

trifft. Spezielle Hardware und entsprechende Mechanismen sind erforderlich, um das Durcheinander zu beseitigen und auf den richtigen Weg zurückzukehren. Dies ginge aber über den Rahmen dieses Buchs hinaus. Als Edsger Dijkstra seinen berühmten Brief „GOTO Statement Considered Harmful“ schrieb [Dijkstra, 1968a], hatte er keine Ahnung, wie recht er haben sollte.

Zwischen der Mic-1 und der Mic-4 liegt ein langer Weg. Die Mic-1 ist eine sehr einfache Hardware, bei der die Software fast die gesamte Steuerung übernimmt. Die Mic-4 ist stark fließbandorientiert mit sieben Stufen und einer weitaus komplexeren Hardware. ►Abbildung 4.24 stellt die Pipeline schematisch dar, wobei die eingekreisten Zahlen auf die Komponenten in Abbildung 4.23 verweisen. Die Mic-4 liest automatisch einen Bytestrom vorab aus dem Speicher ein, decodiert ihn in IJVM-Befehle, wandelt diese mithilfe eines ROMs in eine Sequenz von Mikrooperationen um und stellt sie abrufbereit in Warteschlangen. Die ersten drei Stufen der Pipeline lassen sich bei Bedarf an den Datenpfadakt koppeln. Es steht aber nicht immer Arbeit an. Beispielsweise kann die IFU zweifellos nicht in jedem Taktzyklus einen neuen IJVM-Opcode in die Decodier-einheit einspeisen, weil die Ausführung von IJVM-Befehlen mehrere Zyklen dauert und die Warteschlange schnell überlaufen würde.

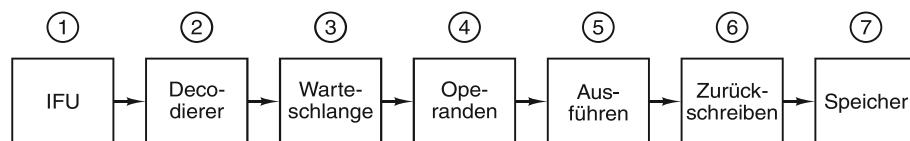


Abbildung 4.24: Die Pipeline der Mic-4

In jedem Taktzyklus werden die MIR-Register vorwärts geschoben und die unten aus der Warteschlange heraustrretende Mikrooperation kommt nach MIR1, um ihre Ausführung zu starten. Dann laufen die Steuersignale von den vier MIR-Registern durch den Datenpfad und lösen bestimmte Aktionen aus. Jedes MIR-Register steuert einen anderen Abschnitt des Datenpfads und somit verschiedene Mikroschritte.

Bei einer fließbandorientierten CPU sind die einzelnen Verarbeitungsschritte sehr kurz, was neben dem höheren Durchsatz eine **höhere Taktfrequenz** ermöglicht.

Dieses Konzept realisiert eine stark fließbandorientierte CPU, bei der die einzelnen Schritte sehr kurz sein können und folglich die Taktfrequenz hoch sein kann. Viele CPUs basieren im Wesentlichen auf diesem Konzept, insbesondere jene, die einen älteren Befehlssatz (CISC) implementieren müssen. Beispielsweise ist die Core-i7-Implementierung konzeptionell in gewisser Weise mit der Mic-4 vergleichbar, wie wir später in diesem Kapitel noch sehen werden.

4.5 Leistungsverbesserung

Alle Computerhersteller möchten, dass ihre Systeme so schnell wie möglich laufen. In diesem Abschnitt betrachten wir eine Reihe fortgeschrittener Techniken, die man derzeit untersucht, um die Systemleistung (vorwiegend von CPU und Speicher) zu verbessern. Durch die starke Konkurrenz in der Computerindustrie ist die zeitliche Verzögerung zwischen neuen Forschungsideen, die einen Computer beschleunigen können, und deren Umsetzung in konkrete Produkte überraschend kurz. Folglich sind die meisten Ideen, die wir hier behandeln, bereits bei einer Vielzahl von Produkten realisiert.

Die hier behandelten Konzepte lassen sich grob zwei Kategorien zuordnen: Verbesserung der Implementierung und Verbesserung der Architektur. Implementierungsverbesserungen sind Möglichkeiten, eine neue CPU oder einen Speicher zu entwickeln, damit das System ohne Änderung der Architektur schneller läuft. Eine entsprechende Änderung der Implementierung bedeutet, dass alte Programme auf dem neuen Computer laufen – ein starkes Verkaufsargument. Beispielsweise kann man die Implementierung durch eine höhere Taktfrequenz verbessern, doch es geht auch anders. Leistungsgewinne vom 80386 über den 80486, Pentium und spätere Konzepte wie beim Core i7 sind auf bessere Implementierungen zurückzuführen, da die Architektur im Wesentlichen durch alle Modelle hindurch gleich geblieben ist.

Einige Verbesserungen lassen sich nur durch Veränderung der Architektur realisieren. Manchmal verlaufen diese Änderungen stufenweise, z.B. durch Hinzufügen neuer Befehle oder Register, sodass alte Programme weiterhin auf den neuen Modellen laufen. Um in diesem Fall die Leistung voll auszuschöpfen, muss man die Software anpassen oder zumindest mit einem neuen Compiler, der die neuen Merkmale nutzt, neu kompilieren.

Alle Jubeljahre sehen auch die Designer ein, dass die alte Architektur ausgedient hat und weiterer Fortschritt nur möglich ist, wenn man vollkommen neue Wege beschreitet. Die RISC-Revolution in den 1980er Jahren war ein solcher Durchbruch. Ein weiterer liegt derzeit in der Luft. *Kapitel 5* zeigt dafür ein Beispiel (die Intel IA-64).

Im Rest dieses Abschnitts sehen wir uns vier unterschiedliche Verfahren an, um die CPU-Leistung zu verbessern. Wir beginnen mit drei etablierten Implementierungsverbesserungen und gehen dann zu einer über, die eine wenig Unterstützung von der Architektur braucht, um optimal zu funktionieren: Cache-Speicher, Sprungvorhersage, Out-of-Order-Ausführung (Umstellen von Befehlen in der Pipeline) mit Registerumbenennung und spekulative Ausführung.

4.5.1 Cache-Speicher

Seit Anbeginn gehört es wohl zu den größten Herausforderungen im Computerdesign, Speichersysteme zu schaffen, die den Prozessor mit Operanden in einer Geschwindigkeit versorgen, mit der er sie verarbeiten kann. Der jüngste Geschwindigkeitszuwachs bei Prozessoren wird nicht von einer entsprechenden Beschleunigung der Speicher begleitet. Im Verhältnis zu CPUs sind Speicher im Laufe von Jahrzehnten langsamer geworden. Angesichts der enormen Bedeutung von Hauptspeicher hat dies die Entwicklung von Hochleistungssystemen stark eingeschränkt und Forschungsarbeiten angeregt, um das Problem zu umgehen, dass Speicher viel langsamer sind als CPUs, was sich – relativ gesehen – jedes Jahr verschlimmert.

