei die aktiven Teile des Datenpfads aus Abbildung 6.10 farblich hervorgehoben sind. Die Konvention für die farbliche Hervorhebung ist dieselbe wie die in Abbildung 6.3. Wie in Kapitel 5 kommt es beim Lesen und Beschreiben der Register zu keinem Durcheinander, da der Inhalt nur bei der Taktflanke geändert wird. Obwohl der Ladebefehl in Stufe 2 nur die oberen Register benötigt, weiß der Prozessor nicht, welcher Befehl entschlüsselt wird, weshalb er die 16-Bit-Konstante mit Vorzeichen erweitert und beide Register in das ID/EX-Pipelinerregister liest. Wir benötigen nicht alle drei Operanden, aber es vereinfacht die Steuerung, wenn wir alle drei erhalten.

2. *Instruction decode and register file read (Befehl entschlüsseln/Register lesen):* Im unteren Teil der Abbildung 6.11 ist der Befehlsteil des IF/ID-Pipelinerregisters, der das 16-Bit-immediate-Feld bereitstellt, das mit dem Vorzeichen auf 32 Bit erwei-

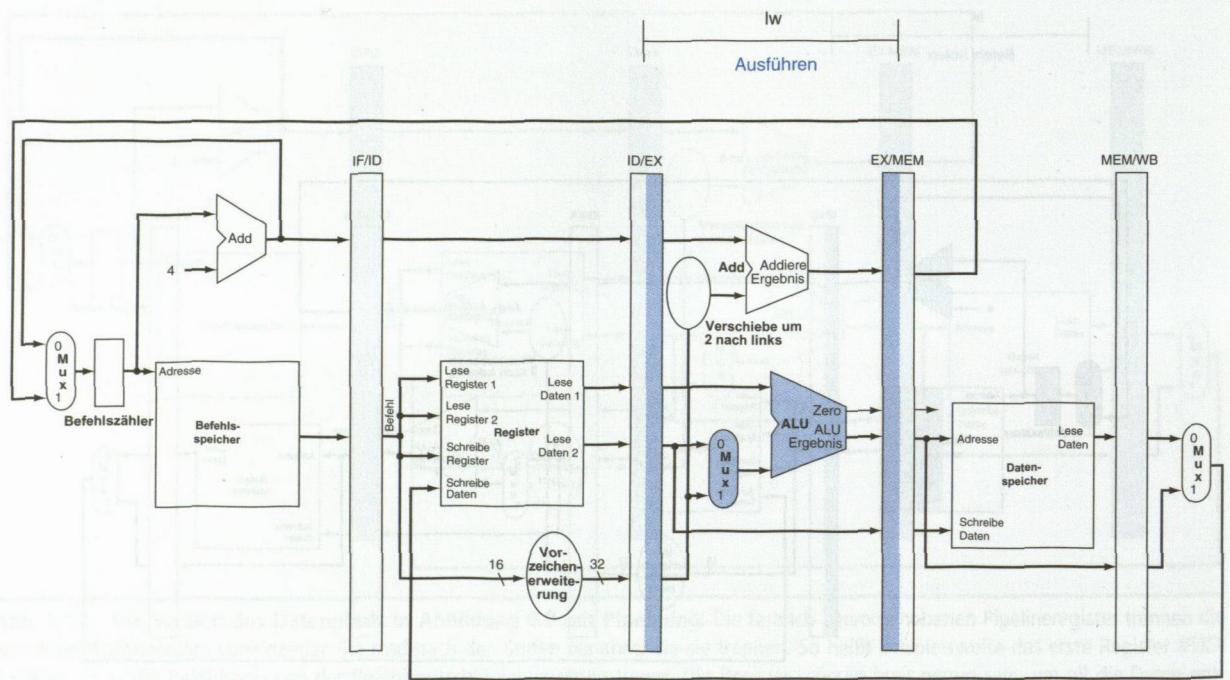


Abb. 6.12 EX: Die dritte Pipelinestufe eines Ladebefehls, wobei die in dieser Pipelinestufe verwendeten Teile des Datenpfads aus Abbildung 6.10 farblich hervorgehoben sind. Das Register wird zum vorzeichenverweiterten Immediate-Feld hinzugefügt, und die Summe wird im EX/MEM-Pipelinerregister gespeichert.

tert wurde, und die Registeradressen zum Lesen der beiden Register dargestellt. Alle drei Werte werden zusammen mit der inkrementierten Befehlszähleradresse im ID/EX-Pipelinerregister gespeichert. Wir übernehmen wieder alle Informationen, die von einem Befehl während eines späteren Taktzyklus möglicherweise benötigt werden könnten.

3. *Execute or address calculation (Befehl ausführen oder Adresse berechnen):* In Abbildung 6.12 ist dargestellt, dass der Ladebefehl den Inhalt von Register 1 und den Inhalt des vorzeichenverweiterten Immediate-Felds aus dem ID/EX-Pipelinerregister liest und mithilfe der ALU addiert. Die Summe wird im EX/MEM-Pipelinerregister gespeichert.
4. *Memory access (Speicherzugriff):* Im oberen Teil der Abbildung 6.13 ist dargestellt, wie der Ladebefehl den Datenspeicher mithilfe der Adresse aus dem EX/MEM-Pipelinerregister liest und die Daten in das MEM/WB-Pipelinerregister lädt.
5. *Write back (Ergebnis rückschreiben):* Im unteren Teil der Abbildung 6.13 ist der letzte Schritt dargestellt: Lesen der Daten aus dem MEM/WB-Pipelinerregister und Schreiben der Daten in den Registersatz in der Mitte der Abbildung.

Anhand dieser Beschreibung des Wegs durch eine Pipeline am Beispiel des Ladebefehls wird deutlich, dass sämtliche Daten, die in einer späteren Pipelinestufe benötigt werden, über ein Pipelinerregister an diese Stufe übergeben werden müssen. Anhand der Beschreibung eines Speicherbefehls wird die Ähnlichkeit der Befehlsausführung sowie der Übergabe der Daten für die späteren Stufen in der Pipeline deutlich. Der Speicherbefehl durchläuft die folgenden fünf Pipelinestufen:

1. *Instruction fetch (Befehl holen):* Der Befehl wird mithilfe der im Befehlszähler gespeicherten Adresse aus dem Speicher gelesen und im IF/ID-Pipelinerregister ge-

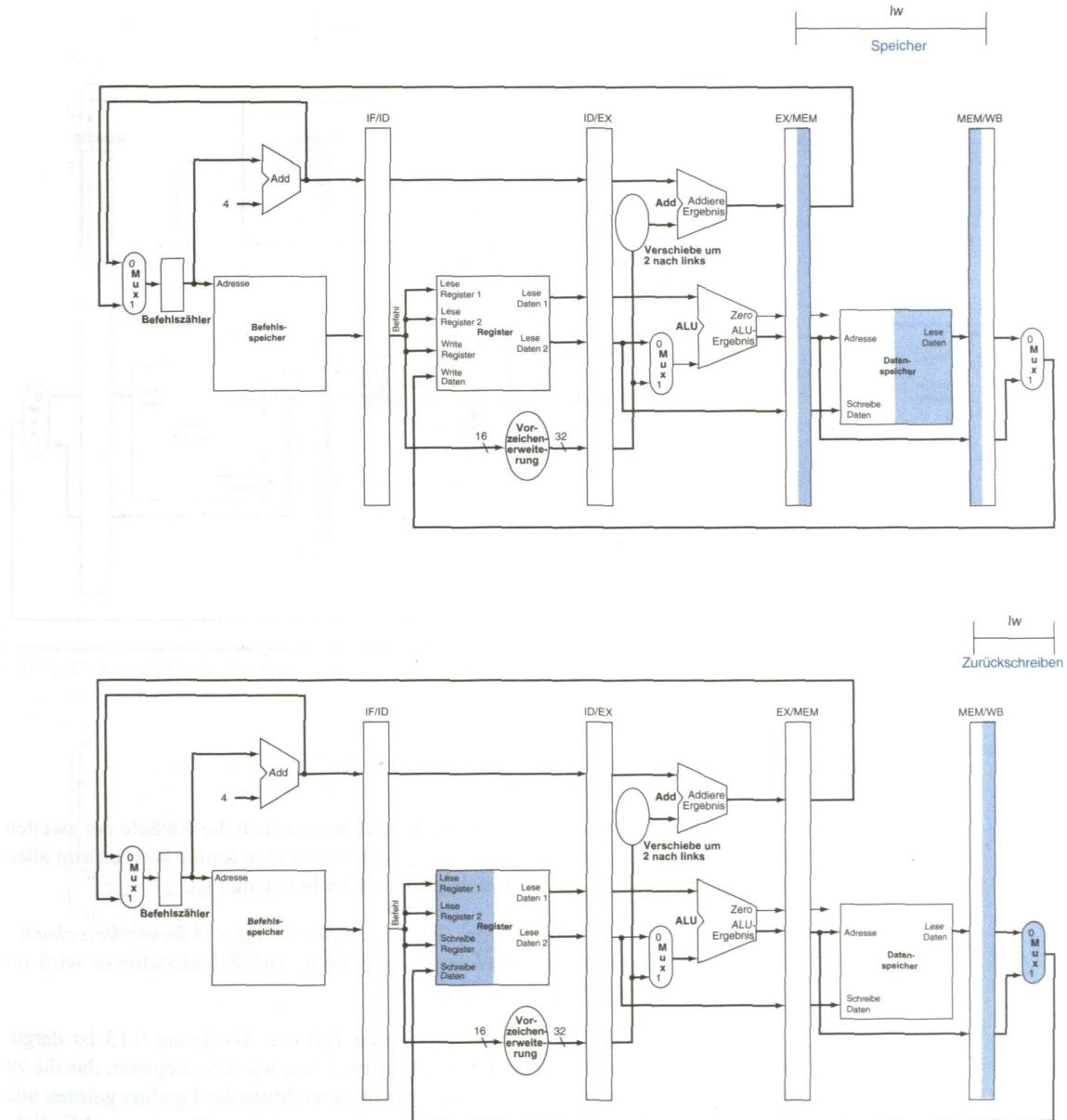


Abb. 6.13 MEM und WB: Die vierte und fünfte Pipelinestufe eines Ladebefehls, wobei die in dieser Pipelinestufe verwen- deten Teile des Datenpfads aus Abbildung 6.10 farblich hervorgehoben sind. Der Datenspeicher wird mithilfe der in den EX/MEM-Pipelinerregistern gespeicherten Adresse gelesen, und die Daten werden im MEM/WB-Pipelinerregister gespeichert. Als Nächstes werden Daten aus dem MEM/WB-Pipelinerregister gelesen und in den Registersatz in der Mitte des Datenpfads geschrieben.

speichert. Diese Stufe kommt vor dem Entschlüsseln des Befehls, d.h. der obere Teil von Abbildung 6.11 arbeitet bei Speicher- und Ladebefehlen gleich.

2. *Instruction decode and register file read (Befehl entschlüsseln/Register lesen):* Der Befehl im IF/ID-Pipelinerregister stellt die Registeradressen zum Lesen zweier Register bereit und erweitert das 16-Bit-immediate-Feld mit dem Vorzeichen. Diese drei 32-Bit-Werte werden zusammen im ID/EX-Pipelinerregister gespeichert. Im

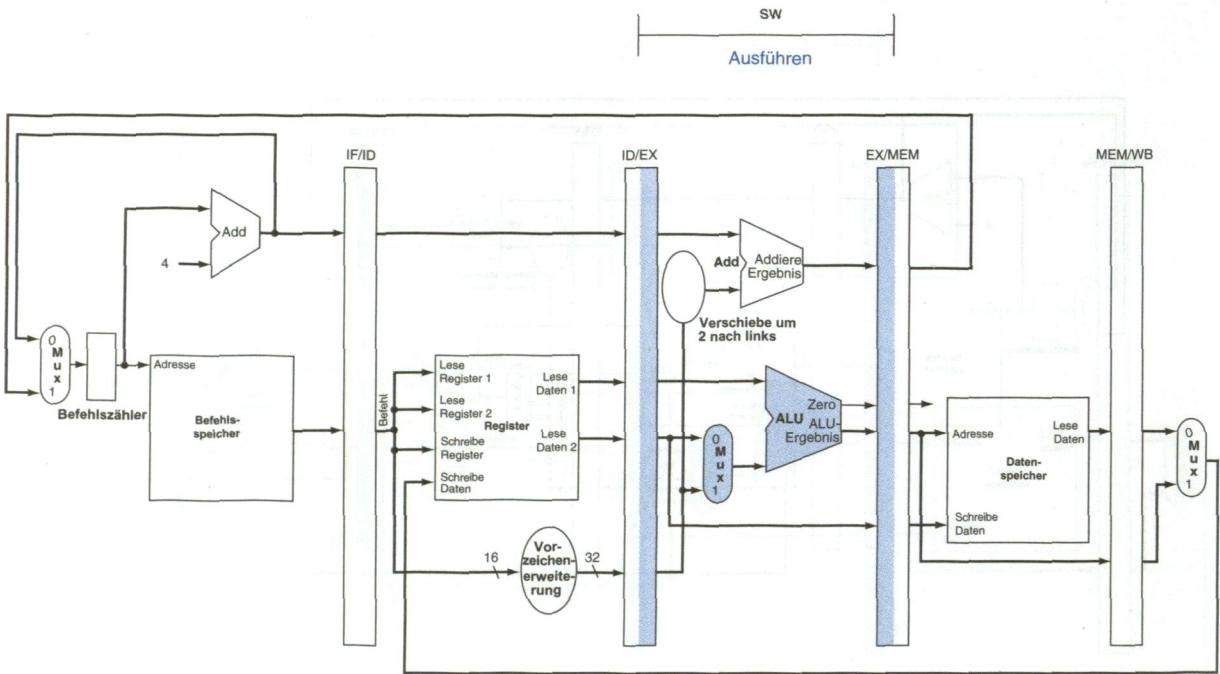
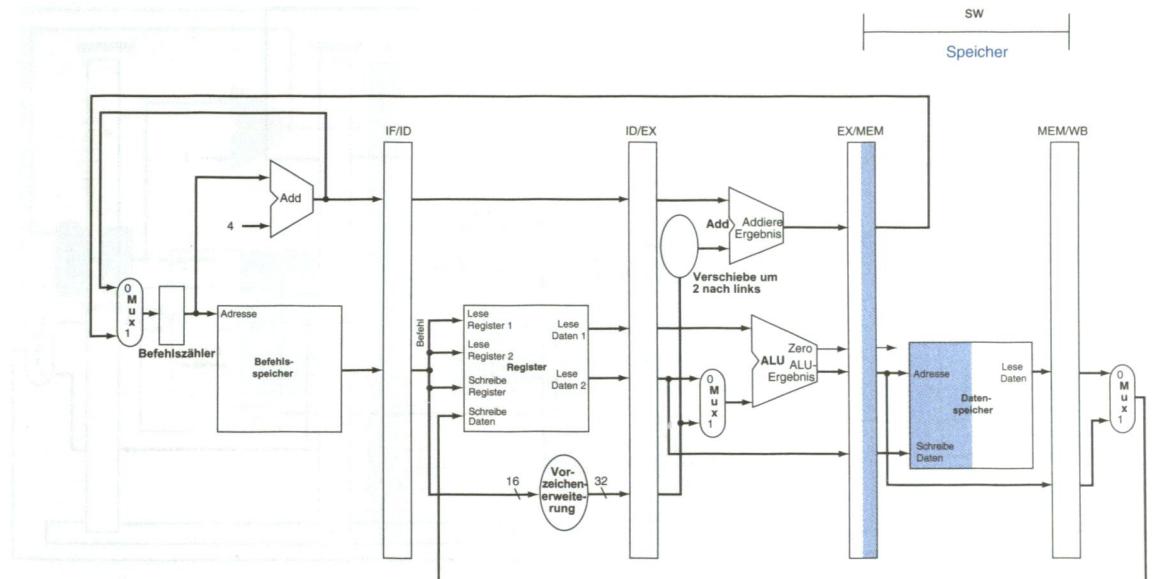


Abb. 6.14 EX: Die dritte Pipelinestufe eines Speicherbefehls. Im Gegensatz zur dritten Stufe des Ladebefehls in Abbildung 6.12 wird der zweite Registerwert für die Verwendung in der nachfolgenden Stufe in das EX/MEM-Pipelinerregister geladen. Obwohl es nicht schaden würde, dieses zweite Register immer in das EX/MEM-Pipelinerregister zu schreiben, schreiben wir das zweite Register nur bei einem Speicherbefehl, damit die Pipeline leichter zu verstehen ist.

unteren Teil der Abbildung 6.11 für Ladebefehle sind auch die Abläufe der zweiten Stufe für Speicherbefehle dargestellt. Diese ersten beiden Stufen werden von allen Befehlen durchlaufen, da der Befehlstyp noch nicht bekannt ist.

3. *Execute and address calculation (Befehl ausführen oder Adresse berechnen):* In Abbildung 6.14 ist die dritte Stufe dargestellt. Die Effektivadresse wird im EX/MEM-Pipelinerregister gespeichert.
4. *Memory access (Speicherzugriff):* Im oberen Teil der Abbildung 6.15 ist dargestellt, wie die Daten in den Speicher geschrieben werden. Das Register, das die zu speichernden Daten enthält, wurde in einer früheren Stufe der Pipeline gelesen und dessen Inhalt wurde im ID/EX-Pipelinerregister gespeichert. Die einzige Möglichkeit, die Daten während der MEM-Stufe verfügbar zu machen, besteht darin, die Daten in der EX-Stufe im EX/MEM-Pipelinerregister zu speichern, ganz analog zur Speicherung der Effektivadresse im EX/MEM-Pipelinerregister.
5. *Write back (Ergebnis rückschreiben):* Im unteren Teil der Abbildung 6.15 ist der letzte Schritt des Speicherbefehls dargestellt. Bei diesem Befehl geschieht in der Rückschreibstufe nichts. Da alle Befehle nach dem Speicherbefehl bereits in der Pipeline ausgeführt werden, haben wir keine Möglichkeit, diese Befehle zu beschleunigen. Somit durchläuft ein Befehl eine Stufe, auch wenn es in dieser Stufe nichts zu tun gibt, da Befehle weiter hinten in der Pipeline bereits mit maximaler Geschwindigkeit ausgeführt werden.

Anhand des Speicherbefehls wird ebenfalls deutlich, dass zum Übergeben von Daten von einer früheren Stufe in der Pipeline an eine spätere Stufe in der Pipeline diese Daten in einem Pipelineregister gespeichert werden müssen. Geschieht dies



arbeitet auf die zuletzt von mir gezeigte Variante und ist ebenfalls für den Speicherbefehl gültig.

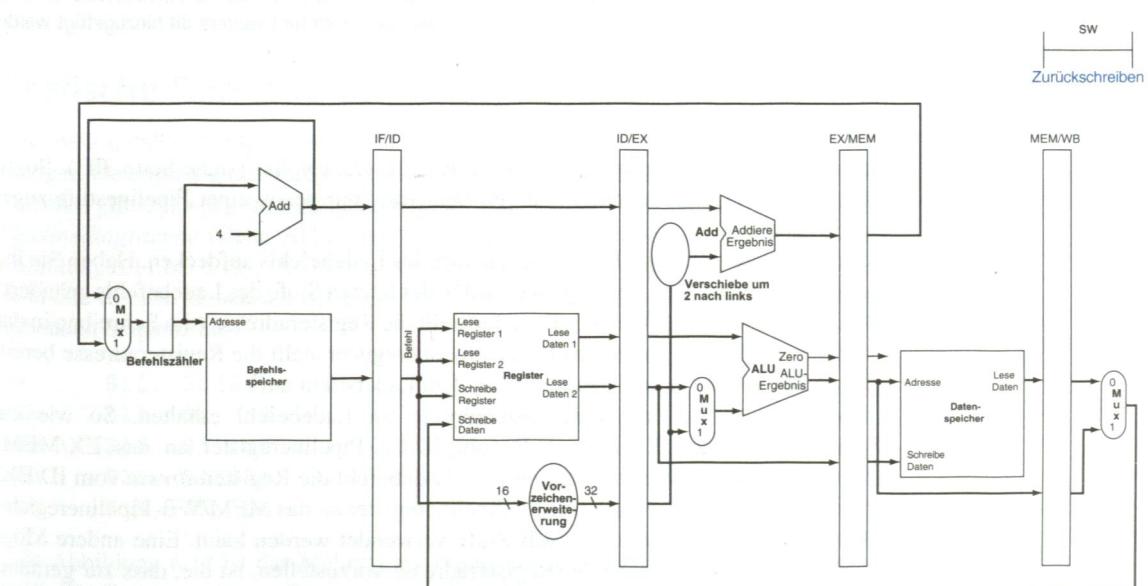


Abb. 6.15 MEM und WB: Die vierte und fünfte Stufe eines Speicherbefehls. In der vierten Stufe werden die Daten zum Speichern in den Datenspeicher geschrieben. Die Daten stammen aus dem EX/MEM-Pipelineregister. Im MEM/WB-Pipelineregister wird nichts verändert. Wenn die Daten in den Speicher geschrieben sind, bleibt für den Speicherbefehl nichts mehr zu tun. Somit geschieht in Stufe 5 nichts.

nicht, sind die Daten verloren, wenn der nächste Befehl in diese Pipelinestufe eintritt. Für den Speicherbefehl müssen wir eines der in der ID-Stufe gelesenen Register an die MEM-Stufe übergeben, in der es im Speicher abgelegt wird. Die Daten werden zuerst im ID/EX-Pipelineregister gespeichert und anschließend an das EX/MEM-Pipelineregister übertragen.

Anhand des Lade- und Speicherbefehls wird ein zweiter wichtiger Punkt deutlich: Jede logische Komponente des Datenpfads (wie Befehlsspeicher, Registerleseports,

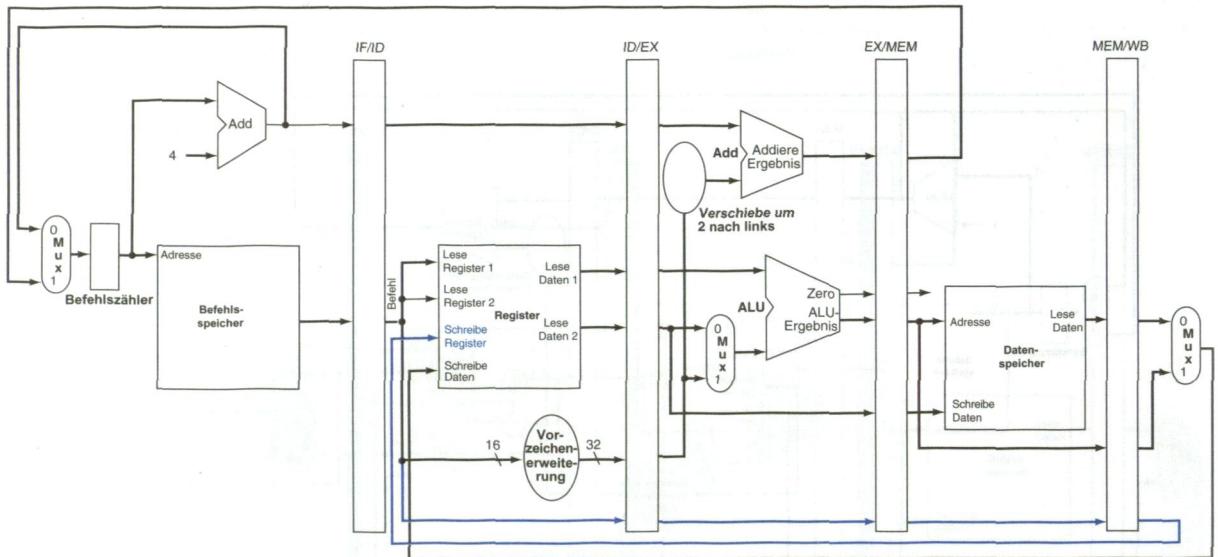


Abb. 6.16 Die korrigierte Pipeline des Datenpfads für eine ordnungsgemäße Bearbeitung des Ladebefehls. Die Registeradresse wird vom MEM/WB-Pipelinerregister zusammen mit den Daten bereitgestellt. Die Registeradresse wird von der ID-Pipelinstufe so lange übergeben, bis sie das MEM/WB-Pipelinerregister erreicht, wodurch in den letzten drei Pipelineregistern fünf weitere Bit hinzugefügt werden müssen. Dieser neue Pfad ist farblich hervorgehoben.

ALU, Datenspeicher und Registerschreibport) kann nur innerhalb einer Pipelinestufe genutzt werden. Andernfalls entsteht ein *Strukturkonflikt* (siehe Seite 303). Somit können diese Komponenten und deren Steuerung nur genau einer Pipelinestufe zugeordnet werden.

Nun können wir einen Fehler im Entwurf des Ladebefehls aufdecken. Haben Sie ihn bereits gefunden? Welches Register wird in der letzten Stufe des Ladebefehls geändert? Genauer ausgedrückt: Welches Register stellt die Registeradresse zum Schreiben in das Register bereit? Der Befehl im IF/ID-Pipelinerregister stellt die Registeradresse bereit. Dieser Befehl tritt jedoch deutlich *nach* dem Ladebefehl auf!

Wir müssen also die Zielregisteradresse im Ladebefehl erhalten. So wie ein Speicherbefehl den Registerinhalt vom ID/EX-Pipelinerregister an das EX/MEM-Pipelinerregister übergeben hat, muss der Ladebefehl die Registeradresse vom ID/EX-Pipelinerregister über das EX/MEM-Pipelinerregister an das MEM/WB-Pipelinerregister übergeben, damit diese in der WB-Stufe verwendet werden kann. Eine andere Möglichkeit, sich die Übergabe der Registeradresse vorzustellen, ist die, dass zur gemeinsamen Nutzung des Pipelinedatenpfads der in der IF-Stufe gelesene Befehl erhalten werden muss, so dass jedes Pipelineregister einen Teil des Befehls enthält, der für diese Stufe und spätere Stufen in der Pipeline benötigt werden.

In Abbildung 6.16 ist die korrigierte Version des Datenpfads dargestellt, bei dem die Registeradresse zum Schreiben in das Register zuerst an das ID/EX-Register, dann an das EX/MEM-Register und schließlich an das MEM/WB-Register übergeben wird. Die Registeradresse wird in der WB-Stufe zum Festlegen des Registers benötigt, in das geschrieben werden soll. Abbildung 6.17 ist eine einfache Darstellung des korrigierten Datenpfads, wobei die in allen fünf Stufen des 1w-Befehls aus Abbildung 6.11 bis 6.13 verwendete Hardware farblich hervorgehoben ist. In Abschnitt 6.6 wird erläutert, was zu tun ist, damit der Sprungbefehl wie erwartet funktioniert.

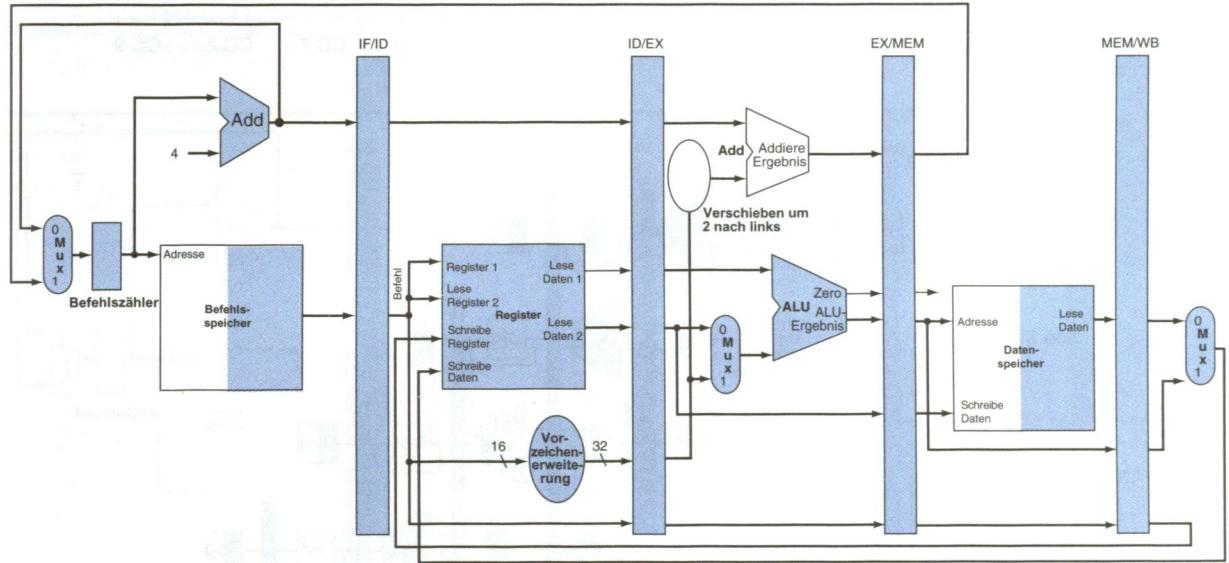


Abb. 6.17 Der Teil des Datenpfads aus Abbildung 6.16, der in allen fünf Stufen eines Ladebefehls verwendet wird.

Grafische Darstellung von Pipelines

Pipelines sind oft schwer verständlich, da in jedem Taktzyklus viele Befehle in einem einzigen Datenpfad gleichzeitig ausgeführt werden. Zum leichteren Verständnis von Pipelines gibt es zwei grundlegende Arten der Darstellung von Pipelines: *Mehrzyklen-Pipelinediagramme* (siehe Abbildung 6.9) und *Einzyklen-Pipelinediagramme* (siehe Abbildung 6.11 bis 6.15). Die Mehrzyklendiagramme sind einfacher, enthalten jedoch nicht alle Details. Gehen wir beispielsweise von der folgenden, aus fünf Befehlen bestehenden Sequenz aus:

```
lw      $10, 20($1)
sub    $11, $2, $3
add    $12, $3, $4
lw      $13, 24($1)
add    $14, $5, $6
```

In Abbildung 6.18 ist das Mehrzyklen-Pipelinediagramm für diese Befehle dargestellt. Die Zeit ist in diesen Diagrammen von links nach rechts über die Seite hinweg abgetragen und Befehle von oben nach unten auf der Seite, ähnlich wie bei der Waschsalonpipeline in Abbildung 6.1. Die Pipelinestufen sind in jedem Teil entsprechend der jeweiligen Taktzyklen entlang der Befehlsachse dargestellt. Diese vereinfachten Datenpfade stellen die fünf Stufen unserer Pipeline dar. Ein Rechteck mit dem Namen der einzelnen Pipelinestufen funktioniert jedoch ebenso gut. In Abbildung 6.19 ist eine traditionellere Version des Mehrzyklen-Pipelinediagramms dargestellt. Während in Abbildung 6.18 die in den einzelnen Stufen verwendeten Hardwareressourcen dargestellt sind, wird in Abbildung 6.19 der *Name* der einzelnen Stufen verwendet. Mehrzyklendiagramme werden hier verwendet, um eine Übersicht über Pipelinesituationen darzustellen.