Moderne Prozessoren stellen äußerst hohe Ansprüche an ein Speichersystem, sowohl hinsichtlich der Latenzzeit (der Verzögerung bei der Bereitstellung eines Operanden) als auch hinsichtlich der Bandbreite (der Datenmenge, die pro Zeiteinheit bereitgestellt wird). Leider stehen diese beiden Aspekte eines Speichersystems in starkem Konflikt zueinander. Viele Techniken zur Steigerung der Bandbreite realisieren das nur durch Erhöhen der Latenzzeit. Beispielsweise lassen sich die in der Mic-3 eingesetzten Fließbandverfahren auf ein Speichersystem anwenden, das mehrere sich überlappende Speicheranforderungen effizient behandeln kann. Leider führt dies wie bei der Mic-3 zu einer höheren Latenz für einzelne Speicheroperationen. Je stärker die Taktgeschwindigkeit des Prozessors zunimmt, umso schwieriger lässt sich ein Speichersystem realisieren, das Operanden in einem oder zwei Taktzyklen bereitstellen kann.

Caches bieten eine Möglichkeit, dieses Problem anzugehen. Wie *Abschnitt 2.2.5* gezeigt hat, nimmt ein Cache die zuletzt benutzten Speicherwörter in einem kleinen, schnellen Speicher auf, sodass sich der Zugriff darauf beschleunigt. Befindet sich ein ausreichend großer Anteil der benötigten Speicherwörter im Cache, so verringert sich die effektive Speicherlatenz enorm.

Caches verringern die Latenz von Speicherzugriffen und erhöhen die Bandbreite. Cachezeilen werden hierzu für weitere Zugriffe zwischengespeichert. Aktuelle Prozessoren verwenden eine Hierarchie von Caches sowie getrennte Befehls- und Datencaches.

Um sowohl Bandbreite als auch Latenz zu verbessern, ist es am wirksamsten, mehrere Caches einzusetzen. Ein sehr effizient arbeitendes grundlegendes Verfahren führt einen separaten Cache für Befehle und Daten ein. Dieser geteilte Cache – der sogenannte **Split-Cache** – bietet mehrere Vorteile. Erstens lassen sich Speicheroperationen unabhängig voneinander in jedem Cache einleiten, sodass sich die Bandbreite des Speichersystems praktisch verdoppelt. Deshalb ist es sinnvoll, zwei getrennte Speicherports bereitzustellen, wie es die Mic-1 gezeigt hat: Jeder Port hat seinen eigenen Cache. Beachten Sie, dass jeder Cache unabhängig von den anderen auf den Hauptspeicher zugreifen kann.

Moderne Speichersysteme sind wesentlich komplizierter. Ein zusätzlicher Cache – der **Level-2-Cache** (auch L2-Cache) – kann zwischen dem Befehls- und dem Datencache auf der einen Seite und dem Hauptspeicher auf der anderen Seite angesiedelt sein. Möglicherweise gibt es drei oder mehr Cache-Ebenen, je nachdem, wie ausgefeilt die Speichersysteme sein müssen. ►Abbildung 4.25 zeigt ein System mit drei Cache-Ebenen. Der CPU-Chip selbst enthält einen kleinen Befehls- und einen kleinen Datencache, normalerweise in der Größe von 16 KB bis 64 KB. Dann gibt es den Level-2-Cache, der sich nicht auf dem CPU-Chip befindet, aber im CPU-Gehäuse neben dem CPU-Chip untergebracht und mit diesem über einen Hochgeschwindigkeitspfad verbunden sein kann. Dabei handelt es sich normalerweise um einen gemeinsamen (unified) Cache, der eine Mischung aus Daten und Befehlen enthält. Ein L2-Cache ist typischerweise 512 KB bis 1 MB groß. Der Cache der dritten Ebene befindet sich auf der Prozessorkarte und besteht aus einigen Megabyte SRAM, einem Speichertyp, der viel schneller als der DRAM-Hauptspeicher ist. Caches sind im Allgemeinen umfassend, d.h., der gesamte Inhalt des Level-1-Cache ist im Level-2-Cache und der gesamte Inhalt des Level-2-Cache im Level-3-Cache enthalten.

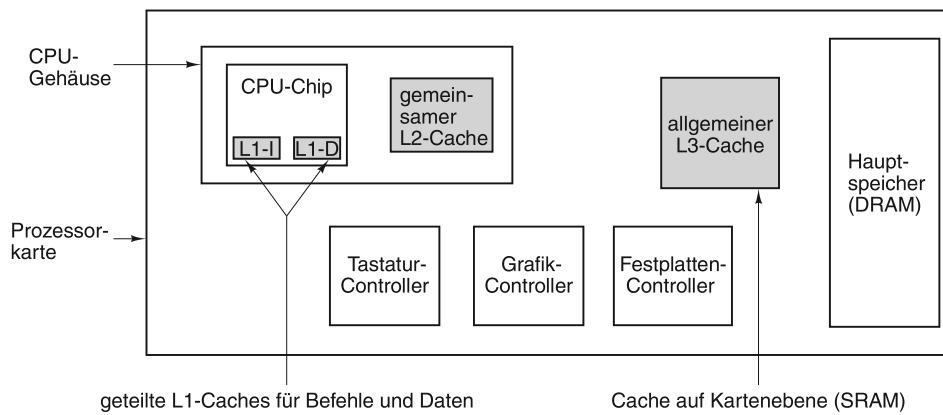


Abbildung 4.25: Ein System mit drei Cache-Ebenen

Caches beruhen auf zwei Arten der Adresslokalität, um ihr Ziel zu erreichen. Die **örtliche Lokalität** (Spatial Locality) beschreibt die Beobachtung, dass auf Speicherstellen mit Adressen, die einer kürzlich angesprochenen Speicherstelle numerisch ähnlich sind, in der nächsten Zeit wahrscheinlich zugegriffen wird. Caches nutzen diese Eigenschaft, indem sie mehr Daten als angefordert einlesen, in der Annahme, dass künftige Anforderungen vorausgesagt werden können. Die **zeitliche Lokalität** (Temporal Locality) liegt vor, wenn auf Speicherstellen, auf die kürzlich zugegriffen wurde, erneut zugegriffen wird. Dies können beispielsweise Speicherstellen nahe der Spitze des Kellers oder Befehle in einer Schleife sein. Die zeitliche Lokalität nutzt man bei der Auslegung eines Cache vor allem zur Entscheidung, was bei einem Cache-Zugriffsfehler zu verwerfen ist. Viele Cache-Ersetzungsalgorithmen verwerfen aufgrund der zeitlichen Lokalität die Einträge, auf die in der letzten Zeit nicht zugegriffen wurde.

Alle Caches basieren auf dem folgenden Prinzip: Man teilt den Hauptspeicher in Blöcke fester Größe namens **Cachezeilen** (Cache Lines) ein. Eine Cachezeile besteht normalerweise aus 4 bis 64 aufeinanderfolgenden Bytes. Die Zeilen sind ab 0 durchnummeriert. Bei einer Zeilengröße von 32 Byte umfasst Zeile 0 die Bytes 0 bis 31, Zeile 1 die Bytes 32 bis 63 usw. Im Cache befinden sich jederzeit mehrere Zeilen. Bei Zugriffen auf den Speicher prüft der Cache-Controller, ob sich das angeforderte Wort im Cache befindet. In diesem Fall kann der Prozessor das Wort verwenden, was einen Weg zum Hauptspeicher spart. Ist das Wort nicht im Cache vorhanden, dann wird ein bestimmter Zeilleneintrag aus dem Cache entfernt und die benötigte Zeile vom Hauptspeicher oder vom Cache einer darunterliegenden Ebene geholt. Für dieses Schema existieren viele Variationen, die aber alle auf dem Konzept basieren, dass die am häufigsten benutzten Zeilen so lange wie möglich im Cache verbleiben, um die Anzahl der Speicherzugriffe, die der Cache bedienen kann, zu maximieren.

Direkt abbildende Caches

Der einfachste Cache ist der **direkt abbildende Cache** (Direct-mapped Cache). ► Abbildung 4.26a zeigt ein Beispiel für einen direkt abbildenden Cache mit einer Ebene, der 2048 Einträge enthält. Jeder Eintrag (Zeile) im Cache kann genau eine Cachezeile aus dem Hauptspeicher aufnehmen. Bei einer Cachezeilengröße von 32 Byte (in diesem Beispiel) nimmt der Cache 64 KB auf. Jeder Cache-Eintrag besteht aus drei Teilen:

- 1 Das Bit **Valid** zeigt an, ob es in diesem Eintrag gültige Daten gibt. Wird das System hochgefahren (gestartet), dann werden alle Einträge als ungültig markiert.
- 2 Das Feld **Tag** besteht aus einem eindeutigen 16-Bit-Wert. Er kennzeichnet die entsprechende Zeile im Speicher, woher die Daten stammen.
- 3 Das Feld **Data** enthält eine Kopie der im Speicher befindlichen Daten. Dieses Feld nimmt eine Cachezeile von 32 Byte auf.

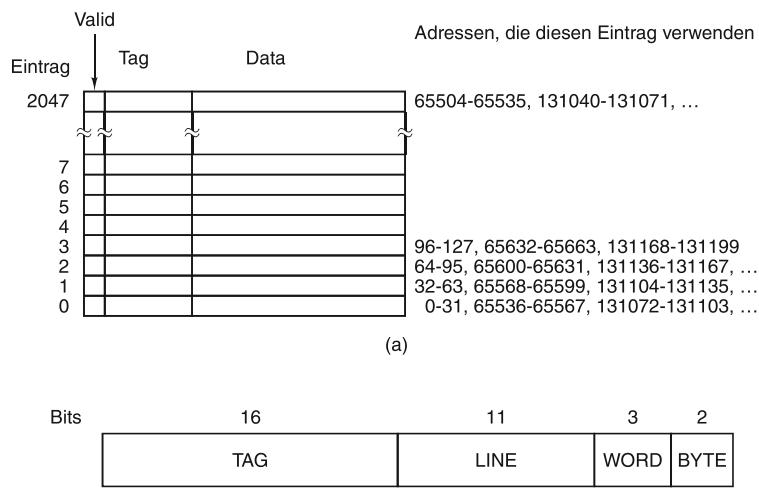


Abbildung 4.26: (a) Ein direkt abbildender Cache (b) eine virtuelle 32-Bit-Adresse

In einem direkt abbildenden Cache kann ein bestimmtes Speicherwort an genau einer Stelle im Cache gespeichert werden. Für eine bestimmte Speicheradresse gibt es nur eine Stelle, an der man im Cache nach dem Wort suchen muss. Steht es dort nicht, befindet es sich auch nicht im Cache. Zum Speichern und Abrufen von Cache-Daten wird die Adresse in vier Komponenten gemäß ► Abbildung 4.26b aufgeteilt:

- 1 Das Feld TAG entspricht den Tag-Bits, die in einem Cache-Eintrag gespeichert sind.
- 2 Das Feld LINE gibt an, in welchem Cache-Eintrag sich die entsprechenden Daten befinden, falls sie vorhanden sind.
- 3 Das Feld WORD gibt an, welches Wort innerhalb einer Zeile angefordert wird.
- 4 Das Feld BYTE wird normalerweise nicht benutzt. Wenn aber nur ein einzelnes Byte angefordert wird, gibt dieses Feld das Byte innerhalb des Wortes an. Bei einem Cache, der nur 32-Bit-Wörter liefert, ist dieses Feld immer 0.

Wenn die CPU eine Speicheradresse erzeugt, zieht die Hardware die 11 LINE-Bits aus der Adresse heraus und indiziert damit den Cache, um einen der 2048 Einträge zu finden. Bei gültigem Eintrag wird das TAG-Feld der Speicheradresse mit dem Tag-Feld im Cache-Eintrag verglichen. Stimmen beide überein, dann enthält der Cache-Eintrag das angeforderte Wort – es liegt ein **Cache-Treffer** (Cache Hit) vor. Bei einem Treffer kann der Prozessor das Wort aus dem Cache lesen, sodass kein Zugriff auf den Speicher notwendig ist. Aus dem Cache-Eintrag wird nur das angeforderte Wort extrahiert. Der Rest des Eintrags wird nicht benutzt. Ist der Cache-Eintrag ungültig oder stimmen die beiden Tags nicht überein, dann ist der benötigte Eintrag nicht im Cache vorhanden. Dies nennt man **Cache-Zugriffsfehler** (Cache Miss). In diesem Fall ruft der Cache-Controller die 32 Byte lange Cachezeile aus dem Speicher ab, speichert sie im Cache-Eintrag und ersetzt damit den ursprünglichen Inhalt dieses Eintrags. Wenn sich jedoch der vorhandene Cache-Eintrag seit dem Laden geändert hat, muss der Controller ihn zuerst in den Hauptspeicher zurückschreiben, bevor er den Eintrag verwerfen kann.

Auch wenn die zu treffenden Entscheidungen relativ kompliziert sind, kann der Zugriff auf ein benötigtes Wort bemerkenswert schnell sein. Sobald die Adresse bekannt ist,

steht auch die genaue Position des Wortes fest, *sofern es sich im Cache befindet*. Es ist also möglich, das Wort aus dem Cache zu lesen und dem Prozessor zu übergeben, und zwar bereits in dem Moment, in dem es als korrektes Wort (durch Vergleichen der Tags) erkannt wird. Der Prozessor erhält also ein Wort aus dem Cache simultan oder möglicherweise noch bevor er weiß, ob es sich um das angeforderte Wort handelt.