Einzyklen-Pipelinediagramme stellen den Zustand des gesamten Datenpfades während eines Taktzyklus dar, und in der Regel werden alle fünf Befehle in der Pipeline über den jeweiligen Pipelinestufen angegeben. Mithilfe dieser Art der Abbildung stellen wir ausführlich dar, was in der Pipeline während der einzelnen

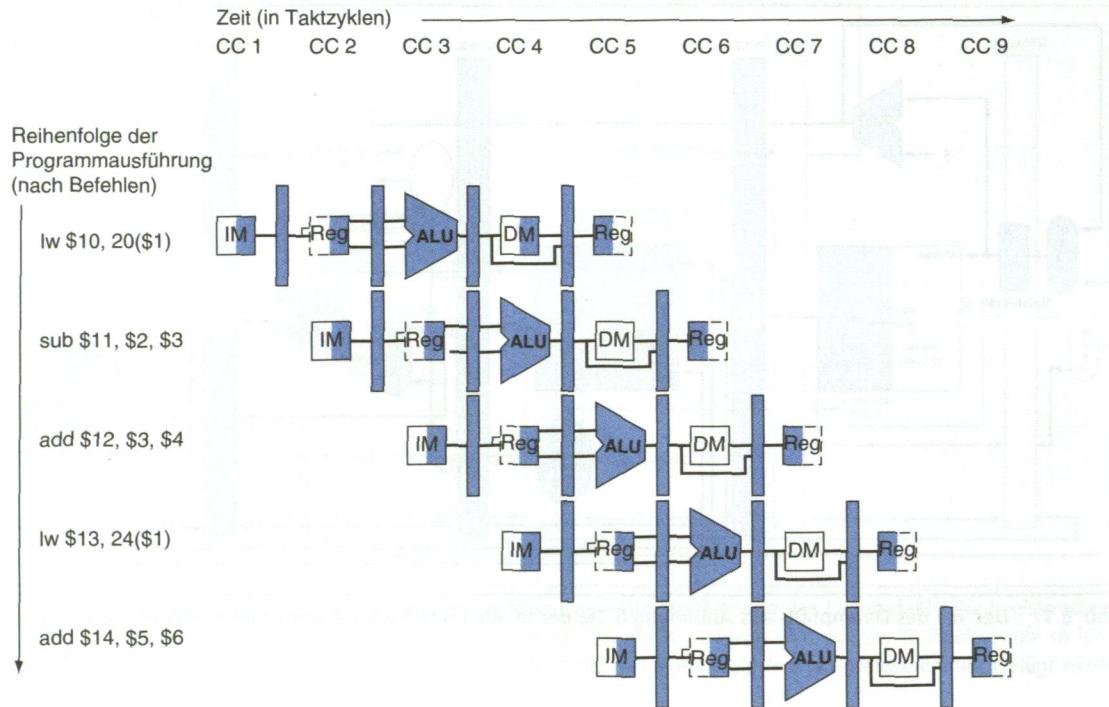


Abb. 6.18 Mehrzyklen-Pipelinediagramm von fünf Befehlen. Mit dieser Art der Pipelinedarstellung wird die vollständige Ausführung von Befehlen in einer einzigen Abbildung dargestellt. Befehle werden in der Reihenfolge der Befehlausführung von oben nach unten und Taktzyklen von links nach rechts dargestellt. Im Gegensatz zu Abbildung 6.3 befinden sich hier zwischen den einzelnen Stufen die Pipelineregister. In Abbildung 6.19 ist die herkömmliche Art der Darstellung dieses Diagramms zu sehen.

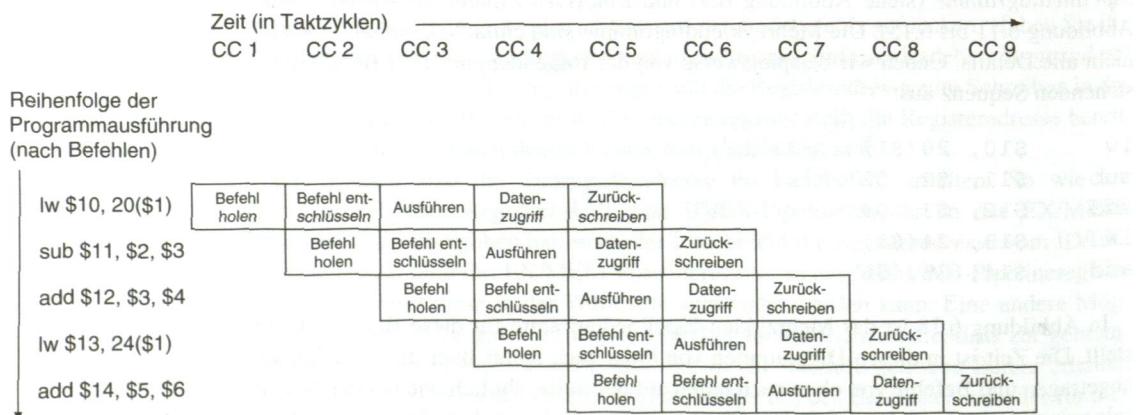


Abb. 6.19 Herkömmliches Mehrzyklen-Pipelinediagramm der fünf Befehle aus Abbildung 6.18.

Taktzyklen geschieht. Meist werden mehrere Zeichnungen dargestellt, um den Pipelinebetrieb über eine Folge von Taktzyklen hinweg zu veranschaulichen. Ein Einzyklen-Pipelinediagramm stellt einen vertikalen Schnitt durch ein Mehrzyklen-Pipelinediagramm dar und veranschaulicht, wie der Datenpfad von den einzelnen Befehlen in der Pipeline zu dem dargestellten Taktzyklus genutzt wird. In Abbildung 6.20 ist beispielsweise ein Einzyklen-Pipelinediagramm dargestellt, das dem Taktzyklus 5 aus Abbildung 6.18 und 6.19 entspricht. Das Einzyklen-Pipelinediagramm ist offensichtlich ausführlicher und benötigt zur Darstellung derselben Anzahl Taktzyklen

add \$14, \$5, \$6	lw \$13, 24 (\$1)	add \$12, \$3, \$4, \$11	sub \$11, \$2, \$3	lw\$10, 20(\$1)
Befehl holen	Befehl entschlüsseln	Ausführen	Speicher	Rückschreiben

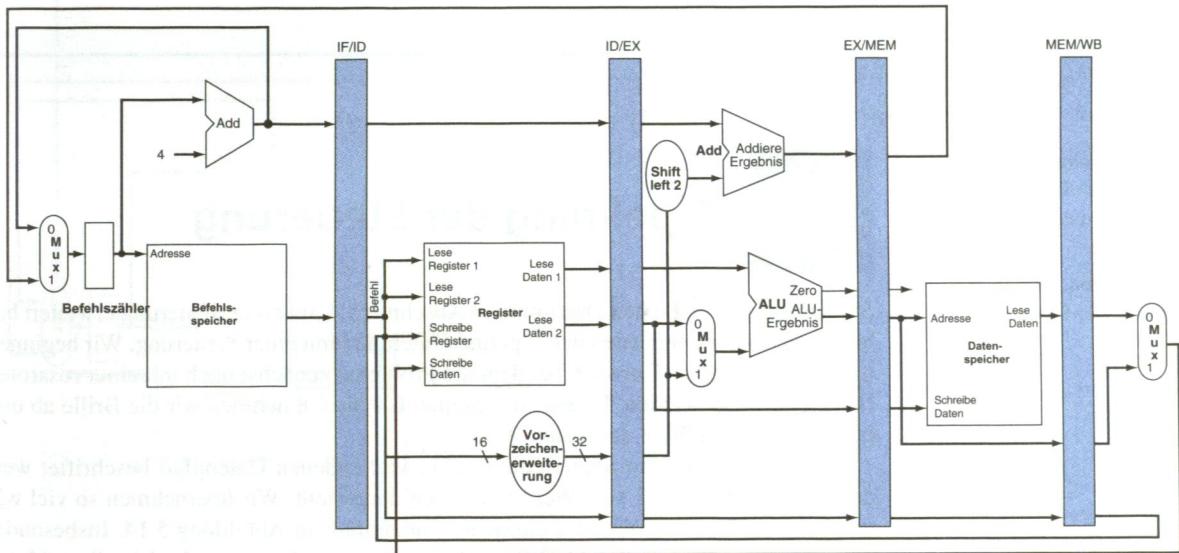


Abb. 6.20 Das Einzyklen-Pipelinediagramm, das dem Taktzyklus 5 aus Abbildung 6.18 und 6.19 entspricht. Wie Sie sehen, stellt ein Einzyklen-Pipelinediagramm einen vertikalen Schnitt durch ein Mehrzyklen-Pipelinediagramm dar.

deutlich mehr Platz. Im Abschnitt **For More Practice** auf der CD sind die entsprechenden Einzyklen-Diagramme für diese beiden Befehle dargestellt und Übungen zum Erstellen von Diagrammen dieser Art für eine andere Codesequenz enthalten.



Eine Gruppe Studenten diskutiert die Leistungsfähigkeit der fünfstufigen Pipeline. Einer der Studenten weist darauf hin, dass nicht alle Befehle in allen Stufen der Pipeline aktiv sind. Nachdem die Studenten beschlossen haben, die Auswirkungen von Pipeline-Konflikten außer Acht zu lassen, stellen sie die folgenden fünf Thesen auf. Welche davon stimmen?

1. Wenn zugelassen wird, dass Sprünge, Verzweigungen und ALU-Befehle weniger als die fünf vom Ladebefehl benötigten Stufen in Anspruch nehmen, wird die Leistungsfähigkeit der Pipeline auf jeden Fall verbessert.
2. Es bringt keinen Vorteil, wenn einige Befehle weniger Zyklen beanspruchen, da der Durchsatz durch den Taktzyklus bestimmt wird. Die Anzahl der Pipelinestufen pro Befehl wirkt sich auf die Pipeline-Latenz, nicht auf den Durchsatz aus.
3. Wenn zugelassen wird, dass Sprünge, Verzweigungen und ALU-Operationen weniger Zyklen beanspruchen, ist dies nur vorteilhaft, wenn sich keine Lade- oder Speicherbefehle in der Pipeline befinden. Der Vorteil ist also nur gering.
4. Aufgrund des Zurückschreibens des Befehls, ist es nicht möglich, dass ALU-Befehle weniger Zyklen beanspruchen, aber Verzweigungen und Sprünge können mit weniger Zyklen auskommen, so dass eine gewisse Chance für eine Optimierung besteht.



5. Anstatt zu versuchen, Befehlen weniger Taktzyklen zur Verfügung zu stellen, sollten wir die Pipeline verlängern, so dass Befehle mehr Zyklen benötigen, die jedoch kürzer sind. Damit ließe sich die Leistung steigern.
-

In the 6600 Computer, perhaps even more than in any previous computer, the control system is the difference.

James Thornton, *Design of a Computer: The Control Data 6600*, 1970

6.3

Pipelining der Steuerung

So, wie wir den einfachen Datenpfad in Abschnitt 5.4 mit einer Steuerung erweitert haben, erweitern wir nun auch den Pipeline-Datenpfad mit einer Steuerung. Wir beginnen mit einem einfachen Entwurf, bei dem die Probleme zunächst noch mit einer rosaroten Brille betrachtet werden. In den Abschnitten 6.4 bis 6.8 nehmen wir die Brille ab und erkennen die Konflikte der realen Welt.

Zunächst müssen die Steuerleitungen im vorhandenen Datenpfad beschriftet werden. In Abbildung 6.21 sind diese Leitungen dargestellt. Wir übernehmen so viel wie möglich aus der Steuerung des einfachen Datenpfads in Abbildung 5.14. Insbesondere verwenden wir dieselbe ALU-Steuerlogik, dieselbe Sprunglogik, denselben Multiplexer zum Bestimmen der Zielregisteradresse und dieselben Steuerleitungen. Diese Funktionen sind in den Tabellen 5.1, 5.3 und 5.4 definiert. In den Tabellen 6.2 bis 6.4 wiederholen wir die wichtigen Informationen, damit sich die folgende Beschreibung besser nachvollziehen lässt.

Wie bei der in Kapitel 5 beschriebenen Einzyklenausführung gehen wir davon aus, dass der Befehlszähler bei jedem Taktzyklus aktualisiert wird, so dass es für den Befehlszähler kein eigenes Schreibsignal gibt. Mit derselben Begründung gibt es auch keine eigenen Schreibsignale für die Pipelineregister (IF/ID, ID/EX, EX/MEM und MEM/WB), da in die Pipelineregister ebenfalls in jedem Taktzyklus geschrieben wird.

Um die Steuerung für die Pipeline zu definieren, müssen wir lediglich die Steuerwerte für die einzelnen Pipelinestufen festlegen. Da jede Steuerleitung einer Komponente zugeordnet ist, die während nur einer Pipelinestufe aktiv ist, können wir die Steuerleitungen entsprechend der Pipelinestufen in fünf Gruppen einteilen.

1. *Instruction fetch (Befehl holen):* Die Steuersignale zum Lesen des Befehlsspeichers und zum Schreiben in den Befehlszähler sind immer auf logisch 1 gesetzt. Somit gibt es in dieser Pipelinestufe nichts Spezielles zu steuern.
2. *Instruction decode/register file read (Befehl entschlüsseln/Register lesen):* Wie bei der vorherigen Stufe geschieht in jedem Taktzyklus dasselbe, so dass keine optionalen Steuerleitungen gesetzt werden müssen.
3. *Execution/address calculation (Befehl ausführen/Adresse berechnen):* Die Signale RegDst, ALUOp und ALUSrc müssen gesetzt werden (siehe Tabelle 6.2 und 6.3). Die Signale wählen das Ergebnisregister, die ALU-Operation und entweder den Lese-Daten-2-Wert oder einen vorzeichenerweiterten Immediate-Wert für die ALU aus.
4. *Memory access (Speicherzugriff):* In dieser Stufe werden die Steuerleitungen Branch, MemRead und MemWrite gesetzt. Diese Signale werden jeweils durch den Branch-equal-Befehl, den Ladebefehl und den Speicherbefehl gesetzt. PCSrc in Tabelle 6.3 wählt die folgende Adresse aus, außer die Steuerung setzt Branch auf logisch 1 und das ALU-Ergebnis war 0.

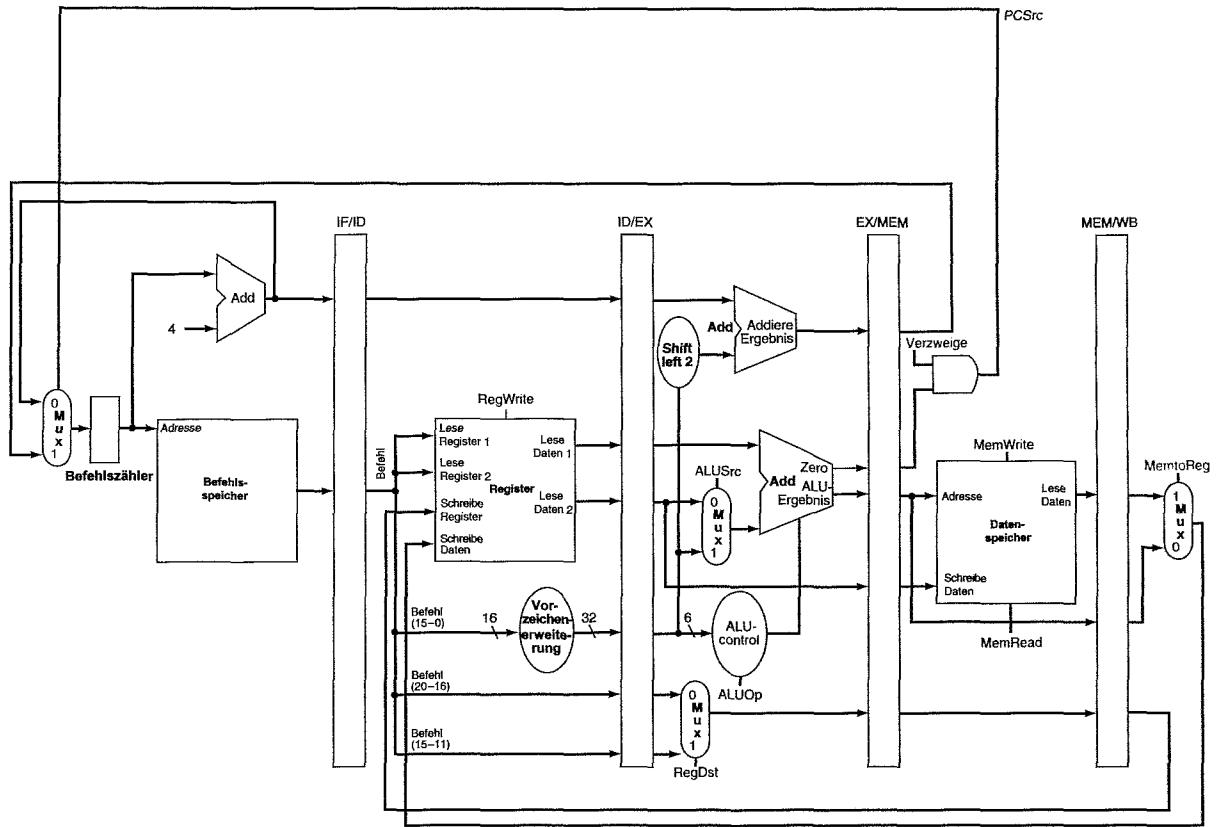


Abb. 6.21 Der Datenpfad mit Pipeline aus Abbildung 6.16 mit gekennzeichneten Steuersignalen. Dieser Datenpfad verwendet die Steuerlogik für Befehlszählerquelle, Registerzieladresse und ALU-Steuerung aus Kapitel 5. Dabei wird nun das 6-Bit-funct-Feld (Funktionscode) des Befehls in der EX-Stufe als Eingang für die ALU-Steuerung benötigt, so dass diese Bits ebenfalls im ID/EX-Pipelinerегистер enthalten sein müssen. Diese 6 Bits sind außerdem die 6 niedrigstwertigen Bits des Immediate-Felds im Befehl, so dass das ID/EX-Pipelinerегистer diese aus dem Immediate-Feld bereitstellen kann, da diese Bits durch die Vorzeichenerweiterung unverändert bleiben.

5. *Write back (Ergebnis rückschreiben):* Die Steuerleitung MemtoReg entscheidet, ob das ALU-Ergebnis oder der Speicherwert an den Registersatz gesendet wird, und die Steuerleitung RegWrite schreibt den so ausgewählten Wert.

Da die Bedeutung der Steuerleitungen durch die Ausstattung des Datenpfads mit einer Pipeline unverändert bleibt, können wir dieselben Steuerwerte wie zuvor verwenden. In Tabelle 6.4 sind dieselben Werte wie in Kapitel 5 angegeben, wobei die neun Steuerleitungen nun jedoch nach Pipelinestufen zusammengefasst sind.

Eine Steuerung implementieren bedeutet, für die neun Steuerleitungen in jeder Stufe für jeden Befehl Werte festzulegen. Dies lässt sich am leichtesten durch die Erweiterung der Pipelineregister um Steuerinformationen realisieren.

Da die Steuerleitungen mit der EX-Stufe beginnen, können wir die Steuerinformationen während der Befehlsentschlüsselung erzeugen. In Abbildung 6.22 ist dargestellt, dass diese Steuersignale anschließend in der entsprechenden Pipelinestufe verwendet werden, während der Befehl die Pipeline durchläuft, so wie die Zielregisteradresse für Ladebefehle in Abbildung 6.16 die Pipeline durchläuft. In Abbildung 6.23 ist der ganze Datenpfad mit den erweiterten Pipelineregistern dargestellt, wobei die Steuerleitungen den entsprechenden Stufen zugeordnet sind.

Tab. 6.2 Eine Kopie von Tabelle 5.1. Diese Abbildung zeigt, dass die ALU-Steuerbits in Abhängigkeit von den ALUOp-Steuerbits und den unterschiedlichen Funktionscodes für den R-Befehl gesetzt werden.

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

Tab. 6.3 Eine Kopie von Tabelle 5.3. Die Funktion der sieben Steuersignale ist definiert. Die ALU-Steuerleitungen (ALUOp) sind in der zweiten Spalte von Tabelle 6.2 definiert. Wenn die 1-Bit-Steuerleitung zum Zweifach-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. PCSrc wird in Abbildung 6.21 durch ein UND-Gatter (AND) gesteuert. Wenn sowohl das Sprungsignal (branch) als auch das ALU Zero-Signal gesetzt sind, ist PCSrc 1, andernfalls ist PCSrc 0. Die Steuerung setzt das Sprungsignal (branch) nur bei einem beq-Befehl, andernfalls wird PCSrc auf 0 gesetzt.

Signalname	Auswirkung, wenn logisch 0	Auswirkung, wenn logisch 1
RegDst	Die Registerzieladresse für den Schreibe-in-Register-Befehl wird vom rt-Feld (Bit 20:16) bereitgestellt.	Die Registerzieladresse für den Schreibe-in-Register-Befehl wird vom rd-Feld (Bit 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 Registerausgang (Lese Daten 2) bereitgestellt.	Der zweite ALU-Operand besteht aus den vorzeichenweiteren, unteren 16 Bit 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 der Register angelegte Wert wird von der ALU bereitgestellt.	Der am Schreibe-Daten-Eingang der Register angelegte Wert wird vom Datenspeicher bereitgestellt.

Tab. 6.4 Die Werte der Steuerleitungen sind dieselben wie in Tabelle 5.4. Sie wurden jedoch entsprechend der letzten drei Pipelinestufen in drei Gruppen unterteilt.

Befehl	Steuerleitungen der Stufe „Execution/address calculation“ (Ausführen/Adresse berechnen)				Branch	Steuerleitungen der Stufe „Memory access“ (Speicherzugriff)		Steuerleitungen der Stufe „Write-back“ (Ergebnis rückschreiben)	
	Reg Dst	ALU Op1	ALU Op0	ALU Src		Mem Read	Mem Write	Reg Write	Mem to Reg
R-Format	1	1	0	0	0	0	0	1	0
lw	0	0	0	1	0	1	0	1	1
sw	X	0	0	1	0	0	1	0	X
beq	X	0	1	0	1	0	0	0	X

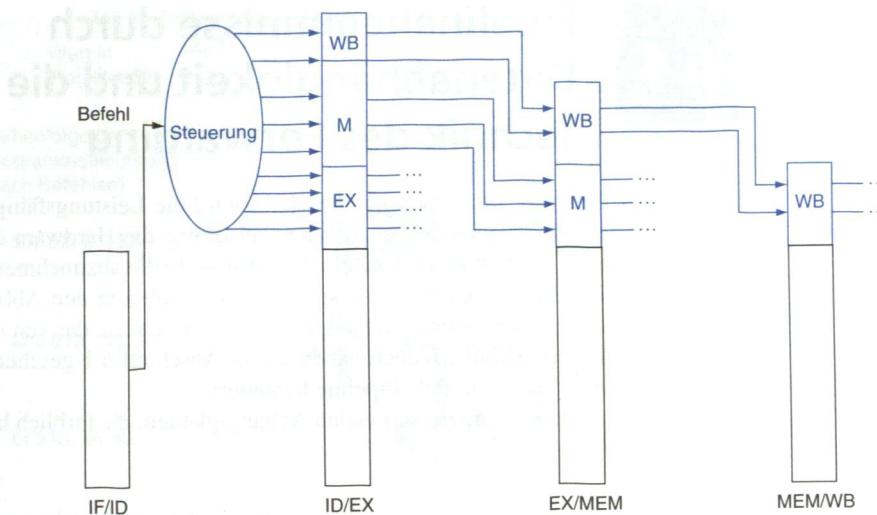


Abb. 6.22 Die Steuerleitungen für die letzten drei Stufen. Vier der neun Steuersignale werden in der EX-Phase verwendet, wobei die restlichen fünf Steuersignale an das EX/MEM-Pipelinerегистер übergeben werden, das erweitert wurde und nun die Steuersignale aufnimmt. Drei Steuersignale werden während der MEM-Stufe benötigt, und die letzten beiden werden an das MEM/WB-Register für die Verwendung in der WB-Stufe übergeben.

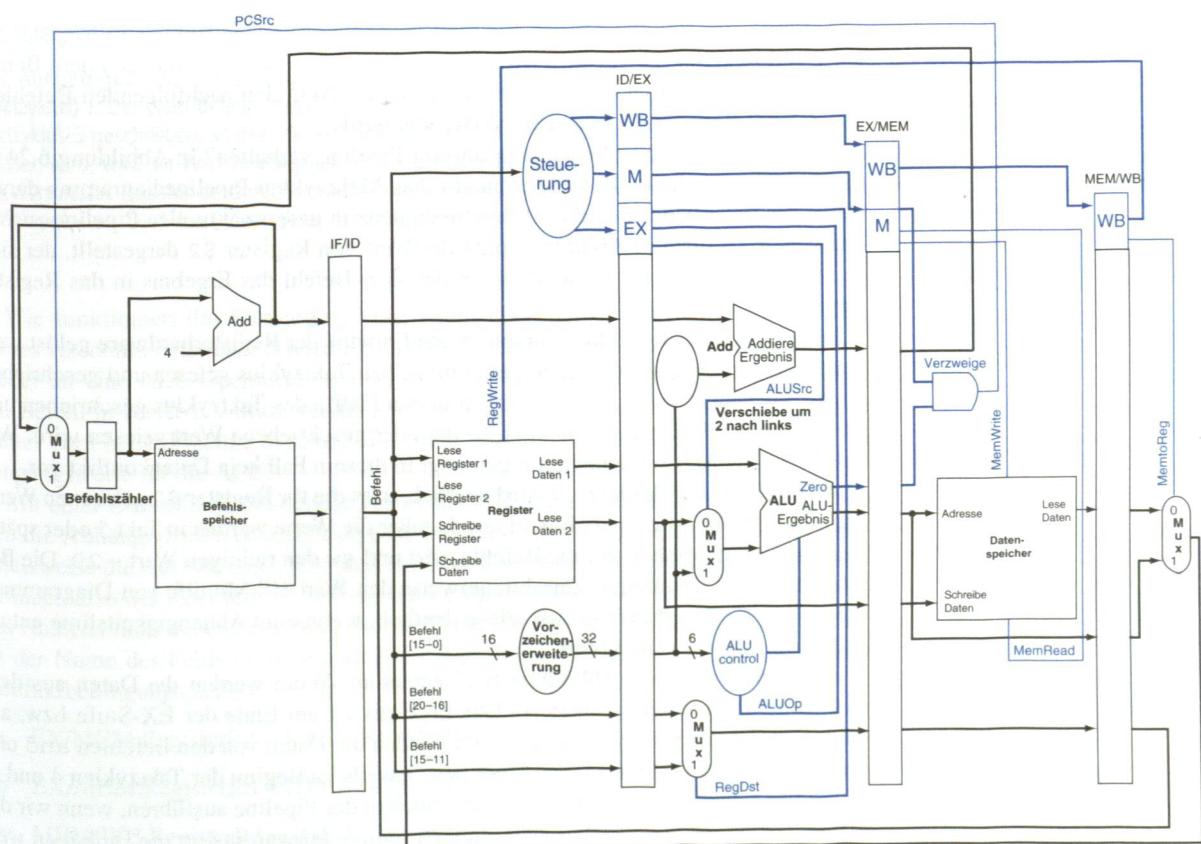


Abb. 6.23 Der Pipeline-Datenpfad aus Abbildung 6.21, wobei die Steuerleitungen mit den Steuerteilen der Pipelineregister verbunden sind. Die Steuerwerte für diese letzten drei Stufen werden während der Befehlsentschlüsselung erstellt und anschließend im ID/EX-Pipelinerегистер gespeichert. Die Steuersignale für die einzelnen Pipelinestufen werden verwendet, und die restlichen Steuersignale werden an die nächste Pipelinestufe übergeben.

*What do you mean, why's it got to be built? It's a bypass.
You've got to build bypasses.*

Douglas Adams, *Hitchhikers Guide to the Galaxy*, 1979

6.4

Pipelinehemmnisse durch Datenabhängigkeit und die Technik des Forwarding

Anhand der Beispiele im vorherigen Abschnitt wird die Leistungsfähigkeit der Ausführung mittels Pipeline sowie die Art und Weise, wie die Hardware diese Aufgabe meistert, deutlich. Nun ist es an der Zeit, die rosarote Brille abzunehmen und zu überlegen, was mit echten Programmen geschieht. Die Befehle in den Abbildungen 6.18 bis 6.20 waren nicht voneinander abhängig: Keiner verwendete das von einem anderen Befehl berechnete Ergebnis. Jedoch haben wir in Abschnitt 6.1 gesehen, dass Datenkonflikte die Ausführung mittels Pipeline behindern.

Betrachten wir eine Sequenz mit vielen Abhängigkeiten, die farblich hervorgehoben sind:

```
sub $2, $1,$3      # In Register $2 wird von sub geschrieben
and $12,$2,$5     # Erster Operand ($2) hängt von sub ab
or $13,$6,$2      # Zweiter Operand ($2) hängt von sub ab
add $14,$2,$2      # Erster und zweiter Operand
                   # (jeweils $2) hängen von sub ab
sw $15,100($2)    # Basis ($2) hängt von sub ab
```

Die letzten vier Befehle hängen alle vom Ergebnis des ersten Befehls in Register \$2 ab. Wenn Register \$2 vor dem sub-Befehl den Wert 10 enthält und nach dem Befehl den Wert -20, erwartet der Programmierer, dass -20 in den nachfolgenden Befehlen, die sich auf Register \$2 beziehen, verwendet wird.

Wie würde sich diese Sequenz in unserer Pipeline verhalten? In Abbildung 6.24 ist die Ausführung dieser Befehle anhand eines Mehrzyklen-Pipelinediagramms dargestellt. Um die Ausführung dieser Befehlssequenz in unserer aktuellen Pipeline zu verdeutlichen, ist oben in Abbildung 6.24 der Wert von Register \$2 dargestellt, der sich in der Mitte des Taktes 5 ändert, wenn der sub-Befehl das Ergebnis in das Register schreibt.

Ein potenzieller Konflikt kann durch den Entwurf der Registerhardware gelöst werden: Was geschieht, wenn ein Register im selben Taktzyklus gelesen und geschrieben wird? Wir gehen davon aus, dass in der ersten Hälfte des Taktzyklus geschrieben und in der zweiten Hälfte gelesen wird, so dass der geschriebene Wert gelesen wird. Wie bei vielen Registerimplementierungen liegt in diesem Fall kein Datenkonflikt vor.

Anhand von Abbildung 6.24 wird deutlich, dass die für Register \$2 gelesenen Werte *nicht* das Ergebnis des sub-Befehls sind, außer die Werte werden in Takt 5 oder später gelesen. Somit erhalten nur die Befehle add und sw den richtigen Wert -20. Die Befehle and und or erhalten fälschlicherweise den Wert 10! Mithilfe von Diagrammen dieser Art werden Probleme wie diese deutlich, wenn eine Abhängigkeitslinie entgegen der Zeitachse zurückführt.

Aber sehen Sie sich Abbildung 6.24 genau an: Wann werden die Daten aus dem sub-Befehl tatsächlich generiert? Das Ergebnis ist am Ende der EX-Stufe bzw. am Ende von Taktzyklus 3 verfügbar. Wann werden die Daten von den Befehlen and und or benötigt? Zu Beginn der EX-Stufe bzw. jeweils zu Beginn der Taktzyklen 4 und 5. Somit können wir dieses Segment ohne Anhalten der Pipeline ausführen, wenn wir die Daten, sobald diese verfügbar sind, einfach mittels Forwarding an die Einheiten weiterleiten, die die Daten benötigen, noch bevor diese zum Lesen aus dem Registersatz verfügbar sind.

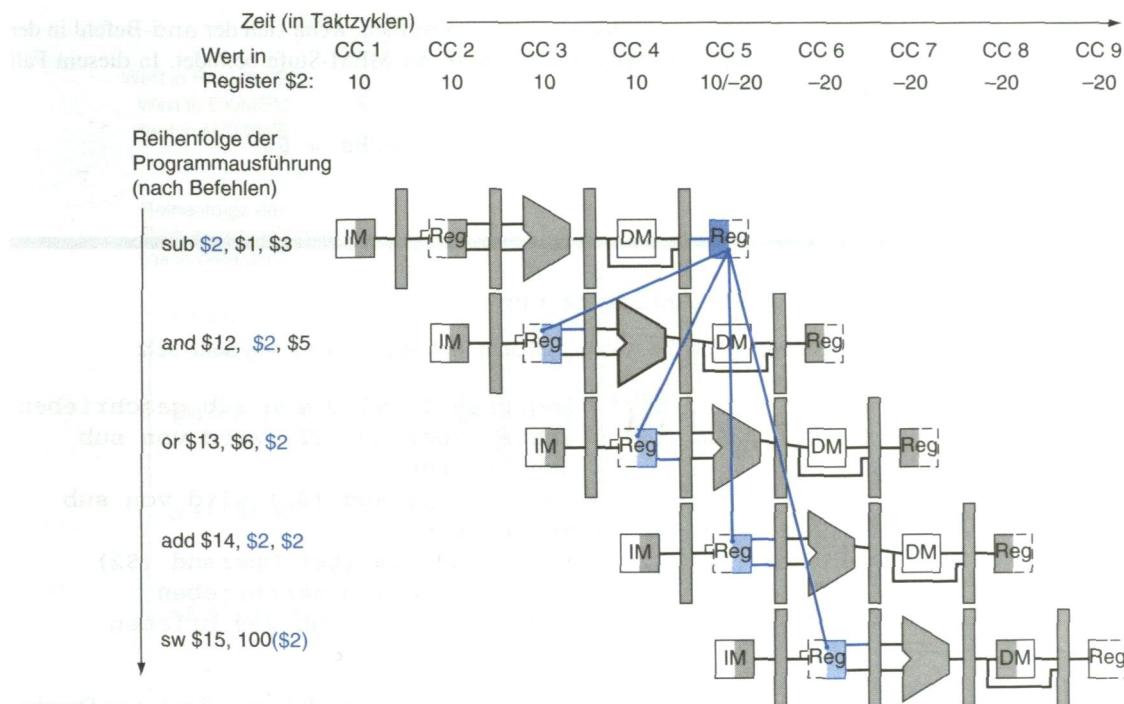


Abb. 6.24 Pipelinehemmnisse in einer Sequenz aus fünf Befehlen dargestellt mit vereinfachten Datenpfaden zur Verdeutlichung der Abhängigkeiten. Alle abhängigen Aktionen sind farblich hervorgehoben, und „CC i “ oben in der Abbildung steht für Clock Cycle (Taktzyklus) i . Der erste Befehl schreibt in das Register \$2, und alle nachfolgenden Befehle lesen Register \$2. In dieses Register wird in Taktzyklus 5 geschrieben, so dass der richtige Wert erst ab Taktzyklus 5 zur Verfügung steht. (Wenn ein Register während eines Taktzyklus gelesen wird, wird der Wert zurückgegeben, der am Ende der ersten Hälfte des Zyklus in das Register geschrieben wurde, sofern überhaupt ein Wert in das Register geschrieben wurde.) Die farbigen Linien vom oberen Datenpfad zu den unteren Datenpfaden veranschaulichen die Abhängigkeiten. Die Linien, die entlang der Zeitachse rückwärts führen, sind Pipelinehemmnisse durch Datenabhängigkeit.