Dieses Abbildungsschema übernimmt aufeinanderfolgende Speicherzeilen in aufeinanderfolgende Cache-Einträge. Tatsächlich können bis zu 64 KB zusammenhängende Datenbytes im Cache gespeichert werden. Andererseits lassen sich zwei Zeilen, die sich in ihrer Adresse um genau 64 KB (65.536 Byte) oder ein ganzzahliges Vielfaches dieser Zahl unterscheiden, nicht gleichzeitig im Cache speichern (weil sie denselben LINE-Wert haben). Wenn ein Programm beispielsweise auf Daten an Position X zugreift und als Nächstes einen Befehl ausführt, der Daten von Position $X + 65.536$ (oder einer anderen Stelle innerhalb derselben Zeile) braucht, erzwingt der zweite Befehl, dass der Cache-Eintrag neu geladen und der vorhandene Inhalt überschrieben wird. Passiert das recht häufig, verschlechtert sich das Leistungsverhalten. Im ungünstigsten Fall ist es sogar mit Cache schlechter, als wäre überhaupt kein Cache vorhanden, da jede Speicheroperation eine komplette Cachezeile und nicht nur ein einzelnes Wort liest.

Direkt abbildende Caches sind die häufigsten Cache-Typen und sie arbeiten recht effektiv, weil Kollisionen, wie die oben beschriebene, selten oder überhaupt nie vorkommen. Beispielsweise kann ein cleverer Compiler von vornherein Kollisionen berücksichtigen, wenn er Befehle und Daten im Speicher platziert. Beachten Sie, dass der oben beschriebene Sonderfall bei einem System mit getrennten Befehls- und Datencaches nicht auftritt, weil verschiedene Caches die kollidierenden Anfragen bedienen. Hier zeigt sich ein zweiter Vorteil von zwei Caches gegenüber nur einem: mehr Flexibilität beim Umgang mit Speicherkonflikten.

Mengenassoziative Caches

Wie bereits erwähnt, konkurrieren viele unterschiedliche Zeilen im Speicher um dieselben Cache-Einträge. Wenn ein Programm, das den Cache gemäß Abbildung 4.26a verwendet, häufig Wörter an den Adressen 0 und 65536 holt, treten ständig Konflikte auf, wobei jede Referenz möglicherweise die andere aus dem Cache verdrängt. Dieses Problem lässt sich beispielsweise mit zwei oder mehr Zeilen in jedem Cache-Eintrag lösen. Ein Cache mit n möglichen Einträgen für jede Adresse heißt **n -fach mengenassoziativer Cache** (n -way Set-associative Cache). ▶ Abbildung 4.27 zeigt einen 4-fach mengenassoziativen Cache.

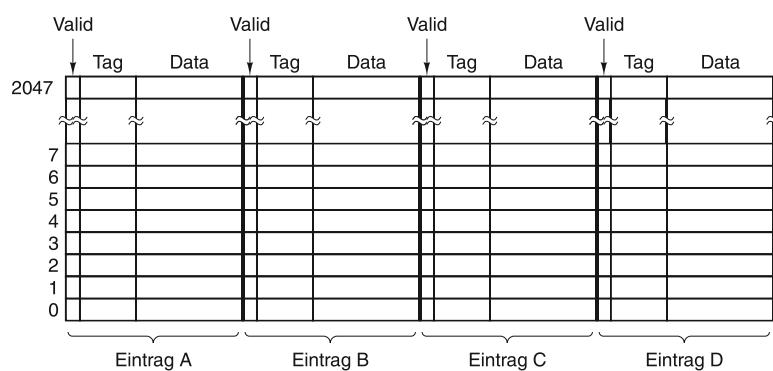


Abbildung 4.27: Ein 4-fach mengenassoziativer Cache

Mengenassoziativer Cache vermeiden Konflikte um einen einzelnen Cache-Eintrag, in dem mehrere Cache-Einträge für eine Speicheradresse zur Verfügung stehen. Diese müssen dann bei einem Cache-Zugriff nach der Adresse durchsucht werden.

Ein mengenassoziativer Cache ist von Haus aus komplizierter als ein direkt abbildender Cache. Zwar lässt sich der richtige Cache-Eintrag aus der referenzierten Speicheradresse berechnen, aber es muss eine Menge von n Cache-Einträgen geprüft werden, um festzustellen, ob die benötigte Zeile vorhanden ist. Und diese Prüfung muss sehr schnell erfolgen. Dennoch zeigen Simulationen und die Erfahrung, dass 2- und 4-fach Caches gute Leistungen bringen, für die sich der zusätzliche Schaltungsaufwand lohnt.

Die Verwendung eines mengenassoziativen Caches stellt den Designer vor eine Entscheidung: Welches der vorhandenen Elemente ist zu verwerfen, wenn ein neuer Eintrag in den Cache überführt wird? Die optimale Wahl setzt natürlich einen Blick in die Zukunft voraus, wobei aber für die meis-

ten Zwecke der Algorithmus **LRU** (Least Recently Used) recht brauchbar ist. Dieser Algorithmus verwaltet eine Reihenfolge jeder Gruppe von Speicherstellen. Erfolgt ein Zugriff auf vorhandene Zeilen, aktualisiert er die Liste und markiert den Eintrag, auf den zuletzt zugegriffen wurde. Ist ein Eintrag zu ersetzen, wird der am Ende der Liste befindliche Eintrag – auf den am längsten nicht zugegriffen wurde – verworfen.

Im Extremfall ist auch ein 2048-fach Cache möglich, der eine einzige Menge von 2048 Zeileneinträgen enthält. Hier werden alle Speicheradressen auf diese Menge abgebildet, sodass die Suche die Adresse mit allen 2048 Tags im Cache vergleichen muss. Beachten Sie, dass jeder Eintrag jetzt eine Logik für die Übereinstimmung der Tags besitzen muss. Da das Feld **LINE** eine Länge von 0 hat, stellt das **TAG**-Feld außer für **WORD**- und **BYTE**-Felder die gesamte Adresse dar. Darüber hinaus kommen alle 2048 Stellen für eine Verdrängung infrage, wenn eine Cachezeile zu ersetzen ist. Die Verwaltung einer geordneten Liste von 2048 Einträgen bedeutet einen erheblichen Verwaltungsaufwand, was die LRU-Ersetzung unbrauchbar macht. (Diese Liste muss bei jeder Speicheroperation und nicht nur bei jedem Zugriffsfehler aktualisiert werden.) Überraschenderweise ist die Leistung von stark assoziativen Caches in den meisten Fällen kaum höher als bei schwach assoziativen Caches und manchmal sogar schlechter. Aus diesen Gründen sind höchstens 4-fach mengenassoziative Caches üblich.