Wie funktioniert das Forwarding? Der Einfachheit halber betrachten wir im Rest dieses Abschnitts nur das Forwarding an eine Verarbeitung in der EX-Stufe, also entweder an eine ALU-Operation oder an eine Berechnung der Effektivadresse. Wenn ein Befehl in seiner EX-Stufe versucht, ein Register zu verwenden, in das ein Befehl weiter vorn in der Pipeline in seiner WB-Stufe schreibt, benötigen wir diese Werte als Eingangswerte für die ALU.

Mit einer Darstellung, bei der die Felder der Pipelineregister benannt werden, können die Abhängigkeiten präziser dargestellt werden. So gibt „ID/EX.RegisterRs“ beispielsweise die Adresse eines Registers an, dessen Wert sich im Pipelineregister ID/EX befindet, also des Registers aus dem ersten Leseport des Registersatzes. Der erste Teil des Namens links neben dem Punkt ist der Name des Pipelineregisters, der zweite Teil ist der Name des Felds in diesem Register. Mit dieser Darstellung lauten die beiden Konfliktbedingungspaare wie folgt:

- 1a EX/MEM.RegisterRd = ID/EX.RegisterRs
- 1b EX/MEM.RegisterRd = ID/EX.RegisterRt
- 2a MEM/WB.RegisterRd = ID/EX.RegisterRs
- 2b MEM/WB.RegisterRd = ID/EX.RegisterRt

Der erste Konflikt in der Sequenz auf Seite 328 bezieht sich auf Register \$2 zwischen dem Ergebnis aus `sub $2, $1, $3` und dem ersten Leseoperanden aus `and`

$\$12, \$2, \$5$. Dieser Konflikt kann erkannt werden, wenn sich der and-Befehl in der EX-Stufe und der vorhergehende Befehl in der MEM-Stufe befindet. In diesem Fall liegt der Konflikt 1a vor:

$$\text{EX/MEM.RegisterRd} = \text{ID/EX.RegisterRs} = \$2$$

BEISPIEL

Erkennen von Abhängigkeiten

Klassifizieren Sie die Abhängigkeiten in dieser Sequenz von Seite 328:

```
sub $2, $1, $3 # Register $2 wird von sub geschrieben
and $12, $2, $5 # Erster Operand ($2) wird von sub
# geschrieben
or $13, $6, $2 # Zweiter Operand ($2) wird von sub
# geschrieben
add $14, $2, $2 # Erster und zweiter Operand ($2)
# werden von sub geschrieben
sw $15, 100($2) # Index wird von sub geschrieben
```

ANTWORT

Wie oben bereits erwähnt, liegt bei sub-and ein Konflikt vom Typ 1a vor. Daneben liegen noch folgende weitere Konflikte vor:

- Bei sub-or liegt ein Konflikt vom Typ 2b vor: $\text{MEM/WB.RegisterRd} = \text{ID/EX.RegisterRt} = \2
- Bei den beiden Abhängigkeiten von sub-add handelt es sich nicht um Konflikte, da der Registersatz die gewünschten Daten in der ID-Stufe des add-Befehls bereitstellt.
- Zwischen den Befehlen sub und sw liegt kein Konflikt vor, da sw das Register \$2 erst in dem Taktzyklus liest, *nachdem* sub in \$2 geschrieben hat.

Da einige Befehle nicht in Register schreiben, ist diese Vorgehensweise ungenau. Manchmal leitet sie mittels Forwarding weiter, wenn dies gar nicht notwendig ist. Dieses Problem kann gelöst werden, indem überprüft wird, ob das RegWrite-Signal aktiv ist: Beim Überprüfen des WB-Steuerfelds des Pipelineregisters während der EX- und der MEM-Stufe kann festgestellt werden, ob RegWrite auf logisch 1 gesetzt ist. Zudem erfordert MIPS, dass der Operand bei jeder Verwendung von \$0 den Wert Null annehmen muss. Wenn beispielsweise ein Befehl in der Pipeline \$0 zum Ziel hat (z.B. sll \$0, \$1, 2), soll das Ergebnis, das möglicherweise nicht null ist, nicht weitergeleitet werden. Wenn Ergebnisse, die für \$0 bestimmt sind, nicht weitergeleitet werden, kann man Festlegungen vermeiden, dass Assembler-Programmierer und Compiler \$0 nicht als Ziel verwenden dürfen. Die obigen Bedingungen funktionieren also einwandfrei, wenn wir $\text{EX/MEM.RegisterRd} \neq 0$ in die erste Konfliktbedingung und $\text{MEM/WB.RegisterRd} \neq 0$ in die zweite aufnehmen.

Nun, da wir Konflikte erkennen können, ist das Problem bereits halb gelöst. Aber wir müssen nach wie vor die richtigen Daten weiterleiten.

In Abbildung 6.25 sind die Abhängigkeiten zwischen den Pipelineregistern und den Eingängen an der ALU für dieselbe Codesequenz wie in Abbildung 6.24 dargestellt.

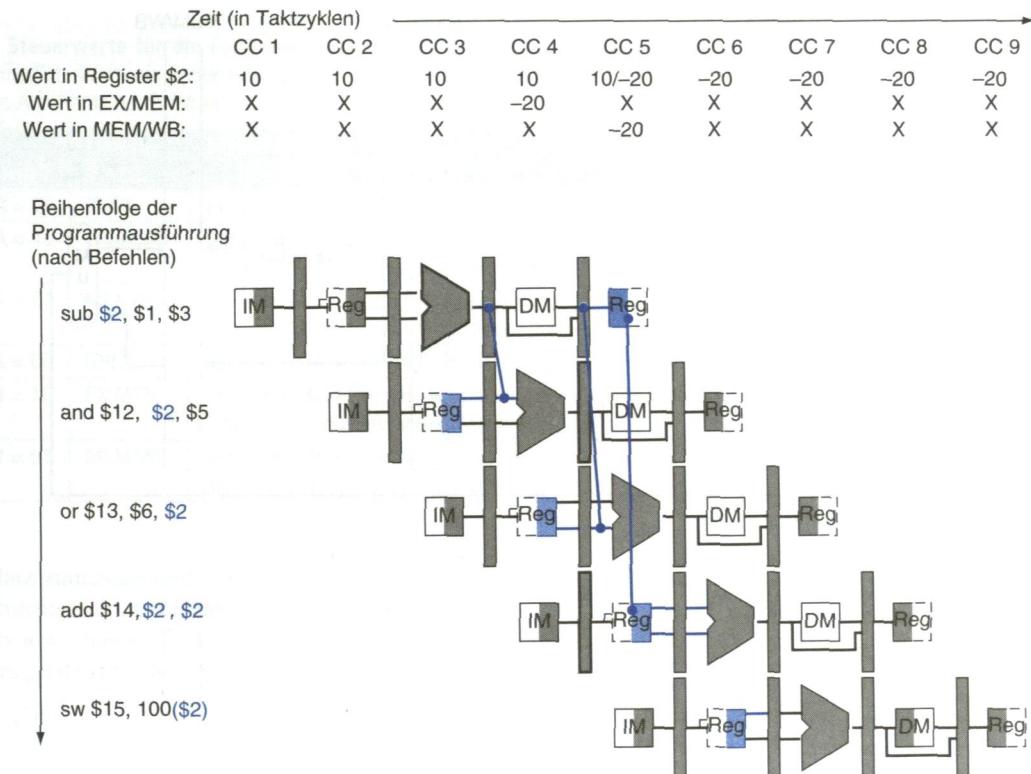
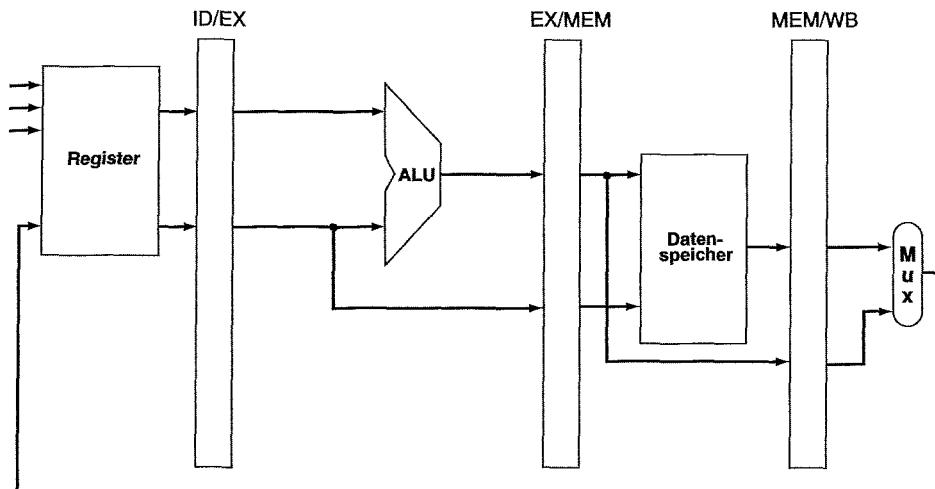


Abb. 6.25 Die Abhängigkeiten zwischen den Pipelineregistern verlaufen alle in Richtung der Zeitachse. Somit können die vom and-Befehl und vom or-Befehl benötigten Eingangswerte durch Weiterleiten der Ergebnisse in den Pipelineregistern mittels Forwarding für die ALU bereitgestellt werden. Die Werte in den Pipelineregistern zeigen, dass der gewünschte Wert verfügbar ist, bevor er in den Registersatz geschrieben wird. Wir gehen davon aus, dass der Registersatz Werte weiterleitet, die im selben Taktzyklus gelesen und geschrieben werden, so dass der add-Befehl nicht angehalten wird, die Werte stammen jedoch nicht von einem Pipelineregister, sondern aus dem Registersatz. Wegen des „Forwarding“ des Registersatzes (d.h. der Lesebefehl erhält den Wert des Schreibbefehls im selben Taktzyklus) befindet sich in Register \$2 zu Beginn von Taktzyklus 5 der Wert 10 und am Ende des Taktzyklus der Wert -20. Wie im Rest dieses Abschnitts werden alle Werte, außer dem Wert, der durch einen Speicherbefehl gespeichert werden soll, mittels Forwarding weitergeleitet.

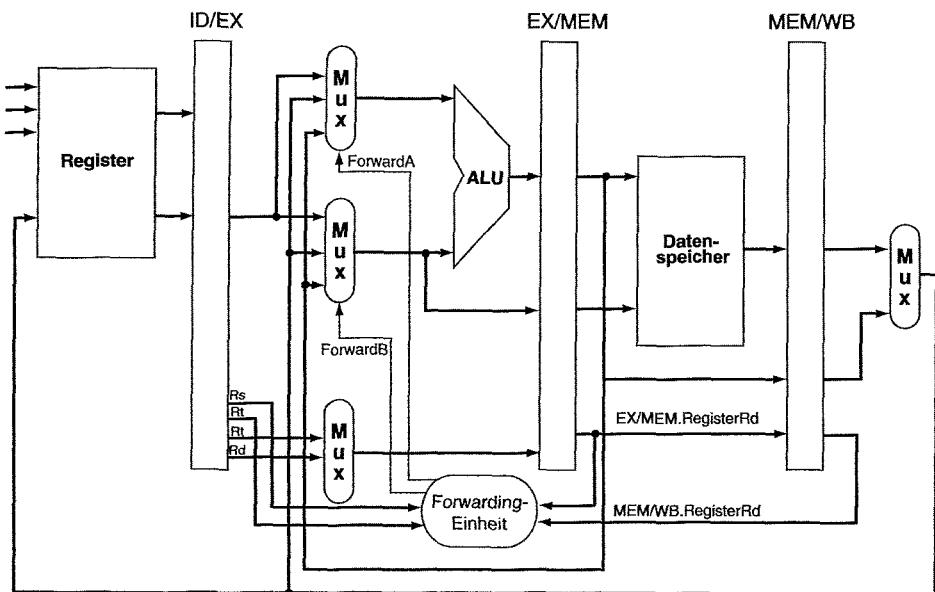
Der Unterschied besteht darin, dass die Abhängigkeit nun von einem *Pipelineregister* aus beginnt und nicht darauf gewartet wird, bis in der WB-Stufe in den Registersatz geschrieben wird. Somit sind die erforderlichen Daten rechtzeitig für spätere Befehle verfügbar, wobei die Daten für das Forwarding in den Pipelineregistern gespeichert werden.

Wenn wir die Eingangssignale für die ALU nicht nur aus dem ID/EX-Pipelinerегистер, sondern aus jedem *beliebigen* Pipelineregister verwenden können, können wir die gewünschten Daten mittels Forwarding weiterleiten. Durch Einfügen von Multiplexern in die Eingangsleitungen der ALU und mit den entsprechenden Steuersignalen können wir die Pipeline trotz dieser Datenabhängigkeiten mit maximaler Geschwindigkeit betreiben.

Im Moment wollen wir davon ausgehen, dass Forwarding nur für die vier R-Befehle erforderlich ist: add, sub, and und or. In Abbildung 6.26 ist eine Nahaufnahme der ALU und der Pipelineregister vor und nach dem Einfügen einer Forwarding-Einheit dargestellt. In Tabelle 6.5 sind die Werte der Steuerleitungen für die ALU-Multiplexer angegeben, die entweder die Registersatzwerte oder einen der mittels Forwarding weitergeleiteten Werte auswählen.



a. Ohne Forwarding



b. Mit Forwarding

Abb. 6.26 In der oberen Hälfte sind ALU und Pipelineregister vor dem Einfügen einer Forwarding-Einheit dargestellt. In der unteren Hälfte wurden die Multiplexer um die Forwarding-Leitungen erweitert und die Forwarding-Einheit ist eingefügt. Die neue Hardware ist farblich hervorgehoben. Bei dieser Abbildung handelt es sich um eine vereinfachte Darstellung ohne die Details des vollständigen Datenpfads wie etwa der Hardware für die Vorzeichenerweiterung. Das ID/EX.RegisterRt-Feld ist zweimal vorhanden: einmal als Verbindung zum MUX und einmal als Verbindung zur Forwarding-Einheit. Es handelt sich jedoch nur um ein Signal. Wie bereits erläutert wird hierbei die Weiterleitung eines Speicherwerts mittels Forwarding an einen Speicherbefehl ignoriert.

Diese Forwarding-Steuerung erfolgt in der EX-Stufe, da sich die Forwarding-Multiplexer der ALU in dieser Stufe befinden. Somit müssen wir die Registeradressen der Operanden aus der ID-Stufe über das ID/EX-Pipelineregister weitergeben, um festzustellen, ob Werte mittels Forwarding weitergeleitet werden sollen. Das rt-Feld (Bit 20-16) haben wir bereits. Vor dem Forwarding benötigte das ID/EX-Register

Tab. 6.5 Steuerwerte für die Forwarding-Multiplexer in Abbildung 6.26. Der vorzeichenbehaftete Immediate-Wert, ein weiteres Eingangssignal der ALU, wird im Abschnitt „Vertiefung“ am Ende dieses Abschnitts beschrieben.

MUX-Steuerung	Quelle	Erläuterung
ForwardA = 00	ID/EX	Der erste ALU-Operand wird vom Registersatz bereitgestellt.
ForwardA = 10	EX/MEM	Der erste ALU-Operand wird aus dem vorhergehenden ALU-Ergebnis mittels Forwarding weitergeleitet.
ForwardA = 01	MEM/WB	Der erste ALU-Operand wird aus dem Datenspeicher oder einem früheren ALU-Ergebnis mittels Forwarding weitergeleitet.
ForwardB = 00	ID/EX	Der zweite ALU-Operand wird vom Registersatz bereitgestellt.
ForwardB = 10	EX/MEM	Der zweite ALU-Operand wird aus dem vorhergehenden ALU-Ergebnis mittels Forwarding weitergeleitet.
ForwardB = 01	MEM/WB	Der zweite ALU-Operand wird aus dem Datenspeicher oder einem früheren ALU-Ergebnis mittels Forwarding weitergeleitet.

keinen Platz zum Speichern des rs-Felds. Also wird nun rs (Bit 25-21) zum ID/EX-Register hinzugefügt.

Schreiben wir nun sowohl die Bedingungen zum Erkennen von Konflikten als auch die Steuersignale zum Beheben der Konflikte auf:

1. EX-Konflikt:

```
if (EX/MEM.RegWrite
and (EX/MEM.RegisterRd ≠ 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRs)) ForwardA = 10
if (EX/MEM.RegWrite
and (EX/MEM.RegisterRd ≠ 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRt)) ForwardB = 10
```

In diesem Fall wird das Ergebnis aus dem vorhergehenden Befehl mittels Forwarding an einen der Eingänge der ALU weitergeleitet. Wenn mit dem vorhergehenden Befehl Werte in den Registersatz geschrieben werden und die Adresse des Registers, in das geschrieben wird, mit der Adresse des Registers, aus dem gelesen wird, der ALU-Eingänge A oder B übereinstimmt und es sich nicht um Register 0 handelt, soll der Multiplexer den Wert und nicht den Inhalt des EX/MEM-Pipelinerregisters auswählen.

2. MEM-Konflikt:

```
if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd + 0)
and (MEM/WB.RegisterRd = ID/EX.RegisterRs)) ForwardA = 01
if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd + 0)
and (MEM/WB.RegisterRd = ID/EX.RegisterRt)) ForwardB = 01
```

Wie bereits erwähnt, liegt in der WB-Stufe kein Konflikt vor, da wir davon ausgehen, dass der Registersatz das richtige Ergebnis bereitstellt, wenn der Befehl in der ID-Stufe dasselbe Register liest, in das der Befehl schreibt, der sich in der WB-Stufe befindet. Ein Registersatz dieser Art führt eine weitere Form des Forwarding aus, die jedoch innerhalb des Registersatzes auftritt.

Eine Schwierigkeit stellen potenzielle Datenkonflikte zwischen dem Ergebnis des Befehls in der WB-Stufe, dem Ergebnis des Befehls in der MEM-Stufe und dem Quelloperanden des Befehls in der ALU-Stufe dar. Bei der Addition eines Vektors von Zahlen in einem einzigen Register muss dasselbe Register von einer Folge von Befehlen gelesen und geschrieben werden:

```
add $1,$1,$2
add $1,$1,$3
add $1,$1,$4
```

In diesem Fall wird das Ergebnis mittels Forwarding aus der MEM-Stufe weitergeleitet, da das Ergebnis in der MEM-Stufe das neuere Ergebnis ist. Somit lautet die Steuerung für den MEM-Konflikt wie folgt (wobei die Ergänzungen farblich hervorgehoben sind):

```
if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd ≠ 0)
and (EX/MEM.RegisterRd ≠ ID/EX.RegisterRs)
and (MEM/WB.RegisterRd = ID/EX.RegisterRs)) ForwardA = 01
if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd ≠ 0)
and (EX/MEM.RegisterRd ≠ ID/EX.RegisterRt)
and (MEM/WB.RegisterRd = ID/EX.RegisterRt)) ForwardB = 01
```

In Abbildung 6.27 ist die Hardware dargestellt, die erforderlich ist, um das Forwarding für Operationen zu unterstützen, die während der EX-Stufe Ergebnisse verwenden.

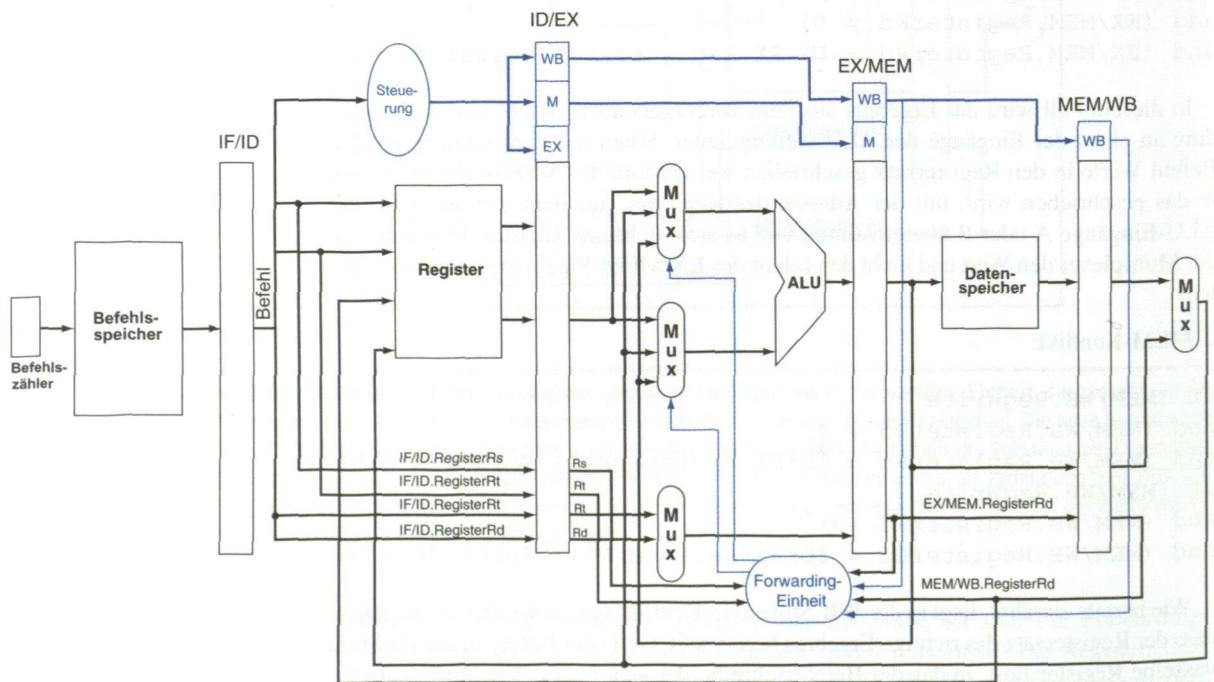


Abb. 6.27 Der Datenpfad, der so geändert wurde, dass Konflikte mittels Forwarding behoben werden. Verglichen mit dem Datenpfad in Abbildung 6.23 wurden die Multiplexer in die Eingangsleitungen zur ALU eingefügt. Bei dieser Abbildung handelt es sich um eine vereinfachte Darstellung ohne die Details des vollständigen Datenpfads wie etwa die Hardware für Sprünge und für die Vorzeichenerweiterung.

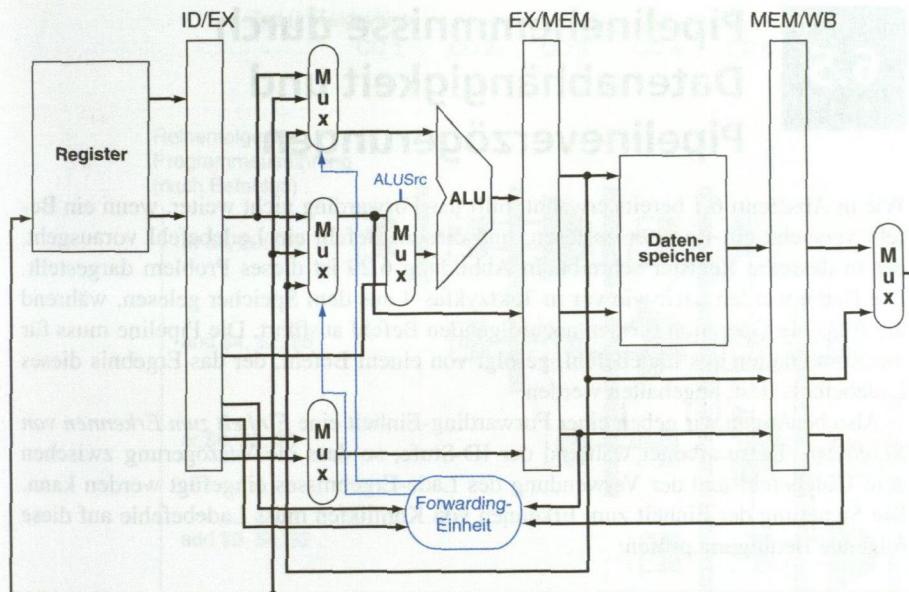


Abb. 6.28 Eine Nahaufnahme des Datenpfads aus Abbildung 6.26, die einen 2:1-Multiplexer darstellt, der ergänzt wurde, um das vorzeichenbehaftete Immediate-Signal als ALU-Eingangssignal auszuwählen.

Vertiefung: Das Forwarding kann auch bei Konflikten hilfreich sein, bei denen Speicherbefehle von anderen Befehlen abhängen. Da diese während der MEM-Stufe nur einen Datenwert verwenden, ist das Forwarding einfach. Aber wie verhält es sich mit Ladebefehlen, denen unmittelbar Speicherbefehle folgen. Wir müssen mehr Forwarding-Hardware verwenden, um das Kopieren von Speicher zu Speicher zu beschleunigen. Wenn wir Abbildung 6.20 neu zeichnen und dabei die Befehle `sub` und `and` durch `lw` und `sw` ersetzen, können wir feststellen, dass sich eine Pipelineverzögerung verhindern lässt, da die Daten im MEM/WB-Register eines Ladebefehls so rechtzeitig vorliegen, dass sie in der MEM-Stufe eines Speicherbefehls verwendet werden können. Dazu müssten wir die MEM-Stufe mit Forwarding-Hardware ergänzen. Wir überlassen Ihnen diese Änderung als Übung.

Daneben fehlt dem Datenpfad in Abbildung 6.27 das vorzeichenbehaftete Immediate-Eingangssignal an der ALU, das von Lade- und Speicherbefehlen benötigt wird. Da die zentrale Steuerung zwischen Registerwert und Immediate-Wert entscheidet und da die Forwarding-Einheit das Pipelineregister für ein Registereingangssignal an der ALU auswählt, ist es am einfachsten, einen 2:1-Multiplexer einzufügen, der zwischen dem ForwardB-Multiplexerausgang und dem vorzeichenbehafteten Immediate-Signal auswählt. In Abbildung 6.28 ist diese Erweiterung dargestellt. Diese Lösung weicht von dem ab, was wir in Kapitel 5 gelernt haben, wo der durch Leitung `ALUSrcB` gesteuerte Multiplexer um das Immediate-Eingangssignal erweitert wurde.



If at first you don't succeed,
redefine success.

Anonym

6.5

Pipelinehemmnisse durch Datenabhängigkeit und Pipelineverzögerungen

Wie in Abschnitt 6.1 bereits erwähnt, hilft das Forwarding nicht weiter, wenn ein Befehl versucht, ein Register zu lesen, und diesem Befehl ein Ladebefehl vorausgeht, der in dasselbe Register schreibt. In Abbildung 6.29 ist dieses Problem dargestellt. Die Daten werden nach wie vor in Taktzyklus 4 aus dem Speicher gelesen, während die ALU die Operation für den nachfolgenden Befehl ausführt. Die Pipeline muss für die Kombination aus Ladebefehl, gefolgt von einem Befehl, der das Ergebnis dieses Ladebefehls liest, angehalten werden.

Also benötigen wir neben einer Forwarding-Einheit eine *Einheit zum Erkennen von Konflikten*. Diese arbeitet während der ID-Stufe, so dass die Verzögerung zwischen dem Ladebefehl und der Verwendung des Lade-Ergebnisses eingefügt werden kann. Die Steuerung der Einheit zum Erkennen von Konflikten muss Ladebefehle auf diese folgende Bedingung prüfen:

```
if (ID/EX.MemRead and
    ((ID/EX.RegisterRt = IF/ID.RegisterRs) or
     (ID/EX.RegisterRt = IF/ID.RegisterRt)))
    stall the pipeline
```

Die erste Zeile prüft, ob es sich bei dem Befehl um einen Ladebefehl handelt: Der einzige Befehl, der den Datenspeicher liest, ist ein Ladebefehl. Die nächsten beiden Zeilen prüfen, ob das Zielregisterfeld des Ladebefehls in der EX-Stufe mit einem Quellregister des Befehls in der ID-Stufe übereinstimmt. Wenn die Bedingung erfüllt ist, hält der Befehl die Pipeline einen Taktzyklus lang an. Nach dieser Verzögerung kann die Abhängigkeit von der Forwarding-Logik behandelt werden, und die Ausführung wird fortgesetzt. (Ohne Forwarding würden die Befehle in Abbildung 6.29 einen weiteren Verzögerungszyklus benötigen.)

Wenn der Befehl in der ID-Stufe angehalten wird, muss auch der Befehl in der IF-Stufe angehalten werden. Wenn dies nicht geschieht, geht der geholte Befehl verloren. Dass diese beiden Befehle in der Pipelinebearbeitung voranschreiten, wird einfach dadurch verhindert, dass man Befehlszähler und IF/ID-Pipelinerегистер davon abhält, neue Werte anzunehmen. Vorausgesetzt, diese Register bleiben erhalten, wird das Lesen des Befehls in der IF-Stufe unter Verwendung desselben Befehlszählers fortgesetzt, und das Lesen der Register in der ID-Stufe wird unter Verwendung derselben Befehlsfelder im IF/ID-Pipelinerегистер fortgesetzt. Wenn wir dies auf unser Beispiel mit dem Wäschewaschen übertragen, ist das so, als würden Sie die Waschmaschine mit derselben Wäscheladung neu starten und als würde der Trockner währenddessen ohne Ladung trocknen. Der hintere Teil der Pipeline, der mit der EX-Stufe beginnt, muss ebenso wie der Trockner auch etwas tun. Dieser Teil führt also so genannte **NOP-Befehle** aus, d.h. Befehle, die keine Auswirkungen haben.

Wie können wir diese NOP-Befehle, die sich wie Bubbles verhalten, in die Pipeline einfügen? In Tabelle 6.4 ist zu sehen, dass ein NOP-Befehl – der „nichts tut“ – generiert wird, wenn alle neun Steuersignale in den Stufen EX, MEM und WB auf logisch 0 gesetzt werden. Wenn also der Konflikt in der ID-Stufe erkannt wird, können wir ein Bubble in die Pipeline einfügen, indem wir die Steuerfelder der Stufen EX, MEM und WB des ID/EX-Pipelinerегистers auf 0 setzen. Diese hilfreichen Steuerwerte werden bei jedem Taktzyklus mit dem entsprechenden Effekt weitergeleitet: Kein Register oder Speicher wird beschrieben, wenn die Steuerwerte alle 0 sind.

NOP-Befehl Ein Befehl, der eine Operation ausführt, die zu keiner Zustandsänderung führt.