Schließlich stellen Schreiboperationen ein besonderes Problem für Caches dar. Schreibt ein Prozessor ein Wort und befindet sich das Wort im Cache, muss er natürlich entweder das Wort aktualisieren oder den Cache-Eintrag verwerfen. Nahezu alle Lösungen aktualisieren den Cache. Wie sieht es aber mit der Aktualisierung der Kopie im Hauptspeicher aus? Diese Operation lässt sich auf später verschieben, wenn die Cachezeile zur Verdrängung durch den LRU-Algorithmus ausgewählt ist. Diese Wahl ist schwierig und es gibt keine wirklich bevorzugte Option. Das sofortige Aktualisieren des Eintrags im Hauptspeicher nennt man **Write Through**. Dieser Ansatz lässt sich im Allgemeinen leichter implementieren und er ist zuverlässiger, weil der Speicher immer auf dem neuesten Stand bleibt – das ist beispielsweise hilfreich, wenn ein Fehler auftritt und der Speicherzustand wiederhergestellt werden muss. Leider bedeutet er andererseits mehr Schreibverkehr zum Speicher, sodass man bei anspruchsvollerem Implementierungen zu der als **Write Deferred** oder **Write Back** bezeichneten Alternative greift.

Für Schreiboperationen ist noch ein verwandtes Problem zu lösen: Was passiert, wenn an eine Stelle geschrieben wird, die sich momentan nicht im Cache befindet? Sollten die Daten in den Cache gebracht oder einfach nur in den Speicher übernommen werden? Wieder sind beide Möglichkeiten nicht immer optimal. Die meisten Konzepte mit verzögertem Schreiben in den Speicher (Write Deferred) bringen die Daten bei einem Schreibzugriffsfehler in den Cache. Diese Technik nennt man **Write Allocation** (Schreibzuordnung).

Demgegenüber ordnen die meisten Lösungen nach dem Write-Through-Verfahren bei einer Schreiboperation keinen Eintrag zu, weil diese Option ein sonst einfaches Konzept verkompliziert. Die Schreibzuordnung ist nur im Vorteil, wenn es wiederholte Schreiboperationen auf dasselbe Wort oder verschiedene Wörter innerhalb einer Cachezeile gibt.

Die Cache-Leistung ist für die Systemleistung entscheidend, weil die Lücke zwischen CPU-Geschwindigkeit und Speichergeschwindigkeit weit auseinanderklafft. Folglich ist die Untersuchung besserer Cache-Strategien weiterhin ein brandaktuelles Thema [Sanchez und Kozyrakis, 2011] und [Gaur et al., 2011].

4.5.2 Sprungvorhersage

Moderne Computer arbeiten stark fließbandorientiert. Zum Beispiel besteht die Pipeline in Abbildung 4.23 aus sieben Stufen. Computer im oberen Leistungsbereich können 10-stufige oder sogar noch längere Pipelines aufweisen. Fließbandverarbeitung funktioniert am besten mit linearem Code, sodass die Abrufeinheit einfach aufeinanderfolgende Wörter aus dem Speicher lesen und sie vorab an die Decodiereinheit weiterreichen kann.

Dieses wunderbare Modell hat nur den kleinen Haken, dass es nicht im Mindesten realistisch ist. Programme bestehen nicht aus linearen Codesequenzen, sondern sind voller Verzweigungsbefehle. Sehen Sie sich die einfachen Anweisungen in ►Listing 4.2a an. Die Variable *i* wird mit 0 verglichen (der in der Praxis sicherlich häufigste Test). Je nach Ergebnis wird einer weiteren Variablen *k* einer von zwei möglichen Werten zugewiesen.

<pre>if (i == 0) k = 1; else k = 2;</pre>	<pre>CMP i,0 ; vergleiche i mit 0 BNE Else ; verzweige zu Else, falls ungleich Then: MOV k,1 ; übertrage 1 nach k BR Next ; unbedingter Sprung zu Next Else: MOV k,2 ; übertrage 2 nach k Next:</pre>
(a)	(b)

Listing 4.2: (a) Ein Programmfragment und (b) seine Übersetzung in eine generische Assemblersprache

►Listing 4.2b zeigt eine mögliche Übersetzung in Assemblersprache. Auf Assemblersprachen kommen wir später in diesem Buch noch zurück und die Details sind hier auch nicht wichtig. Jedenfalls ist ein Code wahrscheinlich, der je nach Computer und Compiler mehr oder weniger dem in Listing 4.2b ähnelt. Der erste Befehl vergleicht *i* mit 0. Die zweite verzweigt zur Marke *Else* (dem Beginn des *else*-Zweigs), falls *i* nicht 0 ist. Der dritte Befehl weist 1 an *k* zu. Der vierte Befehl verzweigt zum Code für den nächsten Befehl. Der Compiler hat praktischerweise die Marke *Next* an die entsprechende Stelle gesetzt, sodass diese Stelle als Ziel einer Verzweigung infrage kommt. Der fünfte Befehl weist 2 an *k* zu.

Hier fällt auf, dass zwei der fünf Befehle Verzweigungen (Sprünge) sind. Außerdem ist einer davon, *BNE*, eine bedingte Verzweigung (ein Sprung, der ausgeführt wird, sofern eine Bedingung erfüllt ist – in diesem Fall, dass die beiden Operanden im vorherigen *CMP*-Befehl ungleich sind). Die längste lineare Codesequenz besteht hier aus zwei Befehlen. Folglich ist es schwierig, Befehle mit einer hohen Rate in die Pipeline abzurufen.

Auf den ersten Blick sieht es so aus, als ob die unbedingten Sprünge, z.B. der Befehl *BR Next* in Listing 4.2b, unproblematisch sind. Immerhin liegt der Weg eindeutig fest.

Warum kann die Abrufeinheit nicht einfach damit fortfahren, Befehle von der Zieladresse an zu lesen (d.h. von der Stelle, zu der die Verzweigung führt)?

Bei Programmverzweigungen ist der nächste auszuführende Befehl abhängig von vorherigen Berechnungen. Die Fließbandverarbeitung wird somit behindert. Um dies zu umgehen, werden komplexe Techniken zur **Sprung-vorhersage** in der Hardware realisiert, die spekulativ den nächsten Befehl bestimmen.

Die Probleme liegen in der Natur der Fließbandverarbeitung. Zum Beispiel zeigt Abbildung 4.23, dass die Befehlsdecodierung in der zweiten Stufe erfolgt. Somit muss die Abrufeinheit entscheiden, von wo als Nächstes zu lesen ist, bevor sie überhaupt weiß, welche Art von Befehl sie gerade erhalten hat. Erst einen Zyklus später erfährt sie, dass sie gerade einen unbedingten Sprung bearbeitet. Inzwischen hat sie aber begonnen, den auf den unbedingten Sprung folgenden Befehl zu lesen. Aus diesem Grund haben die meisten Pipeline-Maschinen (z.B. die UltraSPARC III) die Eigenschaft, den Befehl *nach* einem unbedingten Sprung auszuführen, obwohl das logisch nicht korrekt ist. Die Position nach einer Verzweigung nennt man **Delay-Slot** (etwa „Verzögerungsschlitz“). Beim Core i7 – und der in Listing 4.2b verwendeten Maschine – ist diese Eigenschaft nicht implementiert. Allerdings betreibt man oftmals einen erheblichen internen Aufwand, um dieses Problem zu umgehen. Ein optimierender Compiler versucht im Allgemeinen, einen nützlichen Befehl für den Delay-Slot zu finden. Häufig gelingt das nicht und der Compiler ist gezwungen, einen NOP-Befehl einzufügen. Programme bleiben dadurch korrekt, werden aber größer und langsamer.

So „störend“ unbedingte Verzweigungen sind, bei bedingten Verzweigungen sieht die Lage noch schlechter aus. Abgesehen davon, dass auch hier Delay-Slots vorhanden sind, erfährt die Abrufeinheit erst viel später in der Pipeline, von wo zu lesen ist. Frühere Pipeline-Maschinen haben die Pipeline einfach angehalten, bis bekannt wurde, ob die Verzweigung erfolgen soll oder nicht. Ein **Stillstand** über drei oder vier Zyklen hinweg bei jedem bedingten Sprung hat eine verheerende Wirkung auf die Leistung, insbesondere wenn 20% der Befehle bedingte Sprünge sind.

Deshalb verfährt man bei den meisten modernen Prozessoren wie folgt: Wenn sie auf einen bedingten Sprung stoßen, versuchen sie vorauszusagen, ob dieser auszuführen ist oder nicht. Sicherlich wäre es schön, wenn man einfach eine Glaskugel in einem freien PCIe-Steckplatz (oder besser noch, in der IFU) unterbringen könnte, die bei der Vorhersage behilflich ist. Doch diese Idee hat noch keine Früchte getragen.

Da ein solch schönes Peripheriegerät leider nicht existiert, hat man sich verschiedene Vorhersagestrategien ausgedacht. Eine sehr einfache Methode nimmt an, dass alle Rückwärtsverzweigungen ausgeführt und alle Vorwärtsverzweigungen nicht ausgeführt werden. Die Argumentation für den ersten Teil stützt sich auf die Beobachtung, dass nach hinten führende Sprünge häufig am Ende einer Schleife stehen. Da die meisten Schleifen mehrfach ausgeführt werden, liegt man mit der Vermutung, dass eine Verzweigung an den Anfang einer Schleife erfolgt, oftmals richtig.

Der zweite Teil der Annahme ist weniger gut fundiert. Bestimmte Vorwärtsverzweigungen treten auf, wenn die Software Fehlerbedingungen erkennt (z.B. wenn sich eine Datei nicht öffnen lässt). Da aber Fehler selten sind, finden die meisten derartigen Verzweigungen nicht statt. Natürlich gibt es jede Menge Vorwärtsverzweigungen, die nichts mit der Fehlerbehandlung zu tun haben, sodass hier die Erfolgsrate nicht annähernd so gut ist wie bei Sprüngen nach hinten. Somit ist diese Regel zwar nicht berauschend, aber besser als gar keine.

Trifft die Voraussage für einen Sprung ein, dann ist nichts Besonderes zu unternehmen. Die Programmausführung fährt einfach an der Zieladresse fort. Die Probleme zeigen sich erst, wenn ein Sprung falsch vorhergesagt wird. Das wirkliche Ziel zu ermitteln und dorthin zu

gelangen, ist noch einfach. Der schwierige Teil besteht darin, die Befehle rückgängig zu machen, die bereits ausgeführt wurden, aber nun nicht mehr ausgeführt werden sollen.

Aus dieser Situation führen zwei Wege heraus. Beim ersten lässt man die Ausführung der Befehle zu, die bereits nach einer vorhergesagten bedingten Verzweigung abgerufen wurden, bis sie versuchen, den Zustand der Maschine zu ändern (z.B. in ein Register zu speichern). Anstatt das Register zu überschreiben, kommt der berechnete Wert in ein (geheimes) Notizregister und wird nur dann in das echte Register kopiert, wenn bekannt ist, dass die Vorhersage richtig war. Beim zweiten Weg zeichnet man den Wert eines zu überschreibenden Registers (z.B. in einem geheimen Notizregister) auf, sodass sich die Maschine gegebenenfalls in den Zustand zurückbringen lässt, den sie zum Zeitpunkt der falsch vorhergesagten Verzweigung hatte. Beide Lösungen sind komplex und erfordern wasserdiichte Verwaltungsmechanismen, damit alles glatt geht. Und wenn ein zweiter bedingter Sprung auftaucht, bevor bekannt ist, ob der erste richtig vorhergesagt wurde, ist das Durcheinander im wahrsten Sinne des Wortes vorprogrammiert.

Dynamische Sprungvorhersage

Zweifellos sind richtige Vorhersagen wertvoll, da die CPU dann mit voller Geschwindigkeit fortfahren kann. Deshalb ist das Thema der Algorithmen zur Sprungvorhersage ein wichtiges Forschungsgebiet [Chen et al., 2003], [Falcon et al., 2004], [Jimenez, 2003] und [Parikh et al., 2004]. Bei einer Methode verwaltet die CPU (mit spezieller Hardware) eine History-Tabelle (d.h. eine Tabelle für die Vorgeschichte von Verzweigungen), in der sie die aufgetretenen bedingten Verzweigungen protokolliert, sodass die CPU darauf zurückgreifen kann, wenn die Verzweigungen erneut vorkommen.

► Abbildung 4.28 zeigt die einfachste Version dieses Schemas. Hier nimmt die History-Tabelle einen Eintrag für jeden bedingten Verzweigungsbefehl auf. Der Eintrag enthält die Adresse des Verzweigungsbefehls zusammen mit einem Bit, das angibt, ob der Sprung beim letzten Mal ausgeführt wurde. Nach diesem Schema besteht die Vorhersage einfach darin, dass die Verzweigung denselben Weg wie beim letzten Mal nimmt. Ist die Vorhersage falsch, wird das Bit in der History-Tabelle geändert.

Dynamische Vorhersagestrategien
berücksichtigen im Gegensatz zu statischen die im bisherigen Programmverlauf getroffenen Entscheidungen.