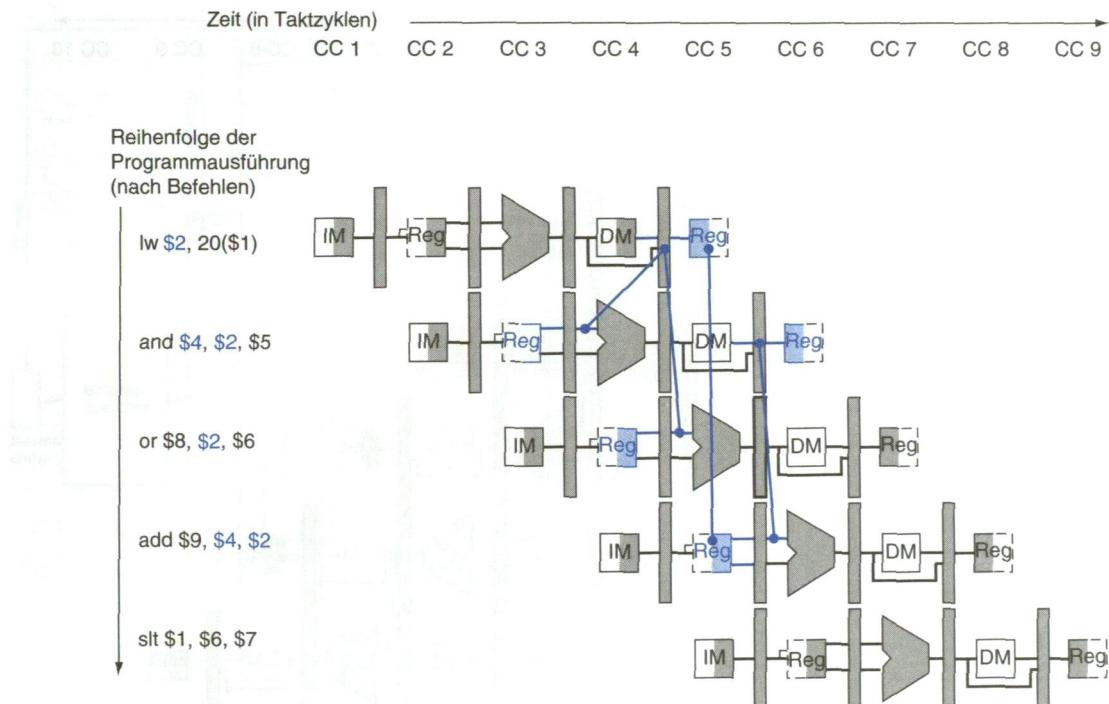


Abb. 6.29 Eine Befehlssequenz in der Pipeline. Da die Abhängigkeit zwischen dem Ladebefehl und dem nachfolgenden Befehl (and) einen zeitlichen Rücksprung erfordert, kann dieser Konflikt nicht durch Forwarding aufgelöst werden. Somit muss diese Befehlssequenz zu einer Verzögerung durch die Einheit zum Erkennen von Konflikten führen.

10. Schritt: Der Befehl and \$4, \$2, \$5 wird abgearbeitet. Der Befehl or \$8, \$2, \$6 ist noch nicht abgearbeitet.

In Abbildung 6.30 wird dargestellt, was in der Hardware tatsächlich geschieht: Die Pipelineausführungsstufe, die dem and-Befehl zugeordnet ist, wird in einen NOP-Befehl geändert und alle Befehle, nach dem and-Befehl, werden um einen Zyklus verzögert. Aufgrund des Konflikts müssen die and- und or-Befehle in Taktzyklus 4 wiederholen, was sie in Taktzyklus 3 bereits getan haben: and liest Register und wird entschlüsselt und or wird erneut aus dem Befehlsspeicher geholt. Wiederholte Arbeitsschritte fallen bei der Verzögerung an, aber eigentlich geht es darum, die Zeitdauer der and- und or-Befehle zu verlängern und das Holen des add-Befehls zu verzögern. Wie eine Luftblase in einer Wasserleitung verzögert ein Bubble alles ihm Nachfolgende und durchläuft die Befehlspipeline, eine Stufe pro Zyklus, bis er die Pipeline am Ende verlässt.

In Abbildung 6.31 sind die Pipelineverbindungen sowohl für die Einheit zum Erkennen von Konflikten als auch für die Forwarding-Einheit dargestellt. Wie zuvor steuert auch hier die Forwarding-Einheit die ALU-Multiplexer, um den Wert aus einem Allzweckregister gegen den Wert aus dem gewünschten Pipelineregister auszutauschen. Die Einheit zum Erkennen von Konflikten steuert das Schreiben in den Befehlszähler und das IF/ID-Register sowie den Multiplexer, der zwischen den echten Steuerwerten und der Möglichkeit, alle Signale auf 0 zu setzen, wählt. Die Einheit zum Erkennen von Konflikten hält die Pipeline an und setzt die Steuerfelder auf logisch 0, wenn die Überprüfung auf einen Load-use-Konflikt positiv ausfällt. Die Eintaktdiagramme finden Sie im Abschnitt **For More Practice** auf der CD.



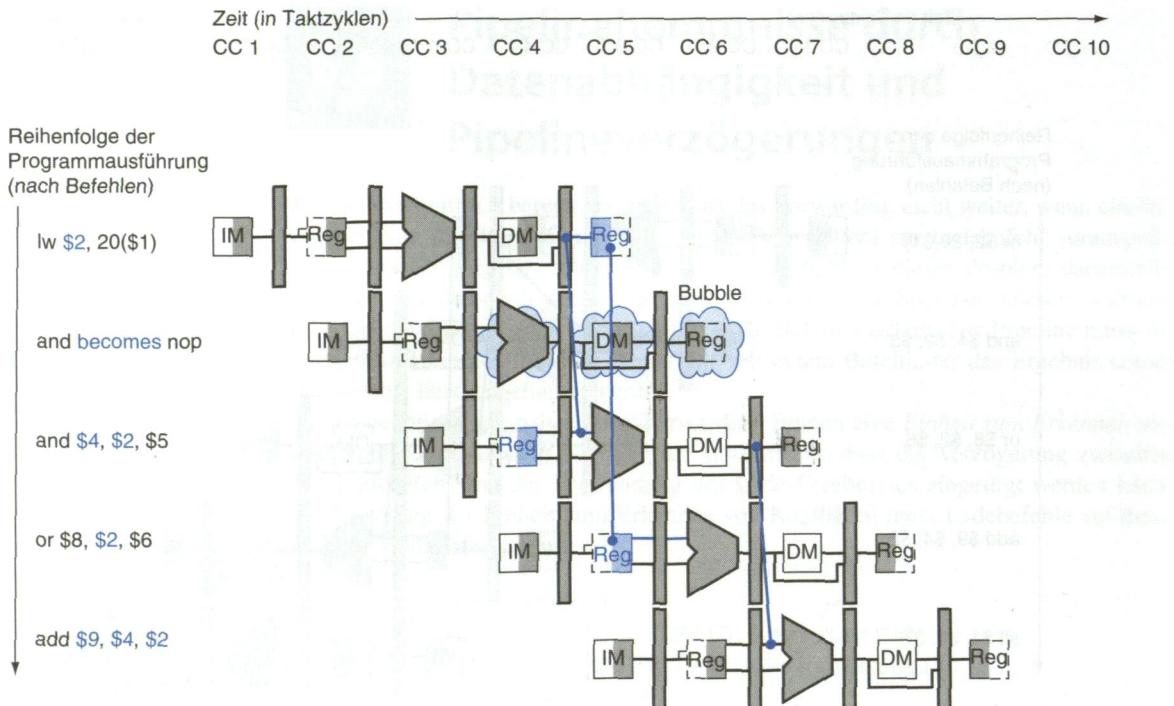


Abb. 6.30 So werden Verzögerungen in die Pipeline eingefügt. Eine Verzögerung (Bubble) wird beginnend in Taktzyklus 4 eingefügt, indem der `and`-Befehl in einen NOP-Befehl geändert wird. Der `and`-Befehl wird in den Taktzyklen 2 und 3 geholt und entschlüsselt, aber seine EX-Stufe wird bis Taktzyklus 5 angehalten (im Gegensatz zur unverzögerten Position in Taktzyklus 4). Entsprechend wird der `or`-Befehl in Taktzyklus 3 geholt, jedoch wird dessen IF-Stufe bis Taktzyklus 5 angehalten (im Gegensatz zur unverzögerten Position in Taktzyklus 4). Nach dem Einfügen des Bubbles sind alle Abhängigkeiten in Richtung der Zeitachse orientiert, und es treten keine weiteren Konflikte mehr auf.



Auch wenn die Hardware sich darum kümmert, Konflikte aufgrund von Abhängigkeiten aufzulösen und dadurch die gewünschte Ausführung gesichert ist, sollte der Compiler die Pipeline verstehen, damit die bestmögliche Leistung erzielt werden kann. Andernfalls wird die Leistung durch unerwartete Verzögerungen des kompilierten Codes beeinträchtigt.



Vertiefung: Eine Anmerkung zu der Bemerkung weiter oben, dass alle Steuersignale auf 0 gesetzt werden, um zu verhindern, dass Register oder Speicher beschrieben werden: Tatsächlich müssen nur die Signale RegWrite und MemWrite auf 0 gesetzt werden. Die anderen Steuersignale können auf „don't care“ gesetzt werden.

There are a thousand hacking at the branches of evil to one who is striking at the root.

Henry David Thoreau, *Walden*,
1854

6.6

Pipelinehemmnisse durch Kontrollflussabhängigkeiten

Bisher haben wir nur Konflikte bei arithmetischen Operationen und Datentransfers betrachtet. Wie wir jedoch in Abschnitt 6.1 gesehen haben, gibt es auch Pipelinehemmnisse bei Sprüngen. In Abbildung 6.32 ist eine Folge von Befehlen dargestellt, und es ist angegeben, an welcher Stelle in dieser Pipeline der Sprung stattfindet. In jedem

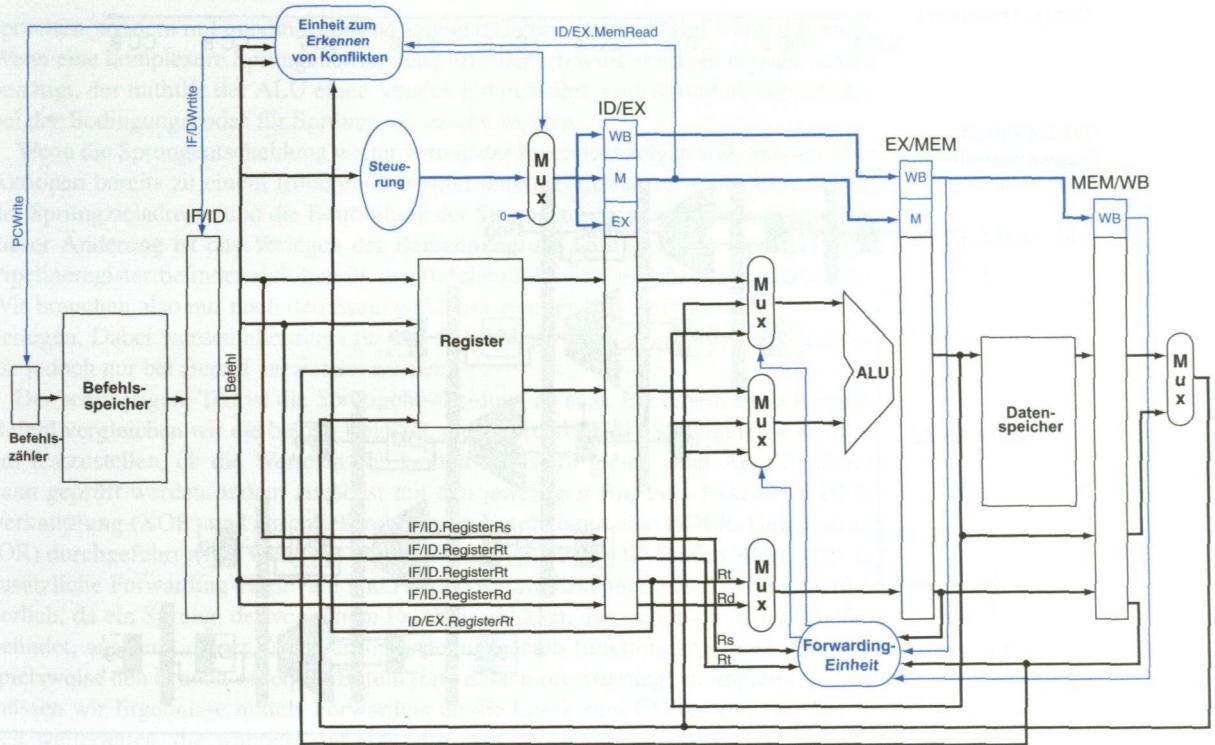


Abb. 6.31 Übersicht über die Steuerung mit Pipeline, mit den beiden Multiplexern für das Forwarding, mit der Einheit zum Erkennen von Konflikten und mit der Forwarding-Einheit. Die ID- und EX-Stufen sind zwar vereinfacht dargestellt (die Logik für vorzeichenverweiterte immediates und Sprünge fehlt), dennoch stellt diese Abbildung die wesentlichen Anforderungen an eine Forwarding-Hardware dar.

Taktzyklus muss ein Befehl geholt werden, damit die Pipeline immer beschäftigt ist. Jedoch steht in unserem Entwurf die Entscheidung, ob ein Sprung ausgeführt wird, erst in der MEM-Pipelinstufe an. Wie in Abschnitt 6.1 bereits erwähnt, wird diese Verzögerung beim Bestimmen des Befehls, der geholt werden soll, im Gegensatz zu den eben beschriebenen Datenkonflikten als *Steuerkonflikt* oder *Pipelinehemmnisse durch Kontrollflussabhängigkeiten* bezeichnet.

Dieser Abschnitt über Steuerkonflikte ist kürzer als die vorhergehenden Abschnitte über Datenkonflikte. Dafür gibt es verschiedene Gründe: Steuerkonflikte sind relativ einfach zu verstehen, Steuerkonflikte treten seltener auf als Datenkonflikte und es gibt nichts, das so wirksam gegen Steuerkonflikte ist, wie das Forwarding gegen Datenkonflikte. Also verwenden wir einfachere Methoden. Wir werden zwei Methoden zum Beheben von Steuerkonflikten und eine Möglichkeit zum Optimieren dieser Methoden betrachten.

Annahme, dass Sprünge nicht ausgeführt werden

Wie wir in Abschnitt 6.1 gesehen haben, dauert es zu lange, wenn wir warten, bis der Sprung durchgeführt ist. Ein gängiges Mittel zur Verbesserung der Leistung gegenüber dem reinen Abwarten bis der Sprung durchgeführt ist, besteht darin, anzunehmen, dass der Sprung nicht ausgeführt wird, und mit der Ausführung der nachfolgenden Befehle fortzufahren. Wenn der Sprung doch ausgeführt wird, werden die Befehle verworfen, die bereits geholt und entschlüsselt wurden. Die Ausführung wird am Sprungziel fort-

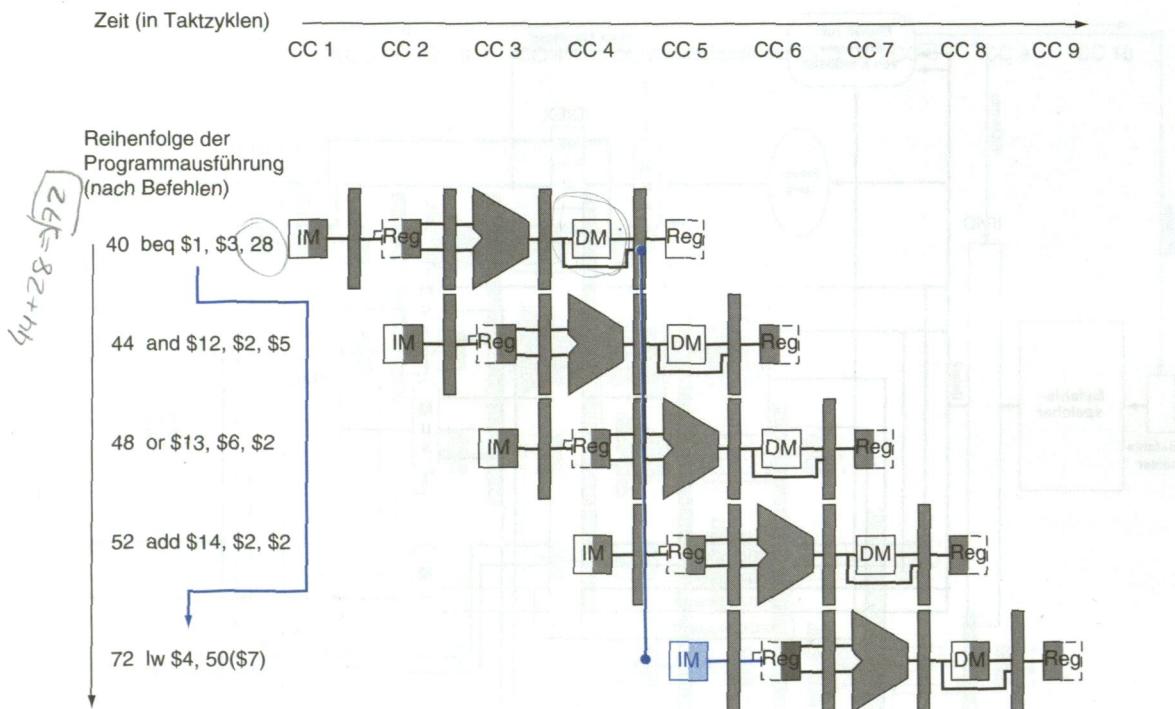


Abb. 6.32 Auswirkung der Pipeline auf den Sprungbefehl. Die Zahlen links neben den Befehlen (40, 44, ...) sind die Befehlsadressen. Da der Sprungbefehl in der MEM-Stufe (Taktzyklus 4 beim obigen `beq`-Befehl) entscheidet, ob ein Sprung ausgeführt wird, werden die drei sequenziellen Befehle, die dem Sprung folgen, geholt und die Ausführung wird begonnen. Ohne Eingriff der Steuerung beginnt die Ausführung dieser drei nachfolgenden Befehle, bevor `beq` zu `lw` auf Adresse 72 springt. (In Abbildung 6.6 wird zusätzliche Hardware vorausgesetzt, um den Steuerkonflikt auf genau nur einen Taktzyklus zu beschränken. In dieser Abbildung ist der nicht optimierte Datenpfad dargestellt.)

gesetzt. Wenn die Hälfte der Sprünge nicht ausgeführt wird, und wenn das Verwerfen der Befehle nur geringe Kosten verursacht, werden die durch Steuerkonflikte verursachten Kosten durch diese Optimierung halbiert.

Um Befehle zu verwerfen, ändern wir lediglich die ursprünglichen Steuerwerte in Nullen, ähnlich wie wir das auch zum Verzögern der Pipeline bei einem Load-use-Datenkonflikt getan haben. Der Unterschied besteht jedoch darin, dass wir hier auch die drei Befehle in den Stufen IF, ID und EX ändern müssen, wenn der Sprung in die MEM-Stufe gelangt. Bei Verzögerungen aufgrund von Load-use-Konflikten haben wir lediglich den Steuerwert in der ID-Stufe auf 0 gesetzt und die Verzögerungen durch die Pipeline mitgeführt. Verwerfen von Befehlen bedeutet somit, dass wir in der Lage sein müssen, in den Stufen IF, ID und EX die **Pipeline zu leeren (flush instructions)**.

Leeren der Pipeline (flush instructions) Beseitigen von Befehlen aus einer Pipeline, in der Regel aufgrund eines unerwarteten Ereignisses.

Reduktion der Verzögerung durch Sprünge

Das Leistungsverhalten von Sprüngen lässt sich durch Reduzieren der Kosten für den ausgeführten Sprung verbessern. Bisher sind wir davon ausgegangen, dass der nächste Befehlszähler für einen Sprung in der MEM-Stufe ausgewählt wird. Wenn wir jedoch die Sprungausführung in der Pipeline weiter nach vorn verlegen, müssen weniger Befehle verworfen werden. Die MIPS-Architektur wurde entwickelt, um schnelle Ein-taktsprünge zu unterstützen, die mit geringen Einbußen aufgrund von Sprüngen in der Pipeline verarbeitet werden können. Die Entwickler beobachteten, dass viele Sprünge nur von einfachen Überprüfungen (beispielsweise auf Gleichheit oder Vorzeichen) abhängen und dass Überprüfungen dieser Art keine komplette ALU-Operation bean-

spruchen, sondern mit maximal einigen wenigen Gattern durchgeführt werden können. Wenn eine komplexere Sprungentscheidung erforderlich wird, wird ein eigener Befehl benötigt, der mithilfe der ALU einen Vergleich durchführt, eine Situation, ähnlich der, bei der Bedingungscodes für Sprünge verwendet werden.

Wenn die Sprungentscheidung weiter vorn in der Pipeline erfolgen soll, müssen zwei Aktionen bereits zu einem früheren Zeitpunkt durchgeführt werden: die Berechnung der Sprungzieladresse und die Beurteilung der Sprungentscheidung. Der einfache Teil dieser Änderung ist das Verlegen der Berechnung der Sprungzieladresse. Im IF/ID-Pipelinerregister befindet sich bereits der Befehlszählerwert und das Immediate-Feld. Wir brauchen also nur noch den Sprungaddierer aus der EX-Stufe in die ID-Stufe zu verlegen. Dabei werden allerdings für alle Befehle die Sprungzieladressen berechnet, die jedoch nur bei Bedarf verwendet werden.

Der schwierigere Teil ist die Sprungentscheidung an sich. Bei einem Branch equal-Befehl vergleichen wir die beiden Register, die während der ID-Stufe gelesen werden, um festzustellen, ob die Werte in den beiden Registern gleich sind. Auf Gleichheit kann geprüft werden, indem zunächst mit den jeweiligen Bits eine Exklusiv-ODER-Verknüpfung (XOR) und anschließend mit den Ergebnissen eine ODER-Verknüpfung (OR) durchgeführt wird. Wenn die Sprungüberprüfung in die ID-Stufe verlegt wird, ist zusätzliche Forwarding-Hardware und Hardware zum Erkennen von Konflikten erforderlich, da ein Sprung, der von einem Ergebnis abhängt, das sich noch in der Pipeline befindet, auch mit dieser Optimierung ordnungsgemäß funktionieren muss. Um beispielsweise den Branch-on-equal-Befehl (und dessen Invertierung) zu implementieren, müssen wir Ergebnisse mittels Forwarding an die Logik zum Überprüfen auf Gleichheit weiterleiten, die während der ID-Stufe aktiv ist. Hierbei spielen zwei kritische Faktoren eine Rolle:

1. Wir müssen den Befehl während der ID-Stufe entschlüsseln, entscheiden, ob eine Weitergabe zur Einheit zum Überprüfen auf Gleichheit erforderlich ist, und die Überprüfung auf Gleichheit durchführen, so dass wir den Befehlszähler auf die Sprungzieladresse setzen können, wenn der Befehl ein Sprungbefehl ist. Das Forwarding der Operanden von Sprüngen wurde bisher von der Forwarding-Einheit der ALU übernommen. Wenn wir die Einheit zum Überprüfen auf Gleichheit jedoch in die ID-Stufe verlegen, benötigen wir eine neue Forwarding-Logik. Die mittels Forwarding weitergeleiteten Quelloperanden eines Sprungs können im Übrigen entweder vom ALU/MEM- oder vom MEM/WB-Pipeline-Register bereitgestellt werden.
2. Da die Werte in einem Sprungvergleich während der ID-Stufe benötigt werden, jedoch eventuell erst zu einem späteren Zeitpunkt generiert werden, tritt möglicherweise ein Datenkonflikt auf und es wird eine Pipelineverzögerung benötigt. Wenn beispielsweise ein ALU-Befehl direkt vor einem Sprung einen der Operanden für den Vergleich im Sprungbefehl generiert, ist eine Pipelineverzögerung erforderlich, da die EX-Stufe des ALU-Befehls nach der ID-Stufe des Sprungs ausgeführt wird.

Trotz dieser Schwierigkeiten stellt das Verlegen der Sprungausführung in die ID-Stufe eine Verbesserung dar, da dadurch die Einbußen aufgrund eines Sprungs auf nur einen Befehl beschränkt werden, wenn der Sprung ausgeführt wird, nämlich auf den Befehl, der zu diesem Zeitpunkt geholt wird. In den Übungen werden die Details zur Implementierung des Forwarding-Pfads und zur Erkennung des Konflikts untersucht.

Um die Pipeline in der IF-Stufe zu leeren, fügen wir eine Steuerleitung ein, die als IF.Flush bezeichnet wird und die das Befehlsfeld des IF/ID-Pipelinerregisters auf 0 setzt. Durch das Löschen des Registers wird der geholte Befehl in einen NOP-Befehl umgewandelt, einen Befehl, der keine Aktion bewirkt und den Zustand nicht verändert.

BEISPIEL

Sprünge mit Pipelining

Zeigen Sie, was geschieht, wenn der Sprung in dieser Befehlsfolge ausgeführt wird, vorausgesetzt, die Pipeline ist für Sprünge optimiert, die nicht ausgeführt werden, und die Sprungausführung wurde in die ID-Stufe verlegt:

```

36 sub $10, $4, $8
40 beq $1, $3, 7 # Befehlszeiger-relative Verzweigung
                  # nach 40 + 4 + 7 * 4 = 72
44 and $12, $2, $5
48 or $13, $2, $6
52 add $14, $4, $2
56 slt $15, $6, $7
...
72 lw $4, 50($7)

```

ANTWORT

In Abbildung 6.33 ist dargestellt, was geschieht, wenn ein Sprung ausgeführt wird. Im Gegensatz zu Abbildung 6.32 wird bei einem ausgeführten Sprung nur ein Bubble benötigt.

Dynamische Sprungvorhersage

Die Annahme, dass ein Sprung nicht ausgeführt wird, ist eine einfache Form der *Sprungvorhersage*. In diesem Fall sagen wir vorher, dass Sprünge nicht ausgeführt werden und leeren die Pipeline, wenn die Vorhersage falsch war. Bei der einfachen fünfstufigen Pipeline ist ein derartiger Ansatz möglicherweise gekoppelt mit einer compilerbasierenden Vorhersage wohl angemessen. Bei Pipelines mit mehr Stufen nehmen die Kosten einer falschen Sprungvorhersage (engl. branch penalty) gemessen in Taktzyklen zu. In ähnlicher Weise nehmen bei der Mehrfachzuordnung (Multiple Issue) die Kosten einer falschen Sprungvorhersage in Form von verlorenen Befehlen zu. Diese Kombination bedeutet, dass in einer aggressiven Pipeline mit einer einfachen statischen Vorhersagemethode möglicherweise zu viel Leistung verloren geht. Wie in Abschnitt 6.1 bereits erwähnt, ist es möglich, mit zusätzlicher Hardware das Sprungverhalten während der Programmausführung vorherzusagen.

Ein Ansatz besteht darin, anhand der Befehlsadresse nachzuschlagen, ob bei der letzten Ausführung des Befehls ein Sprung ausgeführt wurde, und wenn dies der Fall ist, neue Befehle von derselben Stelle wie beim letzten Mal zu holen. Diese Technik wird als **dynamische Sprungvorhersage (dynamic branch prediction)** bezeichnet.

Eine Art der Implementierung dieses Ansatzes stellt der **Sprungvorhersagepuffer (branch prediction buffer)** oder die **Sprungverlaufstabelle (branch history table)** dar. Ein Sprungvorhersagepuffer ist ein kleiner Speicher, der durch den unteren Teil der Adresse des Sprungbefehls adressiert wird. Der Speicher enthält ein Bit, das angibt, ob der Sprung in letzter Zeit ausgeführt wurde.

Dies ist die einfachste Art Puffer. Wir wissen noch nicht einmal, ob die Vorhersage stimmt. Schließlich kann sie durch einen anderen Sprung verursacht werden, der über dieselben niedrigerwertigen Adressbit verfügt. Das hat jedoch keine Auswirkungen auf die Richtigkeit. Eine Vorhersage ist nur ein Hinweis, von dem angenommen wird, dass er stimmt. Daher beginnt das Holen in der vorhergesagten Richtung. Wenn sich der Hinweis als falsch herausstellt, werden die falsch vorhergesagten Befehle gelöscht, das Vorhersagebit wird invertiert und wieder gespeichert, und die richtige Sequenz wird geholt und ausgeführt.

Dynamische Sprungvorhersage (dynamic branch prediction)
Vorhersage von Sprüngen zur Laufzeit mithilfe von Laufzeitinformationen.

Sprungvorhersagepuffer (branch prediction buffer) Auch als **Sprungverlaufstabelle (branch history table)** bezeichnet. Ein kleiner Speicher, der durch den unteren Teil der Adresse des Sprungbefehls adressiert wird und der ein oder mehrere Bit enthält, die angeben, ob der Sprung in letzter Zeit ausgeführt wurde.

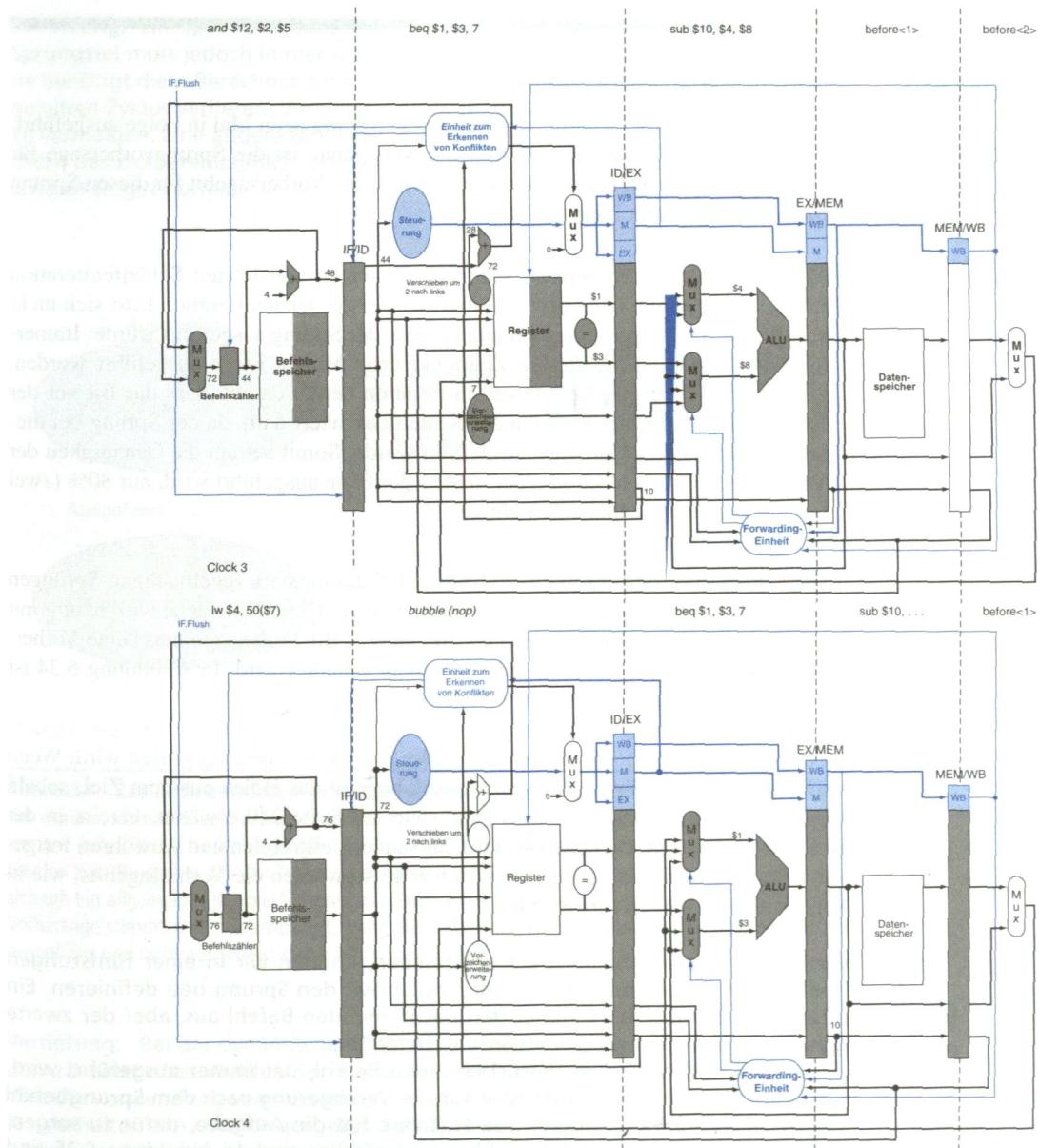


Abb. 6.33 Die ID-Stufe von Taktzyklus 3 bestimmt, dass ein Sprung ausgeführt werden muss, und wählt daher 72 als nächste Befehlszähleradresse aus und setzt den für den nächsten Taktzyklus geholten Befehl auf 0. In Taktzyklus 4 wird der Befehl auf Adresse 72 geholt, und der Bubble- oder NOP-Befehl in der Pipeline wird als Ergebnis des ausgeführten Sprungs dargestellt. (Da der NOP-Befehl eigentlich ein \$11 \$0, \$0, 0-Befehl ist, lässt sich darüber streiten, ob die ID-Stufe in Taktzyklus 4 farblich hervorzuheben ist oder nicht.)

Diese einfache 1-Bit-Sprungvorhersage weist hinsichtlich der Leistung einen Nachteil auf: Auch wenn ein Sprung fast immer ausgeführt wird, ergeben sich nicht nur eine, sondern zwei falsche Vorhersagen, wenn der Sprung nicht ausgeführt wird. Anhand des folgenden Beispiels wird dieses Dilemma deutlich.

BEISPIEL

ANTWORT

Schleifen und Vorhersagen

Stellen Sie sich eine Schleife vor, bei der ein Sprung neun Mal in Folge ausgeführt, und dann einmal nicht ausgeführt wird. Wie genau ist die Sprungvorhersage für diesen Sprung, wenn wir davon ausgehen, dass das Vorhersagebit für diesen Sprung im Vorhersagepuffer bleibt?

Mit der statischen Vorhersage wird bei der ersten und letzten Schleifeniteration falsch vorhergesagt. Die falsche Vorhersage bei der letzten Iteration lässt sich nicht vermeiden, da das Vorhersagebit angibt, dass der Sprung ausgeführt wurde: Immerhin war der Sprung zu diesem Zeitpunkt neun Mal in Folge ausgeführt worden. Die falsche Vorhersage bei der ersten Iteration beruht darauf, dass das Bit vor der Ausführung der letzten Iteration der Schleife aktiviert wird, da der Sprung bei dieser beendenden Iteration nicht ausgeführt wurde. Somit beträgt die Genauigkeit der Vorhersage für diesen Sprung, der in 90% der Fälle ausgeführt wird, nur 80% (zwei falsche Vorhersagen und acht richtige).

Die Genauigkeit des Prädiktors entspricht bei diesen stark regelmäßigen Sprüngen im Idealfall der Häufigkeit der ausgeführten Sprünge. Diese Schwäche wird häufig mit 2-Bit-Sprungvorhersagen ausgeglichen. Bei einer 2-Bit-Vorhersage muss eine Vorhersage zweimal falsch sein, damit die Vorhersage geändert wird. In Abbildung 6.34 ist der endliche Automat für eine 2-Bit-Vorhersage dargestellt.

Ein Sprungvorhersagepuffer kann als kleiner Spezialpuffer implementiert werden, auf den während der IF-Pipelinstufe mit der Befehlsadresse zugegriffen wird. Wenn der Befehl als ausgeführt vorhergesagt wird, beginnt das Holen aus dem Ziel, sobald der Befehlszähler bekannt ist. Das kann, wie auf Seite 340 erwähnt, bereits in der ID-Stufe der Fall sein. Andernfalls wird das sequenzielle Holen und Ausführen fortgesetzt. Wenn sich die Vorhersage als falsch erweist, werden die Vorhersagebits, wie in Abbildung 6.34 dargestellt, geändert.



Verzögerung nach Sprungbefehl (branch delay slot) Das Zeitfenster direkt nach einem verzögerten Sprungbefehl, das bei der MIPS-Architektur mit einem Befehl gefüllt wird, der auf den Sprung keine Auswirkungen hat.

Vertiefung: Wie in Abschnitt 6.1 beschrieben, können wir in einer fünfstufigen Pipeline den Steuerkonflikt beheben, indem wir den Sprung neu definieren. Ein verzögerter Sprung führt immer den nachfolgenden Befehl aus, aber der zweite Befehl nach dem Sprung ist vom Sprung betroffen.

Compiler und Assembler versuchen einen Befehl, der immer ausgeführt wird, nach dem Sprung in das Zeitfenster für die **Verzögerung nach dem Sprungbefehl (branch delay slot)** einzufügen. Die Software hat die Aufgabe, dafür zu sorgen, dass die nachfolgenden Befehle gültig und nützlich sind. In Abbildung 6.35 sind die drei Möglichkeiten dargestellt, wie die Verzögerung nach einem Sprungbefehl genutzt werden kann.

Die Grenzen des Scheduling verzögter Sprünge ergeben sich zum einen aus den Einschränkungen der Befehle, die für die Zeitfenster geplant werden, und zum anderen aus unserer Fähigkeit, zum Zeitpunkt des Kompilierens vorherzusagen, ob ein Sprung ausgeführt wird.

Der verzögerte Sprung war eine einfache und effektive Lösung für eine fünfstufige Pipeline, die pro Taktzyklus einen Befehl zuordnet. Da Prozessoren zunehmend sowohl längere Pipelines verwenden als auch pro Taktzyklus mehrere Befehle zuordnen (siehe Abschnitt 6.9), wird die Verzögerung nach einem Sprung länger und ein einzelnes Zeitfenster reicht für die Verzögerung nicht mehr aus. Der verzögerte Sprung hat daher zugunsten teurerer, aber flexiblerer dynamischer Lösungen an Beliebtheit eingebüßt. Gleichzeitig wurde die dynamische Vorhersage aufgrund der Zunahme der Transistoren pro Chip vergleichsweise günstiger.

Vertiefung: Ein Sprung-Prädiktor sagt uns, ob ein Sprung ausgeführt wird, das Sprungziel muss jedoch immer noch berechnet werden. In der fünfstufigen Pipeline benötigt diese Berechnung einen Zyklus, was bedeutet, dass ausgeführte Sprünge einen Zyklus verlieren. Verzögerte Sprünge sind eine Möglichkeit, diese Kosten zu vermeiden. Eine andere Möglichkeit ist die Verwendung eines Caches zum Speichern des Zielbefehlszählers oder des Zielbefehls mithilfe eines **Sprungzielpuffers** (*branch target buffer*).



Sprungzielpuffer (*branch target buffer*) Eine Struktur, die den Zielbefehlszähler oder den Zielbefehl für einen Sprung im Cache zwischenspeichert. Dieser Puffer ist in der Regel als Cache mit Tags organisiert, wodurch er teurer als ein einfacher Vorhersagepuffer ist.

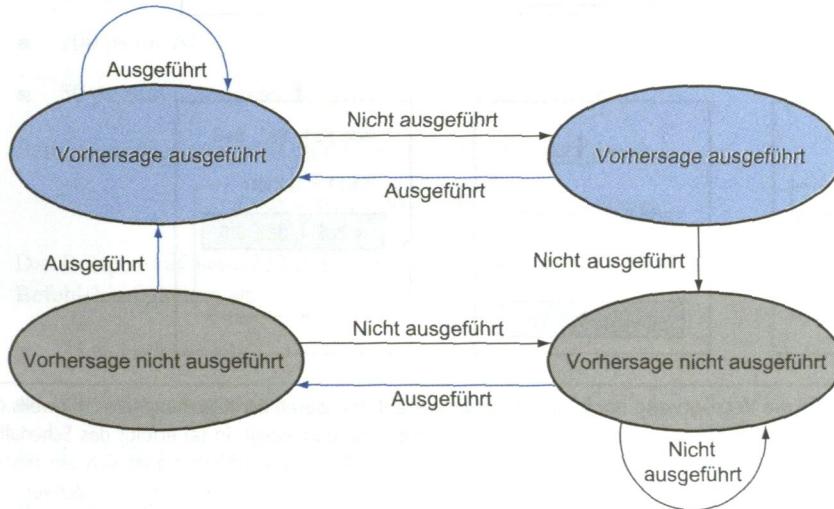


Abb. 6.34 Zustände bei einer 2-Bit-Vorhersage. Weil anstelle von nur einem Bit zwei Bit verwendet werden, kommt es bei Sprüngen, die mit großer Regelmäßigkeit ausgeführt oder nicht ausgeführt werden (wie das bei vielen Sprüngen der Fall ist), nur zu einer falschen Vorhersage. Die zwei Bit werden zum Darstellen der vier Zustände im System verwendet. Bei dieser 2-Bit-Methode handelt es sich um ein allgemeines Beispiel für einen zählerbasierten Prädiktor, der inkrementiert wird, wenn die Vorhersage stimmt, und dekrementiert, wenn die Vorhersage falsch ist, und der die Grenze zwischen ausgeführt und nicht ausgeführt in der Mitte des eigenen Bereichs zieht.

Vertiefung: Bei der dynamischen 2-Bit-Sprungvorhersage werden nur Informationen zu einem bestimmten Sprung verwendet. Forscher haben herausgefunden, dass die Verwendung von Informationen zu einem lokalen Sprung und dem globalen Verhalten von kürzlich ausgeführten Sprüngen bei gleicher Anzahl Vorhersagebits eine größere Vorhersagegenauigkeit ergibt. Prädiktoren dieser Art werden als **Korrelationsprädiktoren (*correlating predictor*)** bezeichnet. Ein typischer Korrelationsprädiktor verfügt beispielsweise über zwei 2-Bit-Prädiktoren für jeden Sprung und hat die Wahl zwischen Prädiktoren, die auf der Grundlage ausgewählt werden, ob der letzte Sprung ausgeführt wurde. Das globale Sprungverhalten können Sie sich wie das Hinzufügen zusätzlicher Indexbit für die Vorhersage vorstellen.

Eine neuere Entwicklung bei der Sprungvorhersage ist die Verwendung von Hybridprädiktoren. Ein **Hybridprädiktor (*tournament branch predictor*)** verwendet mehrere Prädiktoren und ermittelt für jeden Sprung, welcher Prädiktor die besten Ergebnisse liefert. Ein typischer Hybridprädiktor enthält beispielsweise zwei Vorhersagen für jeden Sprungindex: einer, der auf lokalen Informationen beruht, und einen, der auf dem globalen Sprungverhalten beruht. Eine Auswahllogik wählt den Prädiktor aus, der für eine gegebene Vorhersage verwendet wird. Die Auswahllogik kann ähnlich wie ein 1- oder 2-Bit-Prädiktor arbeiten und denjenigen der zwei Prädiktoren bevorzugen, der das genauere Ergebnis liefert. Ausgefeilte Prädiktoren wie diese werden in vielen neuen anspruchsvollen Mikroprozessoren eingesetzt.



Korrelationsprädiktor (*correlating predictor*) Ein Sprung-Prädiktor, der das lokale Verhalten eines bestimmten Sprungs mit globalen Informationen zum Verhalten einiger kürzlich ausgeführter Sprünge zusammen verwendet.

Hybridprädiktor (*tournament branch predictor*) Ein Sprung-Prädiktor mit mehreren Vorhersagen für jeden Sprung und einem Auswahlmechanismus, der den Prädiktor auswählt, der für einen bestimmten Sprung verwendet werden soll.

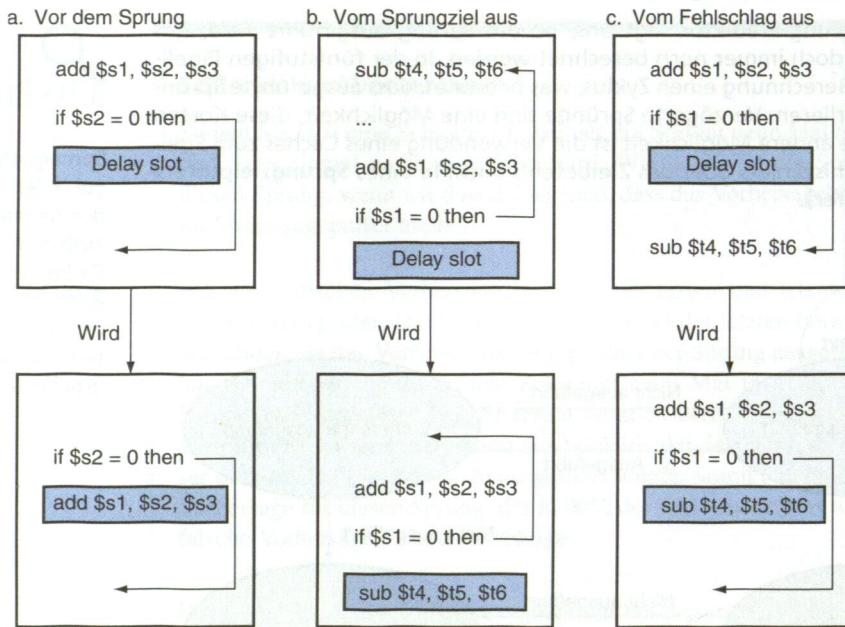


Abb. 6.35 Scheduling des Zeitfensters für die Verzögerung nach einem Sprungbefehl. Im oberen der Kästchenpaare ist jeweils der Code vor dem Scheduling dargestellt, in den unteren Kästchen ist der Code nach dem Scheduling dargestellt. In (a) erfolgt das Scheduling der Verzögerung mithilfe eines unabhängigen Befehls, der ursprünglich vor dem Sprung steht. Diese Möglichkeit eignet sich am besten. Die Strategien (b) und (c) werden verwendet, wenn (a) nicht möglich ist. In der Codesequenz für (b) und (c) wird durch die Verwendung von `$s1` in der Sprungbedingung verhindert, dass der `add`-Befehl (dessen Ziel `$s1` ist) in das Zeitfenster für die Verzögerung nach dem Sprung verschoben wird. In (b) erfolgt das Scheduling der Sprungverzögerung vom Sprungziel aus. Meist muss dabei der Zielbefehl kopiert werden, da ein anderer Pfad auf ihn zugreifen kann. Strategie (b) wird bevorzugt verwendet, wenn der Sprung wie etwa bei einem Sprung in einer Schleife mit großer Wahrscheinlichkeit ausgeführt wird. Schließlich kann das Scheduling des Sprungs wie in (c) vom nicht ausgeführten Verzweigungspfad (fall-through) aus erfolgen. Damit diese Optimierung für (b) oder (c) zulässig ist, muss es in Ordnung sein, den `sub`-Befehl auszuführen, wenn der Sprung in eine unerwartete Richtung geht. Mit „in Ordnung“ meinen wir, dass die Arbeit zwar überflüssig ist, das Programm aber dennoch ordnungsgemäß ausgeführt wird. Dies ist beispielsweise der Fall, wenn `$t4` ein nicht genutztes temporäres Register wäre und der Sprung in eine unerwartete Richtung ginge.

Pipeline – Eine Zusammenfassung

Bisher haben wir drei Modelle der Befehlausführung kennengelernt: die Einzyklenausführung, die Mehrzyklenausführung und die Ausführung mittels Pipeline. Das Ziel der Steuerung mit Pipelining ist einerseits wie bei Einzyklenausführung pro Befehl nur einen Taktzyklus zu benötigen, andererseits aber auch, einen kurzen Taktzyklus wie bei der Mehrzyklenausführung zu erreichen. Betrachten wir noch einmal das Beispiel mit dem Vergleich von Einzyklenprozessoren mit Mehrzyklenprozessoren.

BEISPIEL

Vergleich der Leistung verschiedener Steuermethoden

Vergleichen Sie die Leistung einer Einzyklen-, einer Mehrzyklen- und einer Pipelining-Steuerung mithilfe des SPECint2000-Befehlsmix (siehe Beispiele Seite 259 und 272) und gehen Sie dabei von derselben Zykluszeit pro Einheit wie in dem Beispiel auf Seite 259 aus. Bei der Ausführung mittels Pipeline gehen wir davon aus, dass der Hälfte der Ladebefehle ein Befehl folgt, der das Ergebnis des Ladebefehls verwendet, dass die Sprungverzögerung bei einer falschen Vorhersage 1 Taktzyklus

beträgt und dass ein Viertel der Sprünge falsch vorhergesagt wird. Gehen Sie davon aus, dass Sprünge immer einen ganzen Taktzyklus für die Verzögerung benötigen, so dass deren durchschnittliche Ausführungszeit 2 Taktzyklen beträgt. Lassen Sie andere Konflikte außen vor.

Dem Beispiel auf Seite 259 (Rechenleistung von Einzyklenrechnern) entnehmen wir für die Funktionseinheiten die folgenden Ausführungszeiten:

ANTWORT

- 200 ps für den Speicherzugriff
- 100 ps für ALU-Operationen
- 50 ps zum Lesen oder Schreiben des Registersatzes

Beim Einzyklendatenpfad ergibt dies folgenden Taktzyklus

$$(200 + 50 + 100 + 200 + 50) \text{ ps} = 600 \text{ ps}$$

Das Beispiel auf Seite 272 (CPI-Wert in einer Mehrzyklen-CPU) gibt die folgenden Befehlshäufigkeiten an:

- 25 % Ladebefehle
- 10 % Speicherbefehle
- 11 % Verzweigungen
- 2 % Sprünge
- 52 % ALU-Befehle

Darüber hinaus gibt das Beispiel auf Seite 272 an, dass der CPI-Wert für den Mehrzyklenentwurf 4,12 beträgt. Der Taktzyklus beim Mehrzyklen-Datenpfad und beim Entwurf mit Pipeline muss wie bei der Funktionseinheit mit der längsten Ausführungszeit 200 ps betragen.

Beim Entwurf mit Pipeline benötigen Ladebefehle einen Taktzyklus, wenn keine Load-use-Abhängigkeit vorliegt, und zwei Taktzyklen, wenn diese Abhängigkeit vorliegt. Die durchschnittliche Anzahl Taktzyklen pro Ladebefehl beträgt somit 1,5. Speicherbefehle erfordern ebenso wie ALU-Befehle einen Taktzyklus. Verzweigungen erfordern einen Taktzyklus, wenn sie richtig vorhergesagt werden, und zwei, wenn nicht. Somit beträgt die durchschnittliche Anzahl Taktzyklen pro Verzweigungsbefehl 1,25. Der CPI-Wert für Sprungbefehle beträgt 2. Somit ergibt sich für den durchschnittlichen CPI-Wert Folgendes:

$$1,5 \times 25\% + 1 \times 10\% + 1 \times 52\% + 1,25 \times 11\% + 2 \times 2\% = 1,17$$

Vergleichen wir nun die drei Entwürfe anhand der durchschnittlichen Befehlausführungszeit. Beim Einzyklenentwurf beträgt der Wert 600 ps. Beim Mehrzyklenentwurf beträgt der Wert $200 \times 4,12 = 824$ ps. Beim Entwurf mit Pipeline beträgt die durchschnittliche Befehlausführungszeit $1,17 \times 200 = 234$ ps, womit dieser Entwurf nahezu doppelt so schnell ist wie jeder andere Entwurf.

Dem aufmerksamen Leser ist möglicherweise bereits aufgefallen, dass die lange Zykluszeit bei Speicherzugriffen einen Leistungsengpass sowohl für Entwürfe mit Pipeline als auch für Mehrzyklenentwürfe darstellt. Wenn Speicherzugriffe auf zwei Taktzyklen aufgeteilt werden und der Taktzyklus dabei auf 100 ps reduziert wird, ließe sich die Rechenleistung in beiden Fällen verbessern.

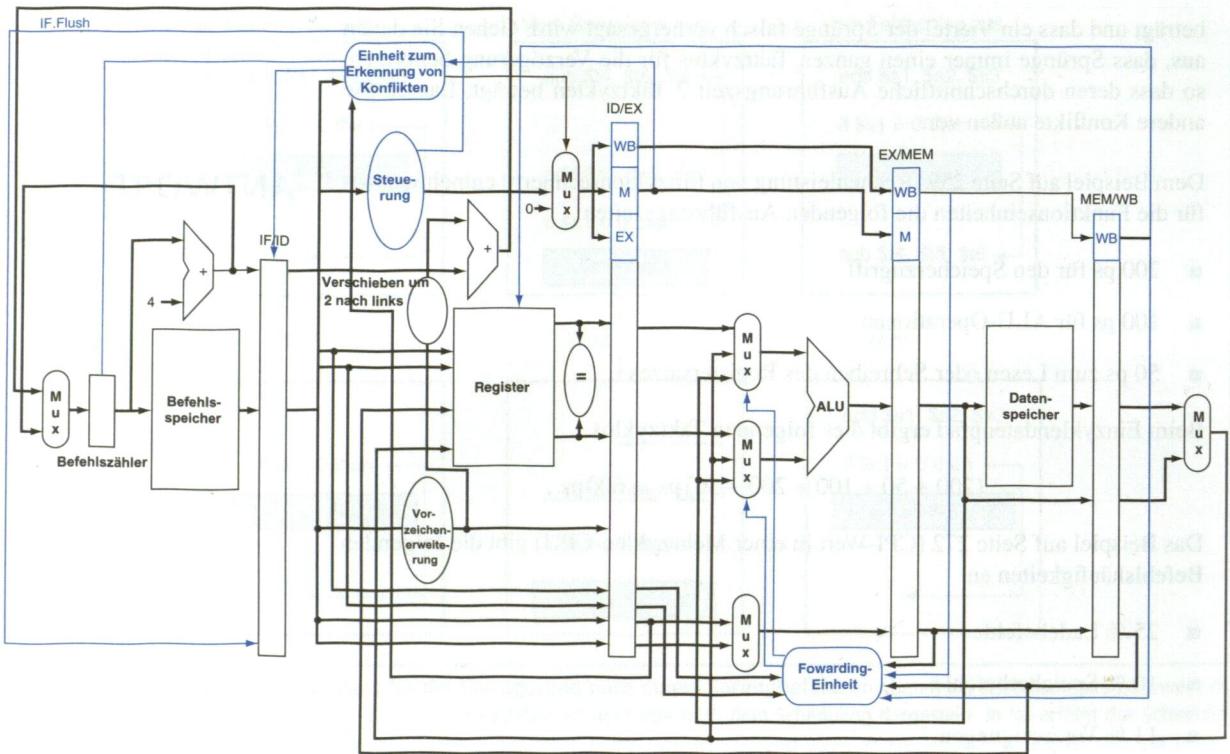


Abb. 6.36 Der vollständig entwickelte Datenpfad mit Steuerung für dieses Kapitel.

Dieses Kapitel hat beim Wäschewaschen begonnen, wo wir die Prinzipien des Pipelining in einer Alltagssituation kennen gelernt haben. Mit dieser Analogie haben wir das Pipelining von Befehlen Schritt für Schritt erkundet. Wir haben mit dem Einzyklen-datenpfad begonnen und haben Pipelineregister, Forwarding-Pfade, eine Einheit zum Erkennen von Datenkonflikten, eine Einheit für die Sprungvorhersage und eine Einheit zum Leeren der Pipeline im Fall von Unterbrechungen eingefügt. In Abbildung 6.36 ist der sich daraus ergebende Datenpfad mit Steuerung dargestellt.



Betrachten wir drei Vorhersagemethoden: Sprung nicht ausgeführt, Vorhersage ausgeführt und dynamische Vorhersage. Nehmen wir an, bei diesen Methoden entstehen keine Kosten, wenn die Vorhersage stimmt, und es entstehen Kosten in Höhe von zwei Zyklen, wenn die Vorhersage nicht stimmt. Nehmen wir außerdem an, die durchschnittliche Vorhersagegenauigkeit des dynamischen Prädiktors liegt bei 90%. Welcher Prädiktor eignet sich für die folgenden Sprünge am besten?

1. Ein Sprung, der mit einer Häufigkeit von 5 % ausgeführt wird
2. Ein Sprung, der mit einer Häufigkeit von 95 % ausgeführt wird
3. Ein Sprung, der mit einer Häufigkeit von 70 % ausgeführt wird

6.7

Verwendung einer Hardwarebeschreibungssprache zum Beschreiben und Modellieren einer Pipeline

Dieser Abschnitt (*Using a Hardware Description Language to Describe and Model a Pipeline*), der sich auf CD befindet, stellt ein Verhaltensmodell der fünfstufigen MIPS-Pipeline in Verilog bereit. Beim ersten Modell werden Konflikte außer Acht gelassen und Ergänzungen des Modells erläutern die Änderungen für Forwarding, Datenkonflikte und Steuerkonflikte.



6.8

Pipelinehemmnisse durch interne Unterbrechungen

Eine weitere Form von Steuerkonflikten stellen interne Unterbrechungen dar. Nehmen wir an, der Befehl

```
add $1,$2,$1
```

weist bei der Ausführung einen arithmetischen Überlauf auf. Wir müssen die Steuerung direkt nach diesem Befehl an die Unterbrechungsbehandlungsroutine übergeben, da wir nicht möchten, dass dieser ungültige Wert andere Register oder Speicherpositionen kontaminiert.

Wie beim ausgeführten Sprung im vorhergehenden Abschnitt müssen wir die Befehle nach dem add-Befehl aus der Pipeline löschen und Befehle aus der neuen Adresse holen. Wir verwenden denselben Mechanismus wie bei ausgeführten Sprüngen, wobei diesmal die Unterbrechung dafür sorgt, dass die Steuersignale auf logisch 0 gesetzt werden.

Bei falsch vorhergesagten Sprüngen haben wir gesehen, wie die Befehle in der IF-Stufe durch Umwandlung in einen NOP-Befehl gelöscht wurden. Um Befehle in der ID-Stufe zu löschen, verwenden wir den bereits in der ID-Stufe vorhandenen Multiplexer. Dieser setzt alle Steuersignale auf 0, um die Pipeline zu verzögern. Das neue Steuersignal ID.Flush wird mittels ODER-Verknüpfung (OR) mit dem Verzögerungssignal aus der Einheit zum Erkennen von Konflikten verknüpft, um den Befehl in der Pipeline während der ID-Stufe zu löschen. Um den Befehl in der EX-Phase zu löschen, verwenden wir das neue Signal EX.Flush, das mithilfe neuer Multiplexer die Steuersignale auf 0 setzt. Um das Holen von Befehlen aus Adresse 8000 0180_H, der Adresse für Unterbrechungsbehandlung bei einem arithmetischen Überlauf, zu beginnen, fügen wir einfach einen zusätzlichen Eingang am Befehlszählermultiplexer ein, der 8000 0180_H an den Befehlszähler sendet. In Abbildung 6.37 sind diese Änderungen dargestellt.

Anhand dieses Beispiels wird ein Problem im Zusammenhang mit Unterbrechungen deutlich: Wenn wir die Ausführung nicht mitten im Befehl unterbrechen, kann der Programmierer den Anfangswert von Register \$1, der zum Überlauf beigetragen hat, nicht erkennen, da er als das Zielregister des add-Befehls überschrieben wird. Dank einer sorgfältigen Planung wird der Überlauf während der EX-Stufe erkannt. Somit können wir mithilfe des EX-Flush-Signals verhindern, dass der Befehl in der EX-Stufe sein Ergebnis in der WB-Stufe schreibt. Bei vielen Unterbrechungen ist es erforderlich, den Befehl, der die Unterbrechung verursacht hat, letztendlich auszuführen, als wenn

To make a computer with automatic program-interruption facilities behave [sequentially] was not an easy matter, because the number of instructions in various stages of processing when an interrupt signal occurs may be large.

Fred Brooks Jr., *Planning a Computer System: Project Stretch*, 1962

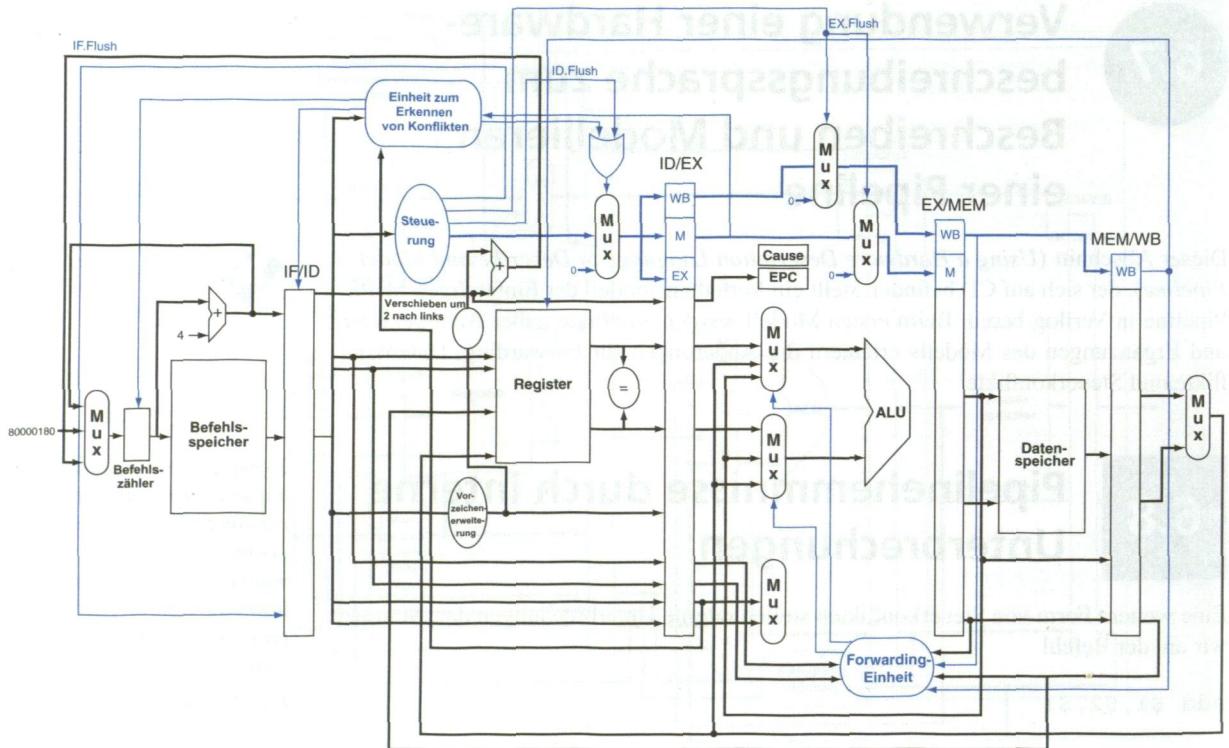


Abb. 6.37 Der Datenpfad mit Steuerung zur Unterbrechungsbehandlung. Zu den wichtigsten Erweiterungen zählen der neue Eingangswert $8000\ 0180_H$ am Multiplexer, der den neuen Befehlszählerwert bereitstellt, ein Unterbrechungseingangsregister (Cause-Register) zum Speichern der Ursache für die Unterbrechung, und ein Unterbrechungsbefehlszähler (EPC) zum Speichern der Adresse des Befehls, der die Unterbrechung verursacht hat. Der Eingangswert $8000\ 0180_H$ am Multiplexer ist die Anfangsadresse, an der Befehle im Fall einer Unterbrechung geholt werden. Das ALU-Überlaufsignal (nicht dargestellt) dient als Eingangssignal für die Steuereinheit.

er normal ausgeführt würde. Das geht am einfachsten, indem der Befehl nach der Unterbrechungsbehandlung aus der Pipeline gelöscht und von Anfang an neu begonnen wird.

In einem letzten Schritt wird die Adresse des verursachenden Befehls wie in Kapitel 5 im Unterbrechungsbefehlszähler EPC (Exception Program Counter) gespeichert. In Wirklichkeit speichern wir die Adresse + 4, so dass die Unterbrechungsbehandlungsroutine vom gespeicherten Wert zunächst 4 subtrahieren muss. In Abbildung 6.37 ist eine vereinfachte Version des Datenpfads mit der Sprunghardware und den erforderlichen Anpassungen für die Behandlung von Unterbrechungen dargestellt.

BEISPIEL

Unterbrechungen bei einem Rechner mit Pipelining

Gehen wir von der folgenden Befehlssequenz aus:

```

40H sub $11, $2, $4
44H and $12, $2, $5
48H or $13, $2, $6
4CH add $1, $2, $1
50H slt $15, $6, $7
54H lw $16, 50($7)
...

```

Wir nehmen an, dass die Befehle, die bei einer Unterbrechung aufgerufen werden sollen, wie folgt beginnen:

40000040_H sw \$25, 1000 (\$0)
40000044_H sw \$26, 1004 (\$0)

...

Zeigen Sie, was in der Pipeline geschieht, wenn im add-Befehl ein Überlauf auftritt.