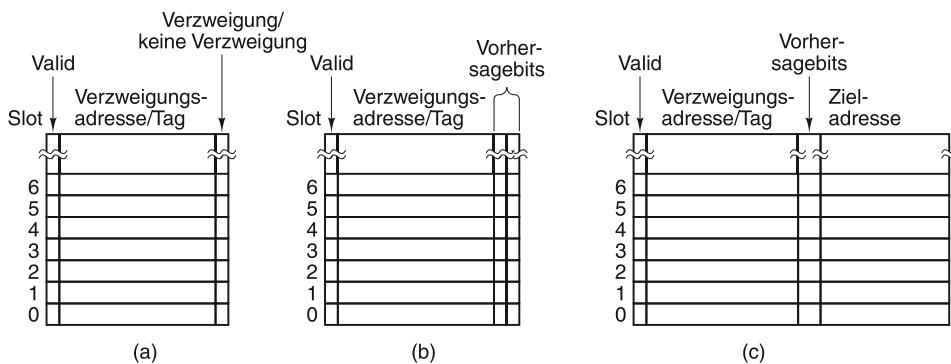


Abbildung 4.28: (a) Ein Verzweigungsverlauf mit 1 Bit (b) ein Verzweigungsverlauf mit 2 Bits (c) eine Abbildung zwischen Sprungbefehls- und Zieladresse

Eine History-Tabelle lässt sich auf verschiedene Weise organisieren und zwar prinzipiell genauso wie bei einem Cache. Nehmen wir einen Prozessor an mit 32-Bit-Befehlen, die an Wörtern ausgerichtet sind, sodass die niederwertigen zwei Bits jeder Speicheradresse 00 sind. Bei einer direkt abgebildeten History-Tabelle mit 2^n Einträgen können die niederwer-

tigen $n + 2$ Bits eines Sprungbefehls herausgezogen und um 2 Bitstellen nach rechts geschoben werden. Diese n -Bit-Zahl kann als Index auf die History-Tabelle dienen. Dort wird geprüft, ob die gespeicherte Adresse mit der Adresse der Verzweigung übereinstimmt. Wie bei einem Cache ist es nicht notwendig, die niederwertigen $n + 2$ Bits zu speichern (d.h. nur die oberen Adressbits – das Tag – werden gespeichert). Im Fall eines Treffers sagt die Logik den Sprung anhand des Vorhersagebits voraus. Ist das falsche Tag vorhanden oder ist der Eintrag ungültig, so handelt es sich – genau wie bei einem Cache – um einen Fehlschlag. In diesem Fall lässt sich die Regel für Vorwärts-/Rückwärtssprünge anwenden.

Sind in der History-Tabelle beispielsweise 4096 Einträge verzeichnet, dann kollidieren die Sprünge an den Adressen 0, 16384, 32768, usw. wie bei einem Cache. Die gleiche Lösung ist auch hier möglich: ein 2-facher, 4-facher oder n -facher assoziativer Eintrag. Wie bei einem Cache ist die obere Grenze ein einziger n -facher assoziativer Eintrag, der eine volle Suche voraussetzt.

Bei einer genügend großen Tabelle und ausreichender Assoziativität funktioniert diese Methode in den meisten Fällen. Ein Problem tritt aber immer auf: Beim Verlassen einer Schleife wird der Sprung am Ende falsch vorhergesagt. Noch schlimmer ist, dass diese falsche Vorhersage das Bit in der History-Tabelle ändert, um künftig „kein Sprung“ vorherzusagen. Beim nächsten Eintritt in die Schleife wird der Sprung am Ende des ersten Durchlaufs falsch vorhergesagt. Liegt die Schleife in einer äußeren Schleife oder in einer häufig aufgerufenen Prozedur, tritt dieser Fehler entsprechend oft auf.

Um diese Fehlvorhersage zu vermeiden, können wir dem Tabelleneintrag eine zweite Chance geben. Bei dieser Methode wird die Vorhersage erst nach zwei aufeinanderfolgenden falschen Vorhersagen geändert. Dieser Ansatz setzt voraus, dass sich in der History-Tabelle zwei Vorhersagebits befinden – eines für das, was der Sprung tun „soll“ und eines für das, was er beim letzten Mal getan hat, wie es ►Abbildung 4.28b zeigt.

Man kann diesen Algorithmus auch als endlichen Automaten mit vier Zuständen betrachten (►Abbildung 4.29). Nach einer Reihe von aufeinanderfolgenden erfolgreichen „kein Sprung“-Vorhersagen ist der endliche Automat im Zustand 00 und sagt beim nächsten Mal „kein Sprung“ voraus. Ist diese Vorhersage falsch, geht er in den Zustand 10 über, sagt aber beim nächsten Mal wieder „kein Sprung“ voraus. Erst wenn auch diese Vorhersage falsch ist, geht er in den Zustand 11 über und sagt dann jedes Mal Sprunge voraus. Praktisch gibt das linke Zustandsbit die Vorhersage an und das rechte Bit sagt aus, wie der Sprung beim letzten Mal erfolgt ist. Dieses Konzept kommt mit nur 2 Verlaufsbits aus, es lässt sich aber auch auf 4 oder 8 Verlaufsbits erweitern.

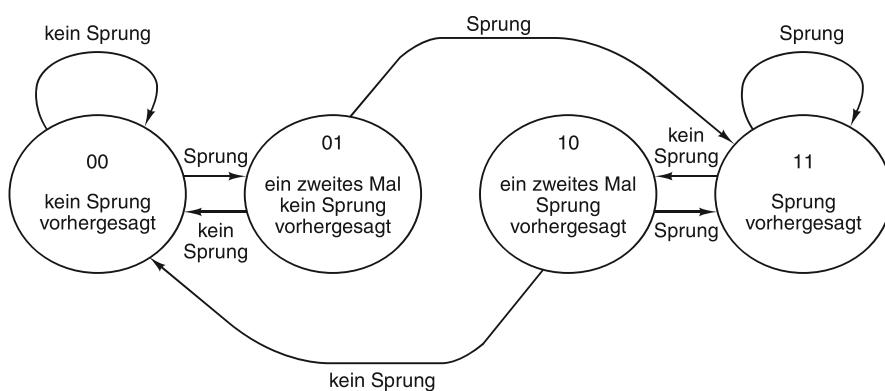


Abbildung 4.29: Ein endlicher Automat mit 2 Bits für die Sprungvorhersage

Einen endlichen Automaten haben Sie bereits in Abbildung 4.19 kennengelernt. Im Grunde lassen sich alle unsere Mikroprogramme als endlicher Automat betrachten, da jede Zeile einen bestimmten Zustand darstellt, in dem sich die Maschine befinden kann, wobei eine endliche Menge von Übergängen zu weiteren Zuständen genau definiert ist. Endliche Automaten sind ein weit verbreitetes Instrument für alle Aspekte des Hardwareentwurfs.

Bisher haben wir angenommen, dass das Ziel jedes bedingten Sprungs bekannt ist, entweder als explizite (im Befehl selbst enthaltene) Zieladresse oder als relativer Offset vom aktuellen Befehl aus (d.h. als Zahl mit Vorzeichen, die zum Programmzähler addiert wird). Diese Annahme trifft oftmals zu, doch berechnen einige bedingte Sprungbefehle die Zieladresse über Registerarithmetik und verzweigen dann zur resultierenden Adresse. Auch wenn der endliche Automat von Abbildung 4.29 genau vorhersagt, ob der Sprung erfolgt, ist eine solche Vorhersage bei unbekannter Zieladresse wertlos. Diese Situation lässt sich verbessern, wenn man die tatsächliche Adresse, zu der beim letzten Mal verzweigt wurde, in der History-Tabelle speichert, wie es Abbildung 4.28c zeigt. Besagt nun die Tabelle, dass die Verzweigung an Adresse 516 beim letzten Mal zu Adresse 4000 geführt hat, und lautet die Vorhersage jetzt „Sprung“, so führt der Sprung wieder zu 4000.