In Abbildung 6.38 sind die Ereignisse dargestellt, wobei mit dem add-Befehl in der EX-Stufe begonnen wird. Der Überlauf wird während dieser Phase erkannt, und 4000 0040_H wird in den Befehlszähler geladen. In Taktzyklus 7 werden der add-Befehl und die nachfolgenden Befehle gelöscht, und der erste Befehl des Unterbrechungsbehandlungsprogramms wird geholt. Die Adresse des Befehls *nach* dem add-Befehl wird gespeichert: 4C_H + 4 = 50_H.

ANTWORT

In Kapitel 5 wurden einige weitere Ursachen für Unterbrechungen genannt:

- Anforderungen von Ein-/Ausgabegeräten
- Aufruf eines Betriebssystemdiensts von einem Benutzerprogramm aus
- Verwendung eines nicht definierten Befehls
- Fehlfunktion der Hardware

Wenn in einem beliebigen Taktzyklus fünf Befehle gleichzeitig aktiv sind, besteht die Schwierigkeit darin, eine Unterbrechung dem richtigen Befehl zuzuordnen. Zudem können in einem Taktzyklus gleichzeitig mehrere Unterbrechungen auftreten. Normalerweise werden den Unterbrechungen Prioritäten zugewiesen und es wird so festgelegt, welche Unterbrechung zuerst behandelt wird. Diese Strategie funktioniert auch bei Prozessoren mit Pipelines. Bei den meisten MIPS-Implementierungen sortiert die Hardware Unterbrechungen so, dass der früheste Befehl unterbrochen wird.

Anforderungen von Ein-/Ausgabegeräten und Fehlfunktionen der Hardware sind keinem bestimmten Befehl zuzuordnen, so dass die Implementierung hinsichtlich des Zeitpunkts, zu dem die Pipeline unterbrochen wird, über eine gewisse Flexibilität verfügt. Die für andere Unterbrechungen verwendeten Verfahren funktionieren daher hier problemlos.

Das EPC-Register speichert die Adresse der unterbrochenen Befehle und das MIPS-Cause-Register speichert alle möglichen Unterbrechungen in einem Taktzyklus, so dass die Unterbrechungsbehandlungsroutine die Unterbrechung dem Befehl zuordnen muss. Hierbei ist es sehr hilfreich zu wissen, in welcher Pipelinestufe ein bestimmter Unterbrechungstyp auftreten kann. So wird beispielsweise ein nicht definierter Befehl in der ID-Stufe erkannt und ein Befehl zum Aufrufen des Betriebssystems in der EX-Stufe. Unterbrechungen werden im Cause-Register gesammelt, so dass die Hardware die Ausführung wegen späterer Unterbrechungen unterbrechen kann, sobald die erste Unterbrechung behandelt wurde.

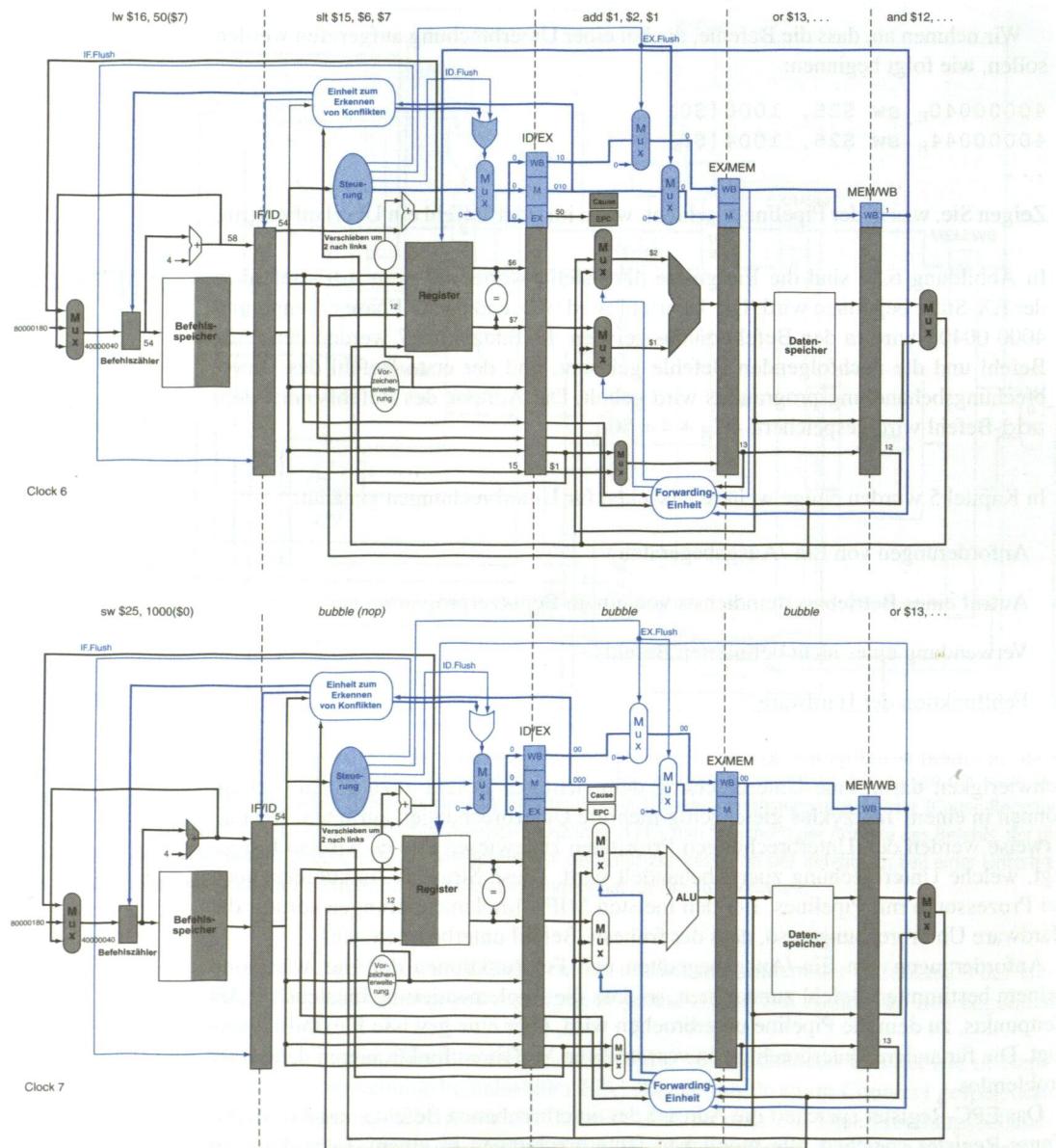


Abb. 6.38 Ergebnis einer Unterbrechung aufgrund eines arithmetischen Überlaufs im add-Befehl. Der Überlauf wird in der EX-Stufe in Takt 6 erkannt und die Adresse nach dem add-Befehl ($4C + 4 = 50_H$) wird im EPC-Register gespeichert. Aufgrund des Überlaufs werden alle Flush-Signale gegen Ende dieses Taktzyklus gesetzt und Steuerwerte für den add-Befehl auf logisch 0 gesetzt. In Taktzyklus 7 werden die Befehle zu Bubbles in der Pipeline umgewandelt und der erste Befehl der Unterbrechungsbehandlungsroutine (sw \$25, 1000(\$0)) wird aus der Befehlsposition $4000\ 0040_H$ geholt. Die Befehle and und or vor dem add-Befehl werden unverändert ausgeführt. Das ALU-Überlaufsignal (nicht dargestellt) dient als Eingangssignal für die Steuereinheit.

Hardware-Software-Schnittstelle

Die Hardware und das Betriebssystem müssen zusammenarbeiten, damit sich Unterbrechungen wie erwartet verhalten. Die Hardware unterbricht normalerweise den verursachenden Befehl sofort während der Ausführung, lässt alle zuvor begonnenen Befehle ausführen, löscht alle nachfolgenden Befehle, setzt ein Register zum Erkennen der Ursache für die Unterbrechung, speichert die Adresse des verursachenden Befehls und springt dann zu einer zuvor festgelegten Adresse. Das Betriebssystem untersucht

die Ursache für die Unterbrechung und verhält sich in angemessener Weise. Bei einem nicht definierten Befehl, einer Fehlfunktion der Hardware oder bei einem arithmetischen Überlauf, beendet das Betriebssystem normalerweise das Programm und zeigt den Grund dafür an. Bei Anforderungen von Ein-/Ausgabegeräten oder dem Aufruf eines Betriebssystemdiensts speichert das Betriebssystem den Zustand des Programms, führt die gewünschte Aufgabe aus und stellt danach das unterbrochene Programm wieder her, um es weiter auszuführen. Bei Anforderungen von Ein-/Ausgabegeräten werden wir vor der Wiederaufnahme des Prozesses, der die Ein-/Ausgabe angefordert hat, häufig einen anderen Prozess ausführen wollen, da der Prozess oft erst nach Abschluss der Ein-/Ausgabe fortgesetzt werden kann. Daher ist es so wichtig, den Zustand jedes Prozesses zu retten und wiederherzustellen. Eine der wichtigsten und häufigsten Verwendungsmöglichkeiten von Unterbrechungen ist die Behandlung von Seitenfehlern und TLB-Unterbrechungen. In Kapitel 7 werden diese Unterbrechungen und deren Behandlung ausführlicher beschrieben.

Die Schwierigkeit des Zuweisens der richtigen Unterbrechung zum richtigen Befehl bei Rechnern mit Pipelining hat einige Rechnerentwickler dazu gebracht, diese Anforderung in nicht kritischen Fällen zu lockern. Prozessoren dieser Art verfügen über so genannte **nicht präzise Interrupts** oder **nicht präzise Unterbrechungen (imprecise interrupt)**. Im obigen Beispiel enthält der Befehlszähler normalerweise zu Beginn des Taktzyklus, nachdem die Unterbrechung erkannt wurde, den Wert 58_H, auch wenn sich der verursachende Befehl an der Adresse 4C_H befindet. Ein Prozessor mit nicht präzisen Unterbrechungen speichert beispielsweise 58_H im EPC-Register und überlässt die Ermittlung des Befehls, der die Unterbrechung verursacht hat, dem Betriebssystem. MIPS und die überwiegende Mehrzahl der heutigen Rechner unterstützen **präzise Interrupts (precise interrupt)** oder **präzise Unterbrechungen**. (Ein Grund dafür ist die Unterstützung von virtuellem Speicher, wie wir in Kapitel 7 sehen werden.)

Die Entwickler von MIPS wollten, dass die Befehle für die Multiplikation und Division von Ganzzahlen parallel mit anderen Ganzzahlbefehlen ausgeführt werden können. Da die Befehle für die Multiplikation und Division mehrere Taktzyklen benötigen, diskutiert eine Gruppe Studenten, ob es möglich ist, präzise Unterbrechungen zu implementieren. Welche der folgenden Argumente sind zutreffend?

1. Es ist nicht möglich, präzise Unterbrechungen zu implementieren, da ein Befehl zum Multiplizieren oder Dividieren eine Unterbrechung erst nach der Ausführung von Folgebefehlen auslösen kann.
2. Es ist einfach, präzise Unterbrechungen zu implementieren, da ein Befehl zum Multiplizieren oder Dividieren nach Beginn der Ausführung keine Unterbrechung auslösen kann, so dass der zeitliche Ablauf aller Unterbrechungen offenkundig präzise ist.
3. Es spielt keine Rolle, ob die Befehle zum Multiplizieren und Dividieren eine Unterbrechung auslösen. Präzise Unterbrechungen können nicht implementiert werden, da Befehle zum Multiplizieren und Dividieren noch ausgeführt und nicht abgeschlossen werden können, wenn ein anderer Befehl eine Unterbrechung auslöst.
4. Obwohl ein Befehl zum Multiplizieren oder Dividieren möglicherweise noch ausgeführt wird, ist sichergestellt, dass er in Kürze abgeschlossen wird, und dann ist jede Unterbrechung für einen Nachfolgebefehl präzise.

Nicht präzise Unterbrechung

(*imprecise interrupt*) Auch als nicht präziser Interrupt bezeichnet. Unterbrechungen oder Interrupts in Computern mit Pipelining, die nicht exakt dem Befehl zugeordnet werden, der die Ursache für die Unterbrechung oder den Interrupt war.

Präzise Unterbrechung Auch als präziser Interrupt (*precise interrupt*) bezeichnet. Eine Unterbrechung oder ein Interrupt, der in einem Computer mit Pipelining immer dem richtigen Befehl zugeordnet wird.



6.9

Weitere Leistungssteigerung durch erweiterte Pipelining-Techniken

Seien Sie gewarnt: Die Abschnitte 6.9 und 6.10 stellen jeweils eine kurze Übersicht über faszinierende, aber auch komplexe Themen dar. Wenn Sie ausführlichere Informationen wünschen, sollten Sie unser Lehrbuch *Computer Architecture: A Quantitative Approach*, dritte Ausgabe, für Fortgeschrittene lesen. Dort wird das hier auf den folgenden 16 Seiten Beschriebene auf über 200 Seiten ausführlich erläutert!

**Befehlsebenen-Parallelität, ILP
(instruction-level parallelism)**
Die Parallelität innerhalb von Befehlen.

Mehrfachzuordnung (multiple issue) Ein Verfahren, bei dem in einem Taktzyklus mehrere Befehle gestartet werden.

Statische Mehrfachzuordnung (static multiple issue) Eine Methode zum Implementieren eines Prozessors mit Mehrfachzuordnung, bei dem viele Entscheidungen vom Compiler vor der Ausführung getroffen werden.

Dynamische Mehrfachzuordnung (dynamic multiple issue) Eine Methode zum Implementieren eines Prozessors mit Mehrfachzuordnung, bei dem viele Entscheidungen während der Ausführung vom Prozessor getroffen werden.

Pipelining nutzt die potenzielle Parallelität von Befehlen. Diese Parallelität wird als **Befehlsebenen-Parallelität (instruction-level parallelism) (ILP)** bezeichnet. Es gibt zwei wichtige Verfahren zum Verbessern des potenziellen Grades an Befehlsebenen-Parallelität. Beim ersten Verfahren wird die Tiefe der Pipeline vergrößert, so dass sich mehr Befehle überlappen. Bei unserem Beispiel mit dem Wäschewaschen könnten wir unter der Voraussetzung, dass der Waschmaschinenzyklus länger als alle anderen ist, die Waschmaschine in drei Maschinen zum Waschen, Spülen und Schleudern unterteilen. Wir würden dann anstelle einer vierstufigen eine sechsstufige Pipeline verwenden. Um die Beschleunigung optimal zu nutzen, müssen wir die restlichen Schritte zeitlich anpassen, so dass sie bei der Wäsche bzw. bei Prozessoren dieselbe Ausführungsdauer aufweisen. Der Parallelitätsgrad ist höher, da mehr Befehle gleichzeitig ausgeführt werden. Die Leistung ist potenziell höher, da der Taktzyklus kürzer gewählt werden kann.

Eine andere Möglichkeit besteht darin, die internen Komponenten des Rechners zu vervielfachen, so dass in jeder Pipelinestufe mehrere Befehle gestartet werden können. Dieses Verfahren wird als **Mehrfachzuordnung** oder **Multiple Issue** bezeichnet. Bei einer Wäscherei mit Mehrfachzuordnung gäbe es anstelle der Haushaltswaschmaschine und des Wäschetrockners beispielsweise drei Waschmaschinen und drei Wäschetrockner. Sie würden außerdem mehr Assistenten zum Zusammenfalten und Wegräumen der dreifachen Menge an Wäsche in derselben Zeit benötigen. Der Nachteil ist die zusätzliche Arbeit, die erforderlich ist, damit alle Maschinen ständig arbeiten, und die Notwendigkeit der Übertragung der Ladungen zur nächsten Pipelinestufe.

Wenn pro Stufe mehrere Befehle gestartet werden, wird die Befehlausführungsrate größer als die Taktrate, oder anders ausgedrückt: Der CPI-Wert wird kleiner als 1. Gelegentlich ist es passender, anstelle dieses Maßes den *IPC-Wert (Instructions per Clock cycle, Befehle pro Taktzyklus)* zu verwenden, insbesondere wenn die CPI-Werte kleiner als 1 werden! Ein Mikroprozessor mit Vierfachzuordnung mit einer Taktfrequenz von 6 GHz kann als theoretische Maximalleistung (peak performance) 24 Milliarden Befehle pro Sekunde ausführen und weist einen maximalen CPI-Wert von 0,25 oder einen IPC-Wert von maximal 4 auf. Wenn wir von einer fünfstufigen Pipeline ausgehen, befinden sich bei einem Prozessor dieser Art zu jedem beliebigen Zeitpunkt 20 Befehle in der Ausführung. Moderne Mikroprozessoren der oberen Leistungsklasse versuchen, in jedem Taktzyklus zwischen drei und acht Befehle zuzuordnen. Es gibt in der Regel jedoch viele Einschränkungen hinsichtlich der Befehlstypen, die gleichzeitig ausgeführt werden können, sowie hinsichtlich der Art und Weise, wie Abhängigkeiten aufgelöst werden.

Es gibt im Wesentlichen zwei Möglichkeiten, einen Prozessor mit Mehrfachzuordnung zu implementieren, die sich in erster Linie durch die Aufteilung der Arbeit zwischen dem Compiler und der Hardware voneinander unterscheiden. Da die Aufteilung der Arbeit bestimmt, ob Entscheidungen statisch (d.h. beim Kompilieren) oder dynamisch (d.h. während der Ausführung) getroffen werden, werden diese beiden Möglichkeiten auch **statische Mehrfachzuordnung (static multiple issue)** und **dynamische Mehrfachzuordnung (dynamic multiple issue)** bezeichnet. Wie wir noch sehen wer-

den, gibt es für beide Methoden andere, häufiger verwendete Bezeichnungen, die weniger präzise oder restriktiver sind.

In einer Pipeline mit Mehrfachzuordnung gibt es zwei wichtige und charakteristische Aufgaben, die wahrgenommen werden müssen:

1. Das Packen von Befehlen in **Zuordnungsfächer (issue slots)**. Wie bestimmt der Prozessor, wie viele und welche Befehle in einem gegebenen Taktzyklus zugeordnet werden können? Bei den meisten Prozessoren mit statischer Zuordnung wird dieser Prozess zumindest teilweise vom Compiler vorgenommen. Bei Entwürfen mit dynamischer Zuordnung übernimmt in der Regel der Prozessor die Zuordnung zur Laufzeit, auch wenn der Compiler häufig bereits versucht hat, die Zuordnungsrate durch eine vorteilhafte Anordnung der Befehle zu verbessern.
2. Der Umgang mit Daten- und Steuerkonflikten: Bei Prozessoren mit statischer Zuordnung werden einige oder alle Folgen aus Daten- und Steuerkonflikten vom Compiler statisch behandelt. Die meisten Prozessoren mit dynamischer Zuordnung versuchen dagegen zumindest einige Konflikte mithilfe von Hardwaretechniken aufzulösen, die zur Laufzeit eingesetzt werden.

Auch wenn wir diese Möglichkeiten als zwei separate Ansätze beschreiben, so weist in der Realität eine Technik Eigenschaften der anderen auf und es gibt eigentlich keinen Ansatz, der sich nur einer dieser Möglichkeiten zuordnen lässt.

Zuordnungsfächer (issue slots)

Die Positionen, von denen Befehle in einem gegebenen Taktzyklus zugeordnet werden können. In Analogie zur Leichtathletik entspricht dies den Positionen an den Startblöcken beim Kurzstreckenlauf.

Das Prinzip der Spekulation

Eine der wichtigsten Methoden zum Feststellen und zur besseren Nutzung der Befehlsebenen-Parallelität ist die Spekulation. **Spekulation (speculation)** ist ein Ansatz, der es dem Compiler oder Prozessor erlaubt, Vermutungen über Eigenschaften eines Befehls anzustellen, um so die Ausführung anderer Befehle zu beginnen, die möglicherweise von diesem spekulierten Befehl abhängen. So können wir beispielsweise über das Ergebnis einer Verzweigung spekulieren, so dass die Befehle nach der Verzweigung früher ausgeführt werden können. Oder wir können spekulieren, dass ein Speicherbefehl vor einem Ladebefehl nicht auf dieselbe Adresse verweist, so dass der Ladebefehl vor dem Speicherbefehl ausgeführt werden kann. Das Problem bei der Spekulation ist, dass sie falsch sein kann. Daher muss jeder Spekulationsmechanismus sowohl eine Methode enthalten, die überprüft, ob die Annahme stimmt, als auch eine, die die Wirkung des Befehls rückgängig macht, wenn die Annahme falsch war. Aufgrund der Implementierung dieser Funktion zum Rückgängigmachen wird ein Prozessor, der Spekulation unterstützt, entsprechend komplexer.

Die Spekulation kann im Compiler oder von der Hardware ausgeführt werden. Der Compiler kann beispielsweise mithilfe der Spekulation Befehle umordnen, einen Befehl über einen Sprung hinweg oder einen Ladebefehl über einen Speicherbefehl hinweg verschieben. Die Hardware des Prozessors kann dieselben Verschiebungen zur Laufzeit durchführen und dabei Techniken verwenden, die weiter unten in diesem Abschnitt beschrieben werden.

Die Wiederherstellungsmechanismen, die im Falle falscher Spekulationen verwendet werden, sind sehr unterschiedlich. Bei Spekulationen in der Software fügt der Compiler in der Regel zusätzliche Befehle ein, die die Richtigkeit der Spekulation überprüfen und eine Wiederherstellungsroutine bereitstellen, für den Fall, dass die Spekulation falsch war. Bei Hardwarespekulationen speichert der Prozessor die spekulativen Ergebnisse normalerweise in einem Puffer, bis er weiß, dass die Ergebnisse nicht mehr spekulativ sind. Wenn die Spekulation zutreffend war, werden die Befehle zu Ende bearbeitet, wobei der Inhalt der Puffer in die Register oder in den Speicher geschrieben wird. Wenn die Spekulation falsch war, leert die Hardware die Puffer und führt die richtige Befehlsfolge aus.

Spekulation (speculation)

Eine Methode, mit deren Hilfe der Compiler oder der Prozessor über das Ergebnis eines Befehls Annahmen macht, um ihn als eine Abhängigkeit aus der Ausführung anderer Befehle zu entfernen.

Die Spekulation bringt ein weiteres mögliches Problem mit sich: Beim spekulativen Ausführen bestimmter Befehle kann es zu Unterbrechungen kommen, die vorher nicht aufgetreten sind. Nehmen wir beispielsweise an, ein Ladebefehl ist spekulativ verschoben worden, aber die verwendete Adresse ist nicht zulässig, wenn die Spekulation falsch ist. Das führt dazu, dass eine Unterbrechung auftritt, die nicht auftreten sollte. Das Problem wird durch die Tatsache erschwert, dass die Unterbrechung auftreten muss, wenn der Ladebefehl nicht spekulativ ausgeführt wird! Bei der compilergestützten Spekulation werden Probleme wie diese durch eine zusätzliche spezielle Spekulationsbehandlung vermieden, mit deren Hilfe Unterbrechungen dieser Art ignoriert werden, bis klar ist, dass diese wirklich auftreten müssen. Bei der hardwaregestützten Spekulation werden Unterbrechungen einfach in einen Puffer gespeichert, bis klar ist, dass der Befehl, der die Unterbrechungen verursacht, nicht mehr spekulativ ist und ausgeführt werden kann. Dann wird die Unterbrechung ausgelöst, und die normale Unterbrechungsbehandlung wird ausgeführt.

Da die Leistung aufgrund einer richtigen Spekulation verbessert und aufgrund einer falschen Spekulation verschlechtert wird, muss mit großer Sorgfalt entschieden werden, ob die Spekulation überhaupt angewendet werden soll. Weiter unten in diesem Abschnitt, werden wir sowohl statische als auch dynamische Spekulationstechniken untersuchen.

Statische Mehrfachzuordnung

Zuordnungspaket (issue packet) Die Befehle, die in einem Taktzyklus zusammen zugeordnet werden. Das Paket kann vom Compiler statisch oder vom Prozessor dynamisch zusammengestellt werden.

Die Prozessoren mit statischer Mehrfachzuordnung verwenden alle den Compiler zum Packen von Befehlen und Behandeln von Konflikten. Bei einem Prozessor mit statischer Mehrfachzuordnung können Sie sich die Befehle, die in einem gegebenen Taktzyklus zugeordnet und als **Zuordnungspaket (issue packet)** bezeichnet werden, als einen großen Befehl mit mehreren Teilbefehlen vorstellen. Diese Vorstellung ist mehr als eine Analogie. Da ein Prozessor mit statischer Mehrfachzuordnung in der Regel den Befehlsmix einschränkt, der in einem gegebenen Taktzyklus initiiert werden kann, ist es hilfreich, sich das Zuordnungspaket als nur einen Befehl vorzustellen, der mehrere Teilbefehle in bestimmten vordefinierten Feldern zulässt. Diese Vorstellung führte zur ursprünglichen Bezeichnung für diesen Ansatz: VLIW (Very Long Instruction Word, sehr langes Befehlswort). Die Intel IA-64-Architektur nutzt diesen Ansatz unter einem eigenen Namen: Explicitly Parallel Instruction Computer (EPIC). Der 2000 auf den Markt gekommene Itanium- und der diesem 2002 folgende Itanium-2-Prozessor sind die ersten Implementierungen der IA-64-Architektur.

Die meisten Prozessoren mit statischer Zuordnung benötigen den Compiler außerdem zur Behandlung von Daten- und Steuerkonflikten. Der Compiler kann dabei beispielsweise für die statische Sprungvorhersage und das Code-Scheduling zum Reduzieren oder Vermeiden aller Konflikte verantwortlich sein.

Im Folgenden werden wir einen MIPS-Prozessor mit einer einfachen statischen Zuordnung betrachten, bevor wir die Verwendung dieser Techniken in Prozessoren mit anspruchsvoller Realisierung beschreiben. Nachdem wir anhand dieses Beispiels diese Aussagen dargelegt haben, beschreiben wir die Eigenschaften der Intel IA-64-Architektur.

Ein Beispiel für die statische Mehrfachzuordnung anhand der MIPS-Befehlssatzarchitektur

Um eine Vorstellung von der statischen Mehrfachzuordnung zu vermitteln, betrachten wir einen MIPS-Prozessor mit einer einfachen Zweifachzuordnung, bei der einer der Befehle eine ALU-Ganzzahloperation oder ein Sprungbefehl und der andere Befehl ein Lade- oder Speicherbefehl sein kann. Ein Entwurf dieser Art entspricht dem Entwurf.

Tab. 6.6 Pipeline mit statischer Zweifachzuordnung im Betrieb. Der ALU-Befehl und der Datentransfer-Befehl werden gleichzeitig zugeordnet. Wir haben hier dieselbe fünfstufige Struktur wie bei der Pipeline mit Einfachzuordnung vorausgesetzt. Obwohl dies nicht unbedingt erforderlich ist, bringt es einige Vorteile mit sich. Wenn sich die Befehle zum Schreiben in die Register am Ende der Pipeline befinden, vereinfacht dies die Behandlung von Unterbrechungen und erleichtert das Beibehalten eines präzisen Unterbrechungsmodells, das bei Prozessoren mit Mehrfachzuordnung entsprechend schwieriger zu realisieren ist.

Befehlstyp	Pipelinstufen						
ALU- oder Sprungbefehl	IF	ID	EX	MEM	WB		
Lade- oder Speicherbefehl	IF	ID	EX	MEM	WB		
ALU- oder Sprungbefehl		IF	ID	EX	MEM	WB	
Lade- oder Speicherbefehl		IF	ID	EX	MEM	WB	
ALU- oder Sprungbefehl			IF	ID	EX	MEM	WB
Lade- oder Speicherbefehl			IF	ID	EX	MEM	WB
ALU- oder Sprungbefehl				IF	ID	EX	MEM
Lade- oder Speicherbefehl				IF	ID	EX	WB

der in einigen eingebetteten MIPS-Prozessoren verwendet wird. Wenn pro Taktzyklus zwei Befehle zugeordnet werden, müssen 64-Bit-Befehle geholt und entschlüsselt werden. Bei vielen Prozessoren mit statischer Mehrfachzuordnung und insbesondere bei allen VLIW-Prozessoren wird das Layout von Befehlen mit gleichzeitiger Zuordnung eingeschränkt, um die Entschlüsselung und Befehlszuordnung zu vereinfachen. Daher ist es erforderlich, die Befehle paarweise anzutragen und an einer 64-Bit-Grenze auszurichten, wobei der ALU- oder Sprungteil zuerst dargestellt wird. Außerdem muss ein Befehl eines Paares, der nicht genutzt werden kann, durch einen NOP-Befehl ersetzt werden. Die Befehle werden also immer paarweise zugeordnet, wobei sich in einem Zuordnungsfach ein NOP-Befehl befinden kann. In Tabelle 6.6 ist dargestellt, wie die Befehle die Pipeline paarweise durchlaufen.

Prozessoren mit statischer Mehrfachzuordnung behandeln Daten- und Steuerkonflikte auf unterschiedliche Art und Weise. Bei einigen Entwürfen ist ausschließlich der Compiler für das Auflösen *aller* Konflikte, für das Scheduling des Codes und das Einfügen von NOP-Befehlen verantwortlich, so dass der Code ohne Konflikterkennung oder von der Hardware generierte Verzögerungen ausgeführt wird. Bei anderen Entwürfen erkennt die Hardware Datenkonflikte und generiert eine Verzögerung zwischen zwei Zuordnungspaketen, wobei der Compiler dafür zuständig ist, alle Abhängigkeiten innerhalb eines Befehlspaares zu vermeiden. Dennoch führt ein Konflikt in der Regel dazu, dass das gesamte Zuordnungspaket, das den abhängigen Befehl enthält, angehalten wird. Unabhängig davon, ob die Software alle Konflikte behandeln oder nur versuchen muss, die Anzahl der Konflikte zwischen verschiedenen Zuordnungspaketen zu reduzieren, wird der Wunsch nach einem großen Einzelbefehl mit mehreren Operationen verstärkt. Wir werden den zweiten Ansatz für dieses Beispiel betrachten.

Um eine ALU- und eine Datentransfer-Operation parallel zuzuordnen, werden neben der normalen Hardware zum Erkennen von Konflikten und der Verzögerungslogik in erster Linie zusätzliche Ports im Registersatz benötigt (siehe Abbildung 6.39). Möglicherweise müssen wir in einem Taktzyklus zwei Register für die ALU-Operation und zwei weitere für einen Speicherbefehl lesen. Zudem benötigen wir einen Schreibport für eine ALU-Operation und einen Schreibport für einen Ladebefehl. Da die ALU mit den ALU-Operationen beschäftigt ist, benötigen wir außerdem einen weiteren Ad-

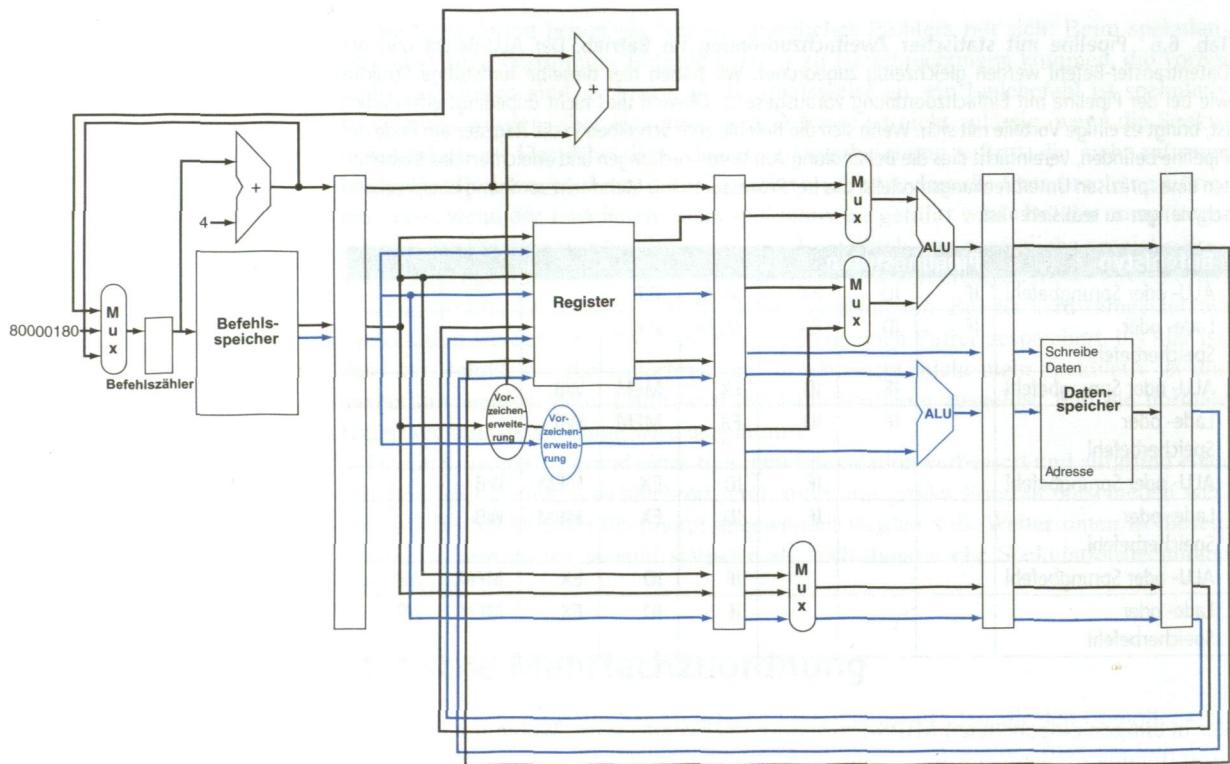


Abb. 6.39 Ein Datenpfad mit statischer Zweifachzuordnung. Die für eine Zweifachzuordnung erforderlichen Ergänzungen sind farblich hervorgehoben: weitere 32 Bits aus dem Befehlsspeicher, zwei weitere Leseports und ein zusätzlicher Schreibport im Registersatz und eine zusätzliche ALU. Die untere ALU ist für die Adressberechnung für Datentransfers zuständig, während die obere ALU für alles Andere verantwortlich ist.

dierer zum Berechnen der Effektivadresse für Datentransfers. Ohne diese zusätzlichen Ressourcen würde unsere Pipeline mit Zweifachzuordnung durch Strukturkonflikte behindert.

Dieser Prozessor mit Zweifachzuordnung kann die Leistung bis zu einem Faktor 2 verbessern. Dazu muss sich jedoch die Ausführung von doppelt so vielen Befehlen überlappen, und diese zusätzlichen Überlappungen erhöhen den relativen Leistungsverlust aus Daten- und Steuerkonflikten. In unserer einfachen fünfstufigen Pipeline weisen Ladebefehle beispielsweise eine Nutzungslatenz von einem Taktzyklus auf, wodurch verhindert wird, dass ein Befehl das Ergebnis ohne Verzögerung verwendet. Bei der fünfstufigen Pipeline mit Zweifachzuordnung kann das Ergebnis eines Ladebefehls im nachfolgenden *Taktzyklus* nicht verwendet werden. Das bedeutet, dass die nächsten *beiden* Befehle das Ergebnis des Ladebefehls nur mit Verzögerung nutzen können. Zudem haben ALU-Befehle, die in der einfachen fünfstufigen Pipeline keine Nutzungslatenz aufwiesen, nun eine Nutzungslatenz von einem Befehl, da das Ergebnis des Befehlspaares nicht verwendet werden kann. Um den in einem Prozessor mit Mehrfachzuordnung verfügbaren Parallelismus effektiv zu nutzen, werden anspruchsvollere Compiler- und Hardware-Schedulingtechniken benötigt, und bei der statischen Mehrfachzuordnung muss der Compiler diese Rolle übernehmen.

Code-Scheduling mit einfacher Mehrfachzuordnung

BEISPIEL

Wie würde der Code für diese Schleife in einer Pipeline mit statischer Zweifachzuordnung bei MIPS angeordnet?

```
Loop:    lw    $t0, 0($s1)      # $t0=Feldelement
addu   $t0,$t0,$s2      # Addiere Skalar in $2
sw     $t0, 0($s1)      # Speichere Ergebnis
addi   $s1,$s1,-4       # Dekrementiere Zeiger
bne    $s1,$zero,Loop    # Verzweige $s1!=0
```

Ordnen Sie die Befehle neu an, um so viele Pipelineleerläufe wie möglich zu vermeiden. Gehen Sie davon aus, dass Sprünge so vorhergesagt werden, dass Steuerkonflikte von der Hardware behandelt werden.

Die ersten drei Befehle weisen ebenso wie die letzten beiden Datenabhängigkeiten auf. In Tabelle 6.7 ist die beste Anordnung für diese Befehle dargestellt. Nur ein Befehlspaar nutzt dabei beide Zuordnungsfächer. Pro Schleifendurchlauf werden vier Taktzyklen benötigt. Wenn in vier Taktzyklen fünf Befehle ausgeführt werden, erzielen wir anstelle des optimalen Werts von 0,5 den enttäuschenden CPI-Wert 0,8 bzw. einen IPC-Wert von 1,25 statt 2,0. Beim Berechnen des CPI-Werts oder des IPC-Werts werden NOP-Befehle nicht als sinnvolle Befehle mitgezählt. Würden diese mitgezählt, würden wir einen besseren CPI-Wert erhalten, aber keine bessere Leistung!

Tab. 6.7 Neu angeordneter Code, wie er in einer MIPS-Pipeline mit Zweifachzuordnung aussehen würde. Die leeren Zuordnungsfächer sind NOP-Befehle.

	ALU- oder Sprungbefehl	Datentransfer-Befehl	Taktzyklus
Loop:		lw \$t0, 0(\$s1)	1
	addi \$s1,\$s1,-4		2
	addu \$t0,\$t0,\$s2		3
	bne \$s1,\$zero,Loop	sw \$t0, 4(\$s1)	4

Eine wichtige Compilertechnik zum Erzielen einer besseren Leistung aus Schleifen ist das **Schleifenabrollen (loop unrolling)**, eine Technik, bei der vom Schleifenrumpf mehrere Kopien erstellt werden. Nach dem Abrollen kann, da sich die Befehle aus unterschiedlichen Iterationen überschneiden, die Befehlsebenen-Parallelität besser genutzt werden.

ANTWORT

Schleifenabrollen (loop unrolling) Eine Technik zur Verbesserung der Leistung von Schleifen, die auf Felder zugreifen, bei der viele Kopien des Schleifenrumpfs erstellt und Befehle aus unterschiedlichen Iterationen im Sinne der Mehrfachzuordnung zusammengefasst werden.

Schleifenabrollen bei Pipelines mit Mehrfachzuordnung

BEISPIEL

Prüfen Sie, wie gut Schleifenabrollen und Scheduling im obigen Beispiel funktionieren. Gehen Sie der Einfachheit halber davon aus, dass der Schleifenindex ein Vielfaches von vier ist.

Um die Schleife ohne Verzögerungen anzurufen, müssen wir vier Kopien des Schleifenrumpfs erstellen. Nach dem Schleifenabrollen und Löschen der unnötigen Schleifenverwaltungsbefehle enthält die Schleife jeweils vier Kopien von `lw`, `add` und `sw` sowie eine von `addi` und eine von `bne`. In Tabelle 6.8 ist der Code nach dem Schleifenabrollen und Scheduling dargestellt.

ANTWORT

Registerumbenennung (*register renaming*) Die Umbenennung von Registern durch den Compiler oder durch die Hardware zum Auflösen von Antiabhängigkeiten.

Antiabhängigkeit (*anti-dependence*) Auch als **Namensabhängigkeit (*name dependence*)** bezeichnet. Eine Befehlsreihenfolge, die nicht durch eine echte Abhängigkeit, bei der ein Wert zwischen zwei Befehlen übertragen wird, sondern durch die Wiederverwendung eines Namens, in der Regel eines Registers, erzwungen wird.

Während des Schleifenabrollens hat der Compiler zusätzliche Register ($\$t1$, $\$t2$, $\$t3$) eingefügt. Dieser Prozess, der als **Registerumbenennung (*register renaming*)** bezeichnet wird, dient dazu, Abhängigkeiten aufzulösen, bei denen es sich zwar nicht um echte Datenabhängigkeiten handelt, die jedoch entweder zu potenziellen Konflikten führen oder verhindern können, dass der Compiler den Code flexibel generiert. Überlegen Sie, wie der Code nach dem Schleifenabrollen aussehen würde, wenn nur $\$t0$ verwendet würde. Es gäbe mehrere Kopien von `lw $t0, 0($$s1)`, `addu $t0,$t0,$s2` gefolgt von `sw t0, 4($$s1)`, aber diese Sequenzen sind trotz der Verwendung von $\$t0$ vollkommen unabhängig voneinander: Es fließen keine Daten zwischen diesem Befehlspaar und dem nachfolgenden. Hierbei handelt es sich nicht um eine echte Datenabhängigkeit, sondern vielmehr um eine so genannte **Antiabhängigkeit (*antidependence*)** oder **Namensabhängigkeit (*name dependence*)**, bei der eine Reihenfolge ausschließlich durch die Wiederverwendung eines Namens erzwungen wird.

Wenn die Register während des Schleifenabrollens umbenannt werden, kann der Compiler anschließend diese unabhängigen Befehle verschieben und so den Code besser packen. Mit der Umbenennung werden Namensabhängigkeiten aufgelöst, während die echten Abhängigkeiten erhalten bleiben.

Nun werden 12 der 14 Befehle in der Schleife paarweise ausgeführt. Für vier Schleifendurchläufe sind acht Taktzyklen oder zwei Taktzyklen pro Durchlauf erforderlich. Das ergibt einen CPI-Wert von $8/14 = 0,57$. Schleifenabrollen und Scheduling bei der Zweifachzuordnung ergab eine Verbesserung um den Faktor 2, die zum Teil auf die Reduzierung der Schleifenverwaltungsbefehle und zum Teil auf die Ausführung mittels Zweifachzuordnung zurückzuführen ist. Für diese Leistungsverbesserung werden anstelle von einem vier temporäre Register sowie eine erheblich größere Codegröße benötigt.

Tab. 6.8 Neu angeordneter Code aus Tabelle 6.7, wie er bei einer MIPS-Pipeline mit statischer Zweifachzuordnung nach dem Schleifenabrollen und Scheduling aussieht. Die leeren Zuordnungsfächer sind NOP-Befehle. Da der erste Befehl in der Schleife $\$s1$ um 16 dekrementiert, sind die Adressen, die geladen werden, der Anfangswert von $\$s1$, dann diese Adresse minus 4, minus 8 und minus 12.

	ALU- oder Sprungbefehl	Datentransfer-Befehl	Taktzyklus
Loop:	addi \$s1,\$s1,-16	lw \$t0, 0(\$\$s1)	1
		lw \$t1,12(\$\$s1)	2
	addu \$t0,\$t0,\$s2	lw \$t2, 8(\$\$s1)	3
	addu \$t1,\$t1,\$s2	lw \$t3, 4(\$\$s1)	4
	addu \$t2,\$t2,\$s2	sw \$t0, 16(\$\$s1)	5
	addu \$t3,\$t3,\$s2	sw \$t1,12(\$\$s1)	6
		sw \$t2, 8(\$\$s1)	7
	bne \$s1,\$zero,Loop	sw \$t3, 4(\$\$s1)	8

Die Intel IA-64-Architektur

Bei der IA-64-Architektur handelt es sich um einen Register-Register, RISC-artigen Befehlssatz wie bei der 64-Bit-Version der MIPS-Architektur (als MIPS-64 bezeichnet) – jedoch mit einer Reihe von besonderen Funktionen zur Unterstützung der expliziten compilergesteuerten Nutzung der Befehlsebenen-Parallelität. Intel nennt dieses Konzept EPIC (Explicitly Parallel Instruction Computer). Die Hauptunterschiede zwischen der IA-64- und der MIPS-Architektur sind:

1. IA-64 hat wesentlich mehr Register als MIPS, darunter 128 Ganzzahl- und 128 Gleitkommaregister, sowie 8 spezielle Register für Sprünge und 64 1-Bit-Bedingungsregister. Darüber hinaus unterstützt die IA-64-Architektur Registerfenster ähnlich wie die ursprüngliche Berkeley RISC- und Sun SPARC-Architektur.
2. IA-64 packt Befehle in Bündel mit festem Format und mit expliziter Kennzeichnung von Abhängigkeiten.
3. IA-64 enthält spezielle Befehle und Funktionen für die Spekulation und zum Eliminieren von Sprüngen, wodurch die Befehlsebenen-Parallelität besser genutzt werden kann.

Die IA-64-Architektur wurde entwickelt, um die wesentlichen Vorteile eines VLIW zu nutzen: implizite Parallelität der Teilbefehle in einem Befehl und feste Formatierung der Befehlsfelder, wobei gleichzeitig eine größere Flexibilität gewahrt wurde, als dies mit einem VLIW normalerweise möglich ist. Bei der IA-64-Architektur wird diese Flexibilität mithilfe von zwei unterschiedlichen Konzepten erreicht: Befehlsgruppen und Befehlsbündel (Bundles).

Eine **Befehlsgruppe (instruction group)** ist eine Folge von Befehlen ohne Registerdatenabhängigkeiten. Alle Befehle in einer Gruppe könnten parallel ausgeführt werden, wenn genügend Hardwareressourcen vorhanden wären und wenn Abhängigkeiten über den Speicher aufrechterhalten blieben. Eine Befehlsgruppe kann beliebig lang sein, aber der Compiler muss die Grenzen zwischen Befehlsgruppen *explizit* angeben. Diese Grenze wird durch Einfügen eines **Stopps (stop)** zwischen zwei Befehlen, die zu unterschiedlichen Gruppen gehören, angegeben.

IA-64-Befehle sind in *Befehlsbündel* oder *Bundles* zusammengefasst, die 128 Bit breit sind. Jedes Bundle besteht aus einem 5 Bit breiten Template-Feld (Muster) und drei Befehlen, die jeweils 41 Bit breit sind. Um die Entschlüsselung und die Befehlszuordnung zu vereinfachen, gibt das Template-Feld eines Bündels an, welche der fünf verschiedenen Ausführungseinheiten von den einzelnen Befehlen im Bündel benötigt wird. Zu den fünf verschiedenen Ausführungseinheiten zählen Ganzzahl-ALU, Nicht-Ganzzahl-ALU (Shifter und Multimediaoperationen), Speichereinheit, Gleitkommaeinheit und Sprungeinheit.

Das 5 Bit lange Template-Feld in jedem Bündel gibt an, ob dem Bündel zugewiesene Stopps vorhanden sind, *und* es gibt den Typ der Ausführungseinheit an, der von den einzelnen Befehlen im Bündel benötigt wird. Die Bündel-Formate können nur einen Teil aller möglichen Kombinationen aus Befehlstyp und Stopps angeben.

Um die Befehlsebenen-Parallelität besser zu nutzen, unterstützt die IA-64-Architektur die bedingte Befehlsausführung (Prädikation) und die Spekulation in großem Stil (siehe den Abschnitt „Vertiefung“ auf Seite 362). Die **bedingte Befehlsausführung** auch Prädikation (*predication*) ist eine Technik, mit deren Hilfe Sprünge eliminiert werden können, indem die Ausführung eines Befehls von einem Prädikat anstelle einer Verzweigung abhängig gemacht wird. Wie wir bereits gesehen haben, kann die Befehlsebenen-Parallelität bei Sprüngen aufgrund der eingeschränkten Beweglichkeit des Codes nicht optimal genutzt werden. Das Schleifenabrollen eignet sich gut zum Beseitigen von Schleifensprüngen. Ein Sprung innerhalb einer Schleife, der beispielsweise aus einer *If-then-else*-Anweisung entstehen kann, kann jedoch mit dem Schleifenabrollen nicht eliminiert werden. Die bedingte Befehlsausführung stellt dagegen eine Methode zum Eliminieren des Sprungs bereit und ermöglicht so die flexiblere Nutzung der Parallelität.

Gehen wir beispielsweise von einer Codesequenz wie der folgenden aus:

```
if (p) statement 1 else statement 2
```

Befehlsgruppe (instruction group) Bei IA-64 eine Folge von Befehlen ohne Registerdatenabhängigkeiten.

Stopp (stop) Bei IA-64 ein expliziter Indikator für einen Wechsel zwischen unabhängigen und abhängigen Befehlen.

Bedingte Befehlsausführung Auch als Prädikation (*predication*) bezeichnet. Eine Technik, bei der Befehle von Prädikaten statt von Sprüngen abhängig gemacht werden.

Tab. 6.9 Eine Übersicht über die Eigenschaften der Prozessoren Itanium und Itanium 2, die beiden ersten Implementierungen der IA-64-Architektur von Intel. Der Itanium 2 weist neben einer höheren Taktfrequenz und einer größeren Anzahl an Funktionseinheiten im Gegensatz zu dem L3-Cache außerhalb des Prozessorchips (Off-Chip-L3-Cache) beim Itanium einen L3-Cache auf dem Prozessorchip (On-Chip-L3-Cache) auf.

Prozessor	Max. Befehlszuordnung/Takt	Funktionseinheiten	Max. Befehle pro Takt	Max. Taktfrequenz	Transistoren (Mio.)	Leistungsaufnahme (Watt)	SPEC int2000	SPEC fp2000
Itanium	6	4 Ganzzahl-/Medieneinheiten 2 Speichereinheiten 3 Sprungeinheiten 2 Gleitkommaeinheiten	9	0,8 GHz	25	130	379	701
Itanium 2	6	6 Ganzzahl-/Medieneinheiten 4 Speichereinheiten 3 Sprungeinheiten 2 Gleitkommaeinheiten	11	1,5 GHz	221	130	810	1427

Mit normalen Kompiliermethoden würde diese Sequenz so übersetzt, dass 2 Sprünge entstehen: ein Sprung nach der Bedingung zum Else-Teil und ein Sprung nach statement 1 zum nächsten Befehl in Folge. Mit der bedingten Befehlausführung könnte diese Sequenz wie folgt kompiliert werden:

(p) statement 1
(~ p) statement 2

wobei die Verwendung von (Bedingung) angibt, dass die Anweisung nur ausgeführt wird, wenn Bedingung erfüllt ist. Andernfalls wird aus der Anweisung ein NOP-Befehl. Die bedingte Befehlausführung kann als Möglichkeit zum Spekulieren sowie als Methode zum Eliminieren von Sprüngen verwendet werden.

Die IA-64-Architektur stellt eine umfassende Unterstützung der bedingten Befehlausführung bereit: Nahezu jeder Befehl in der IA-64-Architektur kann durch die Angabe eines Prädikatregisters, dessen Bezeichnung in den unteren 6 Bit eines Befehlsfelds gespeichert wird, bedingt ausgeführt werden. Eine Folge der vollständig bedingten Befehlausführung ist die Tatsache, dass ein bedingter Sprung einfach ein Sprung mit einem überwachenden Prädikat ist!

IA-64 ist das komplexeste Beispiel für einen Befehlssatz mit Unterstützung einer compilergestützten Nutzung der Befehlsebenen-Parallelität. Bei den Prozessoren Itanium und Itanium 2 von Intel ist diese Architektur implementiert. Eine kurze Übersicht über die Eigenschaften dieser Prozessoren finden Sie in Tabelle 6.9.



Gift (poison) Ein Ergebnis, das generiert wird, wenn ein spekulativer Ladebefehl eine Unterbrechung verursacht oder wenn ein Befehl einen vergifteten Operanden verwendet.

Vertiefung: Die Unterstützung der Spekulation in der IA-64-Architektur besteht aus einer getrennten Unterstützung der Steuerungsspekulation, bei der es um die Verzögerung von Unterbrechungen für speulierte Befehle geht, und der Speicherspekulation, bei der die Spekulation zu Ladebefehlen unterstützt wird. Die Behandlung von verzögerten Unterbrechungen wird durch zusätzliche spekulative Ladebefehle unterstützt, die das Ergebnis als **Gift (poison)** kennzeichnen, wenn eine Unterbrechung auftritt. Wenn ein vergiftetes Ergebnis von einem Befehl verwendet wird, ist das Ergebnis ebenfalls Gift. Die Software kann dann nach einem vergifteten Ergebnis suchen, wenn die Ausführung nicht mehr spekulativ ist.

In der IA-64-Architektur können wir auch über Speicherzugriffe spekulieren, indem wir Ladebefehle vor Speicherbefehle verschieben, von denen diese möglicherweise abhängig sind. Dies geschieht mithilfe eines vorgezogenen spekulativen

Ladebefehls. Ein **vorgezogener spekulativer Ladebefehl (advanced load)** wird normal ausgeführt, wobei jedoch eine spezielle Tabelle zum Aufzeichnen der Adresse verwendet wird, von der der Prozessor geladen hat. Alle nachfolgenden Speicherbefehle prüfen diese Tabelle und generieren ein Flag im Eintrag, wenn die Speicheradresse mit der Ladeadresse übereinstimmt. Mithilfe eines nachfolgenden Befehls muss der Zustand des Eintrags überprüft werden, sobald der Ladebefehl nicht mehr spekulativ ist. Wenn ein Speicherbefehl auf dieselbe Adresse aufgetreten ist, gibt die Befehlsüberprüfung eine Reparaturroutine an, die den Ladebefehl und andere abhängige Befehle erneut ausführt, bevor die Ausführung fortgesetzt wird. Wenn ein Speicherbefehl dieser Art nicht auftritt, wird der Tabelleneintrag gelöscht und somit angezeigt, dass der Ladebefehl nicht mehr spekulativ ist.

Vorgezogener spekulativer Ladebefehl (advanced load) In der IA-64-Architektur ein spekulativer Ladebefehl mit Unterstützung einer Überprüfung auf Aliase, die den Ladebefehl ungültig machen könnten.

Prozessoren mit dynamischer Mehrfachzuordnung

Prozessoren mit dynamischer Mehrfachzuordnung werden auch als **superskalare (superscalar)** Prozessoren bezeichnet. Bei den einfachsten superskalaren Prozessoren werden Befehle in durch das Programm vorgegebener Reihenfolge (in-order-execution) zugeordnet, und der Prozessor entscheidet, ob kein, ein oder mehrere Befehle in einem gegebenen Taktzyklus zugeordnet werden können. Um mit einem Prozessor dieser Art eine gute Leistung zu erzielen, muss der Compiler versuchen, Befehle so einzurichten, dass Abhängigkeiten aufgelöst werden und damit die Befehlszuordnungsrate verbessert wird. Auch bei einem Compiler-Scheduling dieser Art gibt es einen wichtigen Unterschied zwischen diesem einfachen superskalaren und einem VLIW-Prozessor: Die Hardware garantiert, dass der Code, ob mit oder ohne Scheduling, richtig ausgeführt wird. Zudem wird kompilierter Code immer richtig ausgeführt, unabhängig von der Zuordnungsrate oder der Pipelinestruktur des Prozessors. Bei einigen VLIW-Entwürfen war dies nicht der Fall, und der Code musste für unterschiedliche Prozessormodelle neu kompiliert werden. Bei anderen Prozessoren mit statischer Zuordnung wird der Code in unterschiedlichen Implementierungen zwar richtig ausgeführt, aber häufig so schlecht, dass der Code aus Effizienzgründen neu kompiliert werden muss.

Superskalar (superscalar) Eine erweiterte Pipelining-Technik, mit deren Hilfe der Prozessor mehr als einen Befehl pro Taktzyklus ausführen kann.

Bei vielen superskalaren Prozessoren wird das Grundkonzept der dynamischen Zuordnungsentscheidungen mit **dynamischem Pipeline-Scheduling (dynamic pipeline scheduling)** erweitert. Beim dynamischen Pipeline-Scheduling wird ausgewählt, welche Befehle in einem gegebenen Taktzyklus ausgeführt werden, und gleichzeitig wird versucht, Konflikte und Verzögerungen zu vermeiden. Beginnen wir mit einem einfachen Beispiel zum Vermeiden eines Datenkonflikts. Betrachten wir die folgende Codesequenz:

lw \$t0, 20(\$\$2)	
addu \$t1, \$t0, \$t2	
sub \$s4, \$s4, \$t3	
slti \$t5, \$s4, 20	

Dynamisches Pipeline-Scheduling (dynamic pipeline scheduling) Hardwareunterstützung für die Neuanordnung der Reihenfolge der Befehlausführung zum Verhindern von Verzögerungen.

Obwohl der sub-Befehl zum Ausführen bereit ist, muss er warten, bis die Befehle lw und addu ausgeführt sind, was bei einem langsamen Speicher viele Taktzyhlen beanspruchen kann. (In Kapitel 7 werden Cache-Speicher beschrieben, die eingeführt wurden, weil viele Speicherzugriffe sehr langsam sind.) Mithilfe des dynamischen Pipeline-Scheduling können Konflikte wie diese vollständig oder teilweise vermieden werden.

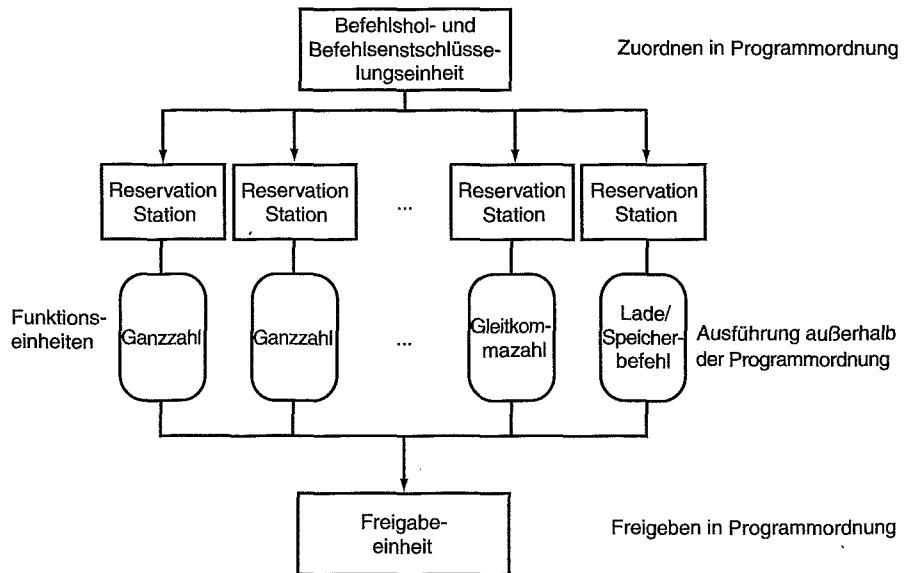


Abb. 6.40 Die drei wichtigsten Einheiten einer Pipeline mit dynamischem Scheduling. Der letzte Schritt, bei dem der Zustand aktualisiert wird, wird auch als Retirement bezeichnet.

Dynamisches Pipeline-Scheduling

Beim dynamischen Pipeline-Scheduling wird festgelegt, welche Befehle als Nächstes ausgeführt werden, wobei die Befehle zum Vermeiden von Verzögerungen möglicherweise neu angeordnet werden. Bei Prozessoren dieser Art ist die Pipeline in drei Haupteinheiten unterteilt: eine Befehlshol- und Befehlszuordnungseinheit, mehrere Funktionseinheiten (bei für das Jahr 2004 typischen Hochleistungsprozessoren 10 oder mehr Funktionseinheiten) und eine **Freigabeeinheit (commit unit)**. In Abbildung 6.40 ist das entsprechende Modell dargestellt. Die erste Einheit holt Befehle, entschlüsselt sie und sendet jeden Befehl zum Ausführen an die entsprechende Funktionseinheit. Jede Funktionseinheit verfügt über Puffer, so genannte **Reservation Stations**, in denen die Operanden und der Befehl gespeichert werden. (Im nächsten Abschnitt beschreiben wir eine Alternative zu den Reservation Stations, die bei vielen neueren Prozessoren zum Einsatz kommt.) Sobald der Puffer alle Operanden enthält und die Funktionseinheit bereit ist, den Befehl auszuführen, wird das Ergebnis berechnet. Wenn das Ergebnis berechnet ist, wird es an alle Reservation Stations, die auf dieses Ergebnis warten, sowie an die Freigabeeinheit gesendet, in der das Ergebnis gespeichert wird, bis es sicher im Registersatz oder, bei einem Speicherbefehl, im Speicher abgelegt werden kann. Der Puffer in der Freigabeeinheit, der häufig als **Rückordnungspuffer (reorder buffer)** bezeichnet wird, wird auch zum Bereitstellen von Operanden verwendet ähnlich wie die Forwarding-Logik in einer Pipeline mit statischem Scheduling. Sobald ein Ergebnis im Registersatz freigegeben ist, kann es wie in jeder normalen Pipeline direkt von dort geholt werden.

Die Kombination aus dem Speichern von Operanden in den Reservation Stations und von Ergebnissen im Rückordnungspuffer stellt eine Form der Registerumbenennung dar, wie die, die vom Compiler in unserem Beispiel für das Schleifenabrollen Seite 359 verwendet wird. Betrachten wir die folgenden Schritte, um die Funktionsweise zu verstehen:

1. Wenn ein Befehl zugeordnet wird und sich einer seiner Operanden im Registersatz oder im Rückordnungspuffer befindet, wird er sofort in die Reservation Station ko-

Freigabeeinheit (*commit unit*)

Die Einheit in einer dynamischen Pipeline oder Pipeline mit Ausführung außerhalb der Programmreihenfolge, die entscheidet, wann es sicher ist, das Ergebnis eines Befehls an für den Programmierer sichtbare Register oder den Speicher freizugeben.

Reservation Station Ein Puffer in einer Funktionseinheit zum Speichern der Operanden und des Befehls.

Rückordnungspuffer (*reorder buffer*) Der Puffer, der Ergebnisse in einem Prozessor mit dynamischem Scheduling speichert, bis es sicher ist, das Ergebnis im Speicher oder in einem Register zu speichern.

piert und dort so lange gespeichert, bis alle Operanden und eine Ausführungseinheit verfügbar sind. Für den zuzuordnenden Befehl wird die Registerkopie des Operanden nicht mehr benötigt, und wenn ein Befehl zum Schreiben in dieses Register auftritt, kann der Wert überschrieben werden.

- Wenn sich ein Operand nicht im Registersatz oder im Rückordnungspuffer befindet, muss er warten, bis er von einer Funktionseinheit generiert wird. Der Name der Funktionseinheit, die das Ergebnis generiert, wird festgehalten. Wenn diese Einheit das Ergebnis generiert, wird dieses aus der Funktionseinheit unter Umgehung der Register direkt in die wartende Reservation Station kopiert.

Bei diesen Schritten werden der Rückordnungspuffer und die Reservation Stations erfolgreich zum Implementieren der Registerumbenennung verwendet.

Prinzipiell können Sie sich eine Pipeline mit dynamischem Scheduling vorstellen, die die Datenflusstruktur eines Programms analysiert, wie wir sie bei der Datenflussanalyse in einem Compiler in Kapitel 2 beschrieben haben. Der Prozessor führt die Befehle dann in einer Reihenfolge aus, mit der die Datenflussordnung des Programms aufrechterhalten wird. Damit sich Programme so verhalten, als würden sie in einer einfachen Pipeline in Programmreihenfolge ausgeführt, muss die Befehlshol- und Befehlsentschlüsselungseinheit Befehle in Programmreihenfolge zuordnen, so dass Abhängigkeiten nachverfolgt werden können, und die Freigabeeinheit muss Ergebnisse in Register und in den Speicher in der Programmausführungsfolge schreiben. Diese konservative Methode wird als *Beenden in Programmreihenfolge* bezeichnet. Wenn also eine Unterbrechung auftritt, kann der Computer auf den zuletzt ausgeführten Befehl zeigen, und es werden nur die Register aktualisiert, in die Befehle vor dem Befehl geschrieben haben, der die Unterbrechung verursacht hat. Auch wenn das Frontend (erste Stufe der Pipeline: Befehl holen und zuordnen) und das Backend (letzte Stufe der Pipeline: Freigabe) der Pipeline in Programmreihenfolge ausgeführt werden, können die Funktionseinheiten die Ausführung beginnen, sobald die erforderlichen Daten verfügbar sind. Heute beenden alle Pipelines mit dynamischem Scheduling Befehle in Programmordnung, auch wenn das nicht immer so war.

Das dynamische Scheduling wird häufig, insbesondere bei Sprungergebnissen, durch eine hardwaregestützte Spekulation erweitert. Durch die Vorhersage der Richtung einer Verzweigung kann ein Prozessor mit dynamischem Scheduling weiter Befehle entsprechend dem vorhergesagten Pfad holen und ausführen. Da die Befehle in **Programmreihenfolge freigegeben (in-order commit)** werden, wissen wir, bevor ein Befehl aus dem vorhergesagten Pfad freigegeben wird, ob der Sprung richtig vorhergesagt wurde. Eine spekulative Pipeline mit dynamischem Scheduling kann auch die Spekulation bezüglich der Adressen von Ladebefehlen unterstützen und ermöglicht so die Umordnung von Lade- und Speicherbefehlen und vermeidet mithilfe der Freigabeeinheit falsche Spekulationen. Im nächsten Abschnitt werden wir uns mit der Verwendung des dynamischen Scheduling und der Spekulation im Pentium-4-Prozessor beschäftigen.

Vertiefung: Eine Freigabeeinheit steuert die Aktualisierung des Registersatzes und des Speichers. Bei einigen Prozessoren mit dynamischem Scheduling wird der Registersatz sofort während der Ausführung aktualisiert. Dazu werden zusätzliche Register verwendet, mit deren Hilfe die Umbenennungsfunktion implementiert und die alte Kopie eines Registers beibehalten wird, bis der Befehl, der das Register aktualisiert, nicht mehr spekulativ ist. Bei anderen Prozessoren wird das Ergebnis in der Regel in einer Struktur gespeichert, die als Rückordnungspuffer bezeichnet wird. Der Registersatz wird erst zu einem späteren Zeitpunkt im Rahmen der Freigabe aktualisiert. Speicherbefehle müssen bis zur Freigabe entweder in einem **Speicherbefehlpuffer** (siehe Kapitel 7) oder im Rückordnungspuffer gespeichert werden. Mithilfe der Freigabeeinheit kann der Speicherbefehl aus dem Puffer in

Freigeben in Programmreihenfolge (in-order commit) Eine Freigabe, bei der alle Ergebnisse der Ausführung mittels Pipeline in die für den Programmierer sichtbaren Register und Speicherstellen, den so genannten Zustand, in derselben Reihenfolge geschrieben werden, in der Befehle geholt wurden.



Ausführung außerhalb der Programmreihenfolge (*out-of-order execution*) Eine Situation bei der Ausführung mittels Pipeline, in der ein Befehl, dessen Ausführung blockiert ist, nachfolgende Befehle nicht zwingt zu warten.



den Speicher schreiben, wenn der Puffer eine gültige Adresse und gültige Daten enthält, und wenn der Speicherbefehl nicht mehr von vorhergesagten Sprüngen abhängt.

Vertiefung: Speicherzugriffe profitieren von *nichtblockierenden Caches*, die Cache-Zugriffe während eines Cache-Fehlzugriffs weiter bearbeiten (siehe Kapitel 7). Prozessoren mit **Ausführung außerhalb der Programmreihenfolge (*out-of-order execution*)** benötigen nichtblockierende Caches, damit Befehle bei einem Fehlzugriff ausgeführt werden können.

Hardware-Software-Schnittstelle

Angesichts der Tatsache, dass Compiler auch Befehle über Datenabhängigkeiten hinweg verschieben können, fragen Sie sich vielleicht, warum bei einem superskalaren Prozessor überhaupt dynamisches Scheduling eingesetzt wird. Hierfür gibt es im Wesentlichen drei Gründe. Erstens sind nicht alle Verzögerungen der Pipeline vorhersagbar. Insbesondere Cache-Fehlzugriffe (siehe Kapitel 7) verursachen nicht vorhersagbare Pipelineverzögerungen. Mithilfe des dynamischen Scheduling kann der Prozessor einige dieser Verzögerungen verbergen, indem er weiter Befehle ausführt, während er auf das Ende der Verzögerung wartet.

Zweitens: Wenn der Prozessor mithilfe der dynamischen Sprungvorhersage über das Ergebnis von Sprüngen spekuliert, kann er die exakte Reihenfolge der Befehle beim Kompilieren nicht kennen, da diese vom vorhergesagten und tatsächlichen Verhalten von Sprüngen abhängt. Wenn die dynamische Spekulation zur besseren Nutzung der Befehlsebenen-Parallelität ohne dynamisches Scheduling integriert wird, werden dadurch die Vorteile einer derartigen Spekulation erheblich geschmälert.

Drittens: Da die Pipelinelatenz und die Parallelität bei der Zuordnung von Implementierung zu Implementierung unterschiedlich sind, unterscheidet sich auch die jeweils beste Möglichkeit, eine Codesequenz zu kompilieren. Die Art und Weise, wie eine Folge von abhängigen Befehlen angeordnet wird, hängt beispielsweise sowohl von der Zuordnungsparallelität als auch von der Latenz ab. Die Pipelinestruktur wirkt sich sowohl darauf aus, wie oft eine Schleife abgerollt werden muss, um ein Leerlaufen der Pipeline zu vermeiden, als auch auf den Prozess der compilerbasierten Registerumbenennung. Mithilfe des dynamischen Scheduling kann die Hardware einen Großteil dieser Details verbergen. Somit benötigen Benutzer und Softwarehändler für unterschiedliche Implementierungen desselben Befehlssatzes keine unterschiedlichen Versionen eines Programms. Entsprechend können auch ältere Programme die Vorteile einer neuen Implementierung nutzen, ohne neu kompiliert werden zu müssen.



Sowohl das Pipelining als auch die Ausführung mit Mehrfachzuordnung erhöhen den maximalen Befehlsdurchsatz und versuchen, die Befehlsebenen-Parallelität zu nutzen. Daten- und Kontrollflussabhängigkeiten in Programmen stellen jedoch eine obere Grenze für eine dauerhafte Spitzenleistung dar, da der Prozessor gelegentlich warten muss, bis eine Abhängigkeit aufgelöst ist. Softwareorientierte Konzepte für die Nutzung der Befehlsebenen-Parallelität hängen davon ab, ob der Compiler die Auswirkungen von Abhängigkeiten dieser Art erkennt und reduziert kann, während hardwareorientierte Konzepte auf die Erweiterung der Pipeline und der Zuordnungsmechanismen setzen. Vom Compiler oder von der Hardware angestellte Spekulationen können dazu beitragen, dass die Befehlsebenen-Parallelität besser genutzt werden kann, wobei jedoch mit Sorgfalt vorgegangen werden muss, da eine falsche Spekulation zu einer Beeinträchtigung der Leistung führen kann.

Moderne Hochleistungsmikroprozessoren können pro Taktzyklus mehrere Befehle zuordnen, haben jedoch Schwierigkeiten, diese Zuordnungsrate permanent aufrechtzuhalten. Obwohl es beispielsweise Prozessoren gibt, die vier bis sechs Zuordnungen pro Taktzyklus vornehmen, können nur sehr wenige Anwendungen im Schnitt mehr als zwei Befehle pro Taktzyklus über die gesamte Laufzeit des Programms ausführen. Hierfür gibt es im Wesentlichen zwei Gründe.

Zum einen entstehen die größten Leistungsengpässe in der Pipeline aufgrund von Abhängigkeiten, die nicht aufgelöst werden können, so dass die Parallelität von Befehlen und die für Programme durchschnittlich nutzbare Zuordnungsrate beeinträchtigt wird. Zwar kann gegen echte Datenabhängigkeiten nur wenig unternommen werden, doch erkennt der Compiler oder die Hardware häufig nicht genau, ob eine Abhängigkeit vorliegt, und muss daher vorsichtshalber davon ausgehen, dass dies der Fall ist. So führt ein Programm, das Zeiger insbesondere auf eine Art und Weise verwendet, die vermehrtes Aliasing zur Folge hat, vermehrt zu impliziten potenziellen Abhängigkeiten. Im Gegensatz dazu kann ein Compiler aufgrund der größeren Regelmäßigkeit von Zugriffen auf Felder häufig ableiten, dass keine Abhängigkeiten vorliegen. In ähnlicher Weise begrenzen Sprünge, die weder zur Laufzeit noch beim Kompilieren exakt vorhergesagt werden können, die Nutzung der Befehlsebenen-Parallelität. Häufig könnte die Befehlsebenen-Parallelität besser genutzt werden, aber die Fähigkeit des Compilers oder der Hardware, die manchmal über die Ausführung von Tausenden von Befehlen weit verstreute Befehlsebenen-Parallelität zu erkennen, ist begrenzt.

Zum anderen begrenzen Verluste im Speichersystem (das Thema von Kapitel 7) die Fähigkeit, die Pipeline nicht leer laufen zu lassen. Einige durch das Speichersystem verursachte Leerläufe können verborgen werden, aber ein eingeschränktes Maß an Befehlsebenen-Parallelität schränkt auch das Maß ein, zu dem Leerläufe dieser Art verborgen werden können.

Geben Sie an, ob die folgenden Techniken oder Komponenten in erster Linie einem softwaregestützten oder einem hardwaregestützten Konzept zur Nutzung der Befehlsebenen-Parallelität zuzuordnen sind. Bei einigen Techniken und Komponenten können auch beide Konzepte angegeben werden.

Grundlegendes zur Leistungsfähigkeit von Programmen

1. Sprungvorhersage
2. Mehrfachzuordnung
3. VLIW-Prozessor
4. Superskalarer Prozessor
5. Dynamisches Scheduling
6. Ausführung außerhalb der Programmreihenfolge
7. Spekulation
8. EPIC-Prozessor
9. Rückordnungspuffer
10. Registerumbenennung
11. Bedingte Befehlsausführung



6.10

Fallstudie: Die Pentium-4-Pipeline

Im letzten Kapitel haben wir beschrieben, wie der Pentium 4 IA-32-Befehle holt und in micro-operations übersetzt. Diese micro-operations werden dann von einer komplexen, spekulativen Pipeline mit dynamischem Scheduling ausgeführt, die eine Ausführungsrate von drei micro-operations pro Taktzyklus aufrechterhalten kann. In diesem Abschnitt geht es um genau diese Pipeline. Im Pentium 4 ist die Mehrfachzuordnung mit vielstufigem Pipelining kombiniert, um sowohl einen niedrigen CPI-Wert als auch eine hohe Taktrate zu erzielen.

Wenn wir den Entwurf von komplexen Prozessoren mit dynamischem Scheduling betrachten, vermischt sich der Entwurf der Funktionseinheiten, der Caches und des Registersatzes, der Befehlszuordnung und der gesamten Pipelinesteuerung, wodurch

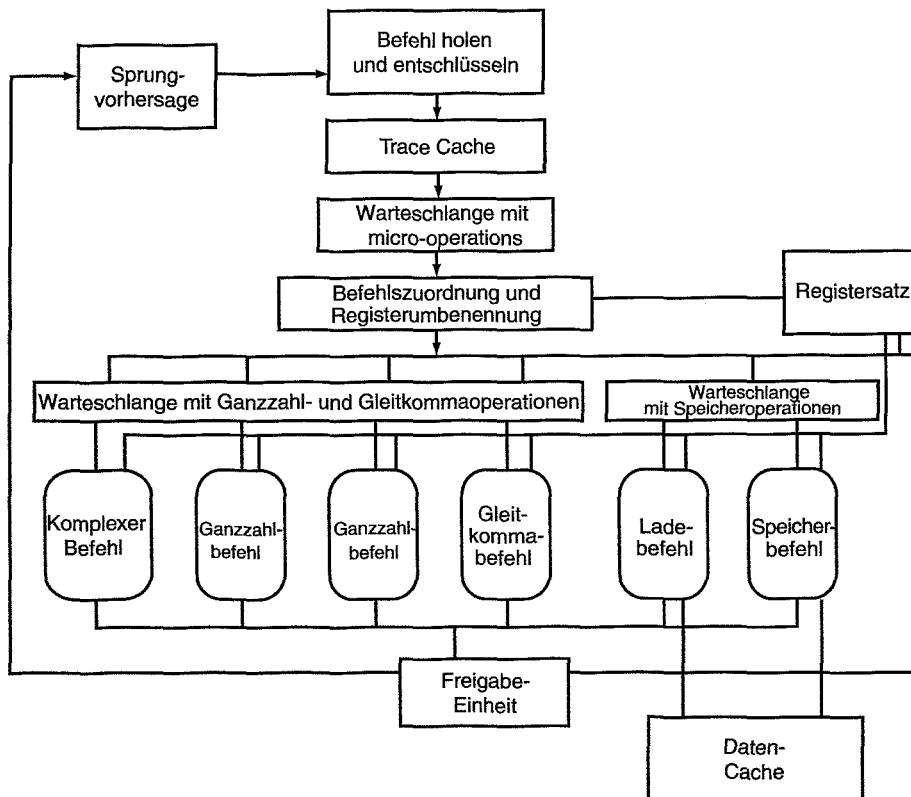


Abb. 6.41 Mikroarchitektur des Intel Pentium 4. Dank der umfangreichen Warteschlangen können jederzeit bis zu 126 micro-operations – darunter 48 Ladebefehle und 24 Speicherbefehle – zur Bearbeitung anstehen. Es gibt sieben Funktionseinheiten, da die Gleitkommaeinheit eine eigene Einheit speziell für Gleitkommaschiebebefehle enthält. Die Lade- und Speichereinheiten bestehen aus zwei Teilen, wobei der eine Teil für die Adressberechnung und der zweite Teil für den Speicherzugriff zuständig ist. Die Ganzzahl-ALUs arbeiten mit der doppelten Taktfrequenz, so dass die beiden Ganzzahleinheiten in einem Taktzyklus zwei ALU-Ganzzahloperationen durchführen können. Wie in Kapitel 5 bereits beschrieben, verwendet der Pentium 4 einen speziellen Cache, der als Trace Cache bezeichnet wird und zum Speichern von vorab entschlüsselten Folgen von micro-operations, die IA-32-Befehlen entsprechen, verwendet wird. Die Funktionsweise eines Trace Cache wird in Kapitel 7 ausführlicher beschrieben. Die Gleitkommaeinheit verarbeitet außerdem die MMX-Multimedia- und SSE2-Befehle. Zwischen den Funktionseinheiten gibt es ein ausgeprägtes Bypassnetzwerk. Da die Pipeline nicht statisch, sondern dynamisch arbeitet, erfolgt die Umleitung durch Kennzeichnung der Ergebnisse und Nachverfolgen der Quelloperanden, so dass ein entsprechender Abgleich durchgeführt werden kann, wenn für einen Befehl in einer der Warteschlangen, der das Ergebnis benötigt, ein Ergebnis generiert wird. Von Intel wird erwartet, dass später neue Versionen des Pentium 4 mit einer veränderten Mikroarchitektur auf den Markt kommen.

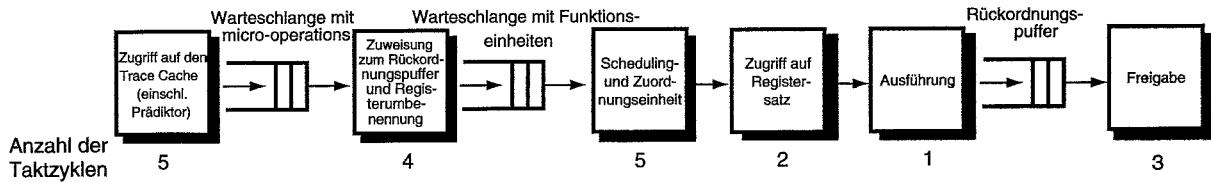


Abb. 6.42 Pentium-4-Pipeline mit dem Pfad durch die Pipeline für einen typischen Befehl und Anzahl der Taktzyklen für die wichtigsten Schritte in der Pipeline. Außerdem sind die wichtigsten Puffer dargestellt, in denen Befehle warten.

es schwierig wird, den Datenpfad von der Pipeline zu trennen. Aus diesem Grund verwenden viele Ingenieure und Forscher für die detaillierte interne Architektur eines Prozessors den Ausdruck **Mikroarchitektur (microarchitecture)**. In Abbildung 6.41 ist die Mikroarchitektur des Pentium 4 mit den Strukturen für die Ausführung der micro-operations dargestellt.

Die Pipelinestufen, die ein typischer Befehl durchläuft, stellen eine andere Möglichkeit dar, den Pentium 4 zu betrachten. In Abbildung 6.42 ist die Pipelinestruktur und die Anzahl der Taktzyklen dargestellt, die ein Befehl in den einzelnen Stufen üblicherweise verbringt. Die Anzahl der Taktzyklen variiert allerdings je nach der Art des dynamischen Scheduling sowie nach den Anforderungen der einzelnen micro-operations.

Beim Pentium 4 und bei dessen Vorgängern Pentium III und Pentium Pro wurde die Technik eingesetzt, die IA-32-Befehle in micro-operations zu übersetzen und diese micro-operations dann mithilfe einer spekulativen Pipeline mit mehreren Funktionseinheiten auszuführen. Die Mikroarchitektur ist im Grundsatz ähnlich, und alle diese Prozessoren können pro Taktzyklus bis zu drei micro-operations ausführen. Der Pentium 4 weist jedoch gegenüber dem Pentium III aufgrund der folgenden Erweiterungen eine bessere Leistung auf:

1. Eine Pipeline, die etwa doppelt so viele Stufen hat (etwa 20 Taktzyklen statt 10) und mit derselben Technologie nahezu doppelt so schnell ist
2. Mehr Funktionseinheiten (7 statt 5)
3. Unterstützung einer größeren Anzahl an zur Ausführung anstehenden Operationen (126 statt 40)
4. Verwendung eines Trace Cache (siehe Kapitel 7) und eines wesentlich verbesserten Sprung-Prädiktors (4000 Einträge statt 512)
5. Andere Erweiterungen des Speichersystems, die in Kapitel 7 beschrieben werden

Vertiefung: Der Pentium 4 verwendet eine Methode zum Auflösen von Antiabhängigkeiten und falschen Spekulationen, für die ein Rückordnungspuffer zusammen mit der Registerumbenennung verwendet wird. Bei der Registerumbenennung werden die für den Programmierer sichtbaren **Architekturregister (architectural registers)** (acht bei einem IA-32-Prozessor) in einen größeren Satz physikalischer Register (128 beim Pentium 4) explizit umbenannt. Der Pentium 4 löst mithilfe der Registerumbenennung Antiabhängigkeiten auf. Für die Registerumbenennung muss der Prozessor eine Zuordnungstabelle verwalten, aus der die Zuordnung der Architekturregister zu den physikalischen Registern ersichtlich wird, und die angibt, welches physikalische Register die aktuelle Kopie eines Architekturregisters darstellt. Durch das Nachverfolgen der erfolgten Umbenennungen stellt die Registerumbenennung ein weiteres Konzept zur Wiederherstellung im Fall einer falschen Spekulation dar: Es müssen einfach nur die seit dem ersten falsch spekulierten Befehl erfolgten Zuordnungen rückgängig gemacht werden. Dadurch wird

Mikroarchitektur (microarchitecture) Organisation des Prozessors, die die wichtigsten Funktionseinheiten, deren Verbindungen untereinander und deren Steuerung beinhaltet.



Architekturregister (architectural registers) Für den Befehlssatz sichtbare Register eines Prozessors. Bei MIPS sind das beispielsweise die 32 Ganzzahl- und 16 Gleitkommaregister.

der Zustand des Prozessors zum letzten richtig ausgeführten Befehl zurückgesetzt, wobei die richtige Zuordnung zwischen den Architekturregistern und physikalischen Registern beibehalten wird.

Grundlegendes zur Leistungs- fähigkeit von Programmen

Beim Pentium 4 wird eine vielstufige Pipeline (durchschnittlich 20 oder mehr Pipelinestufen pro Befehl) zusammen mit einer aggressiven Mehrfachzuordnung verwendet, um eine hohe Leistungsfähigkeit zu erzielen. Dadurch, dass die Latenzen für Back-to-Back-Befehle gering gehalten werden (0 für ALU-Operationen und 2 für Ladebefehle), werden die Auswirkungen von Datenabhängigkeiten reduziert. Welches sind die schwerwiegendsten potenziellen Leistungsengpässe für Programme, die auf diesem Prozessor ausgeführt werden? Die folgende Liste enthält einige potenzielle Leistungsengpässe, von denen die letzten drei in der einen oder anderen Form jeden Hochleistungsprozessor mit Pipelining betreffen können.

- Die Verwendung von IA-32-Befehlen, die nicht auf drei oder weniger micro-operations abgebildet werden können
- Sprünge, die sich nur schwer vorhersagen lassen und die ein Leerlaufen der Pipeline aufgrund falscher Vorhersagen und einen Wiederanlauf der Pipeline bei falschen Spekulationen verursachen
- Geringe Befehlslokalität, aufgrund derer der Trace Cache nicht effizient arbeitet
- Lange Abhängigkeiten (in der Regel durch Befehle mit langen Ausführungszeiten oder durch einen Fehlzugriff auf den Daten-Cache verursacht), die ein Leerlaufen der Pipeline verursachen
- Verzögerungen beim Zugriff auf den Speicher (siehe Kapitel 7), die dazu führen, dass der Prozessor die Pipeline blockiert



Sind die folgenden Aussagen richtig oder falsch?

1. Der Pentium 4 kann pro Taktzyklus mehr Befehle zuordnen als der Pentium III.
2. Die Pipeline mit Mehrfachzuordnung des Pentium 4 führt IA-32-Befehle direkt aus.
3. Der Pentium 4 verwendet dynamisches Scheduling, aber keine Spekulation.
4. Die Pentium-4-Mikroarchitektur weist wesentlich mehr Register auf, als für IA-32 erforderlich.
5. Die Pentium-4-Pipeline weist weniger Stufen auf als die des Pentium III.
6. Der Trace Cache im Pentium 4 ist exakt dasselbe wie ein Befehlscache.

6.11 Fallstricke und Fehlschlüsse

Fehlschluss: Pipelining ist einfach.

Anhand unserer Bücher werden die Feinheiten einer einwandfreien Pipelineausführung deutlich. In unserem Buch für Fortgeschrittene befand sich in der ersten Auflage ein Pipelinefehler, obwohl das Buch von mehr als 100 Personen geprüft und in 18 Universitäten in den Vorlesungen verwendet wurde. Der Fehler wurde erst entdeckt, als jemand versuchte, den Computer aus diesem Buch zu bauen. Die Tatsache, dass der Verilog-Code zum Beschreiben einer Pipeline wie der im Pentium 4 Tausende von Zeilen umfasst, ist ein Hinweis auf die Komplexität einer Pipeline. Hier ist also Vorsicht geboten!

Fehlschluss: Pipelining-Konzepte können unabhängig von der Technologie implementiert werden.

Als eine fünfstufige Pipeline aufgrund der Anzahl der Transistoren auf dem Chip und der Geschwindigkeit der Transistoren die beste Lösung darstellte, war der verzögerte Sprung (siehe den Abschnitt „Vertiefung“ auf Seite 345) die einfachste Lösung zur Bearbeitung von Konflikten. Angesichts der längeren Pipelines, der superskalaren Ausführung und der dynamischen Sprungvorhersage ist dies nun nicht mehr der Fall. Anfang der 90er-Jahre beanspruchte das dynamische Pipeline-Scheduling zu viele Ressourcen und war für hohe Leistung nicht gefragt. Als sich die Anzahl der Transistoren jedoch weiter erhöhte und Logik wesentlich schneller als Speicher wurde, wurden Multifunktionseinheiten und dynamisches Pipelining entsprechend sinnvoller. Heute arbeiten alle Prozessoren der oberen Preisklasse mit Mehrfachzuordnung und bei den meisten Prozessoren ist außerdem eine aggressive Spekulationsmethode implementiert.

Fallstrick: Nicht bedenken, dass der Entwurf eines Befehlssatzes negative Auswirkungen auf das Pipelining haben kann.

Viele der Schwierigkeiten beim Pipelining entstehen aufgrund der Kompliziertheit des Befehlssatzes. Im Folgenden einige Beispiele hierzu:

- Stark unterschiedliche Befehlsformate und Ausführungszeiten können zu einer Unausgewogenheit bei den Pipelinestufen führen und die Erkennung von Konflikten in einem Entwurf, bei dem sich das Pipelining auf der Ebene des Befehlssatzes abspielt, erheblich erschweren. Dieses Problem wurde zum ersten Mal im DEC VAX 8500 Ende der 80er-Jahre mithilfe der Mikropipelinemethode gelöst, die heute im Pentium 4 verwendet wird. Der Aufwand, der durch die Übersetzung und Aufrechterhaltung der Kommunikation zwischen den Microoperations und den eigentlichen Befehlen entsteht, bleibt allerdings bestehen.
- Komplexe Adressierungsarten können unterschiedliche Probleme verursachen. Adressierungsarten, die wie die Aktualisierungsadressierung (siehe Kapitel 3) Register aktualisieren, erschweren die Erkennung von Konflikten. Andere Adressierungsarten, die mehrere Speicherzugriffe erfordern, erschweren die Steuerung und ein reibungsloses beständiges Arbeiten der Pipeline.

Das vielleicht beste Beispiel ist der DEC Alpha und der DEC NVAX. Bei vergleichbarer Technologie ermöglicht die neuere Befehlsarchitektur des Alpha eine Implementierung, die doppelt so schnell ist wie die der NVAX. In einem anderen Beispiel haben Bhandarkar und Clark [1991] den MIPS M/2000 und die DEC VAX 8700 miteinander

*Nine-tenths of wisdom consists
of being wise in time.*
Amerikanisches Sprichwort

6.12 Schlussbetrachtungen

Das Pipelining verbessert die durchschnittliche Ausführungszeit pro Befehl. Je nachdem, ob Sie mit einem Eintakt- oder Mehrzyklendatenpfad beginnen, können Sie sich diese Verbesserung in Form einer Reduzierung der Zykluszeit für den Takt oder in Form einer Reduzierung der Anzahl der Taktzyklen pro Befehl (CPI-Wert) vorstellen. Wir haben mit dem einfachen Eintaktdatenpfad begonnen, so dass sich das Pipelining als Reduzierung der Zykluszeit für den Takt des einfachen Datenpfads darstellte. Bei der Mehrfachzuordnung liegt der Fokus dagegen eindeutig auf der Reduzierung des CPI-Werts (bzw. auf der Erhöhung des IPC-Werts). In Abbildung 6.43 sind die Auswirkungen auf den CPI-Wert und die Taktrate für die einzelnen Mikroarchitekturen aus Kapitel 5 und 6 dargestellt. Die Leistung ist rechts oben am besten, da sie das Produkt aus IPC-Wert und Taktrate ist, wodurch die Leistung eines gegebenen Befehlssatzes bestimmt wird.

Das Pipelining verbessert den Durchsatz, aber nicht die eigentliche Ausführungszeit oder *Latenz* von Befehlen. Die Dauer der Latenz ist ähnlich wie beim Mehrzyklendatenpfad. Im Gegensatz zu diesem Konzept, bei dem während der Befehlausführung dieselbe Hardware wiederholte Male verwendet wird, beginnt beim Pipelining mit jedem Taktzyklus die Ausführung eines Befehls, da die entsprechende Hardware als getrennte

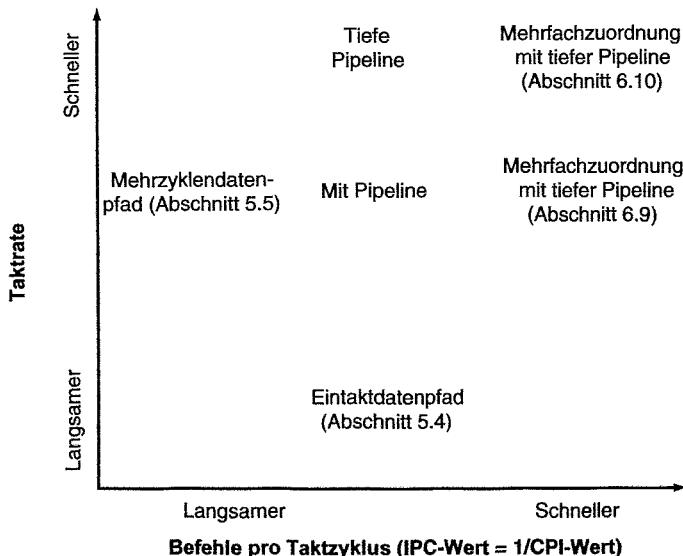


Abb. 6.43 Auswirkungen auf die Leistung des einfachen (Eintakt-)Datenpfads und des Mehrzyklendatenpfads aus Kapitel 5 und das Modell der Ausführung mit Pipeline in Kapitel 6. Die Prozessorleistung ist abhängig vom Produkt aus IPC-Wert und Taktrate. Daher ist die Leistung rechts oben in der Abbildung am besten. Obwohl die Anzahl der Befehle pro Taktzyklus beim einfachen Datenpfad etwas höher ist, ist die Leistung beim Datenpfad mit Pipeline ähnlich hoch und dieser Datenpfad weist zudem eine Taktrate auf, die nahezu so hoch wie die beim Mehrzyklendatenpfad ist.

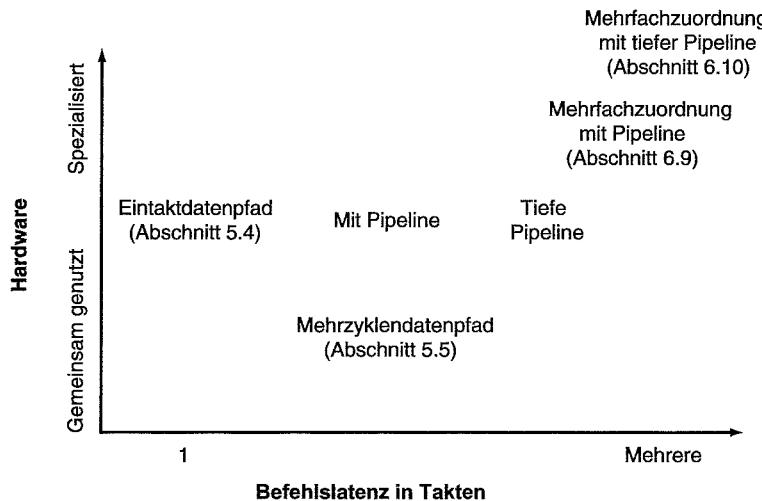


Abb. 6.44 Grundsätzliches Verhältnis der Datenpfade aus Abbildung 6.43. An der x -Achse ist die Befehlslatenz abgetragen, die angibt, wie schwierig es ist, dafür zu sorgen, dass die Pipeline nicht leer läuft. Der Datenpfad mit Pipelining ist als Mehrzyklandatenpfad für die Befehlslatenz dargestellt, da die Ausführungszeit eines Befehls nicht kürzer ist. Es ist der Befehlsdurchsatz, der verbessert wird.

Einheit jeweils zur Verfügung steht. Ähnlich wird die Hardware bei der Mehrfachzuordnung um zusätzliche Datenpfadhardware erweitert, damit mit jedem Taktzyklus die Ausführung mehrerer Befehle begonnen werden kann, jedoch mit einer Zunahme der effektiven Latenz. In Abbildung 6.44 sind die Datenpfade aus Abbildung 6.43 entsprechend dem Anteil an gemeinsam verwandelter Hardware und der **Befehlslatenz (instruction latency)** angeordnet dargestellt.

Sowohl das Pipelining als auch die Mehrfachzuordnung versuchen die Befehlsebenen-Parallelität zu nutzen. Daten- und Kontrollflussabhängigkeiten, aus denen Konflikte entstehen können, sind die wichtigsten Einschränkungen im Hinblick darauf, wie stark die Parallelität genutzt werden kann. Das Scheduling und die Spekulation stellen sowohl in der Hardware als auch in der Software die wichtigsten Methoden zur Reduktion der Leistungsbeeinträchtigung durch Abhängigkeiten dar.

Der Wechsel hin zu längeren Pipelines, Mehrfachzuordnung von Befehlen und dynamischem Scheduling Mitte der 90er-Jahre hat dazu beigetragen, dass die Rechenleistung von Prozessoren pro Jahr wie seit den frühen 80er-Jahren weiterhin um 60% zunahm. In der Vergangenheit musste man sich zwischen Prozessoren mit der höchsten Taktrate und den komplexesten superskalaren Prozessoren entscheiden. Wie wir gesehen haben, vereint der Pentium 4 beides und erzielt damit eine bemerkenswerte Rechenleistung.

Nach dem Gesetz von Amdahl wird nach der enormen Leistungssteigerung bei den Prozessoren ein anderer Bereich des Systems zum Engpass. Dieser Engpass ist das Thema des nächsten Kapitels: das Speichersystem.

Statt bei Uniprozessoren die automatische Nutzung der Parallelität auf der Befehls Ebene weiter zu steigern, kann als Alternative natürlich versucht werden, Multiprozessoren einzusetzen, die die Parallelität auf deutlich größeren Ebenen nutzen. Die parallele Verarbeitung ist das Thema von **Kapitel 9**.

Befehlslatenz (instruction latency) Die Ausführungsdauer für einen einzelnen Befehl.