Die **Sprungvorhersage** kann auch die Zieladresse umfassen, sodass auf die Adressberechnung nicht gewartet werden muss.

Ein anderes Verfahren zur Sprungvorhersage protokolliert, ob die letzten k bedingten Sprünge ausgeführt wurden, wobei es die Art des Befehls nicht berücksichtigt. Diese k -Bit-Zahl wird im sogenannten **Branch History Shift Register** (Schieberegister für die Vorgeschichte von Verzweigungen) geführt und dann parallel zu allen Einträgen einer History-Tabelle mit einem k -Bit-Schlüssel verglichen. Bei einem Treffer wird die hier verzeichnete Vorhersage verwendet. Es mag überraschen, dass sich diese Technik in der Praxis als recht brauchbar erwiesen hat.

Statische Sprungvorhersage

Alle bisher behandelten Techniken zur Sprungvorhersage arbeiten dynamisch, d.h. zur Laufzeit des Programms. Zudem können sie sich an das momentane Verhalten des Programms anpassen. Nachteilig ist, dass sie spezielle und teure Hardware sowie ein ziemlich komplexes Chipdesign verlangen.

Ein anderer Weg führt über die Hilfe des Compilers. Trifft der Compiler auf eine Anweisung wie

```
for (i = 0; i < 1000000; i++) {...}
```

dann weiß er natürlich, dass der Sprung am Ende der Schleife fast jedes Mal erfolgt. Wenn er nur eine Möglichkeit hätte, dies der Hardware begreiflich zu machen, ließe sich eine Menge Aufwand sparen.

Obwohl es sich hierbei um eine Änderung der Architektur (und nicht nur um eine Implementierungsfrage) handelt, verfügen einige Prozessoren – z.B. der UltraSPARC III – über einen zweiten Satz von Sprungbefehlen (zusätzlich zu den normalen, die wegen der Abwärtskompatibilität erforderlich sind). In einem speziellen Bit der neuen Sprungbefehle kann der Compiler festlegen, dass der Sprung seiner Meinung nach erfolgen (bzw. nicht erfolgen) soll. Wenn im Programmablauf derartige Befehle erscheinen, richtet sich die Abrufeinheit einfach nach diesem Verzweigungsbit. Außerdem muss man für diese Befehle keinen kostbaren Platz in der History-Tabelle vergeuden, was die Wahrscheinlichkeit für Konflikte verringert.

Unsere letzte Sprungvorhersagetechnik basiert auf Profiling [Fisher und Freudenberger, 1992]. Dabei handelt es sich ebenfalls um ein statisches Verfahren. Anstatt aber den Compiler herausfinden zu lassen, welche Sprünge auszuführen sind und welche nicht, wird das Programm tatsächlich (normalerweise auf einem Simulator) ausgeführt und das Sprungverhalten erfasst. Der Compiler erhält diese Informationen und generiert dann die speziellen bedingten Sprungbefehle, um der Hardware die Sprungrichtung mitzuteilen.

4.5.3 Out-of-Order-Ausführung und Registerumbenennung

Die meisten modernen CPUs sind sowohl fließbandorientiert als auch superskalar, wie es aus Abbildung 2.5 hervorgeht. Das bedeutet im Allgemeinen, dass eine Abrufeinheit aus dem Speicher Befehlswörter holt, bevor sie benötigt werden, um sie an eine Decodiereinheit zu übergeben. Die Decodiereinheit leitet die decodierten Befehle an die entsprechenden Funktionseinheiten zur Ausführung weiter. In manchen Fällen kann sie einzelne Befehle in Mikrooperationen auflösen, bevor sie sie an die Funktionseinheiten ausgibt, je nachdem, wofür die Funktionseinheiten ausgelegt sind.

Natürlich sieht das Design eines Prozessors am einfachsten aus, wenn er alle Befehle in der Reihenfolge ausführt, in der er sie abruft (wobei wir fürs Erste annehmen, dass der Algorithmus zur Sprungvorhersage immer richtig entscheidet). Allerdings ergibt sich bei dieser In-Order-Ausführung infolge von Abhängigkeiten zwischen den Befehlen nicht immer die optimale Leistung. Benötigt ein Befehl einen Wert, den ein vorheriger Befehl berechnet, dann lässt sich dieser zweite Befehl erst ausführen, wenn der erste den erforderlichen Wert bereitgestellt hat. In dieser Situation (einer RAW-Abhängigkeit) muss der zweite Befehl warten. Wie Sie gleich sehen werden, gibt es daneben noch weitere Abhängigkeiten.

Um diese Probleme zu umgehen und eine bessere Leistung zu erreichen, können manche CPUs abhängige Befehle überspringen und zu nachfolgenden Befehlen übergehen, die nicht abhängig sind. Eigentlich ist es überflüssig zu betonen, dass der interne Algorithmus zur Befehlsplanung dieselbe Wirkung gewährleisten muss, als ob das Programm in der geschriebenen Reihenfolge ausgeführt würde. Wir zeigen jetzt anhand eines ausführlichen Beispiels, wie das Umstellen von Befehlen funktioniert.

Um das Wesen des Problems zu veranschaulichen, beginnen wir mit einer Maschine, die Befehle immer in der Programmreihenfolge startet und verlangt, dass sie in der Programmreihenfolge fertiggestellt werden. Die Bedeutung der zweiten Bedingung wird später klar.

Unsere Beispielmaschine hat acht Register, die für den Programmierer sichtbar sind – R0 bis R7. Alle arithmetischen Befehle verwenden drei Register: zwei für die Operanden und eines für das Ergebnis, genau wie bei der Mic-4. Wir nehmen an, dass die Ausführung in Zyklus $n + 1$ beginnt, wenn der Befehl in Zyklus n decodiert wird. Bei einem einfachen Befehl, wie zum Beispiel einer Addition oder Subtraktion, erfolgt das Zurückschreiben in das Zielregister am Ende von Zyklus $n + 2$. Bei einem komplizierteren Befehl, wie zum Beispiel einer Multiplikation, findet das Zurückschreiben am Ende von Zyklus $n + 3$ statt. Um das Beispiel realistisch zu machen, erlauben wir der Decodiereinheit, bis zu zwei Befehle pro Taktzyklus auszulösen. Kommerzielle superskalare CPUs können oftmals vier oder sogar sechs Befehle pro Taktzyklus ausgeben.

►Tabelle 4.12 zeigt die Ausführungssequenz für das Beispiel. Hier gibt die erste Spalte (Z) die Zyklusnummer und die zweite Spalte (B) die Befehlsnummer an. In der dritten Spalte steht der decodierte Befehl. Die vierte Spalte (A) nennt den auszugebenden Befehl (wobei maximal zwei Befehle je Taktzyklus möglich sind). Die fünfte Spalte (R)

gibt Auskunft über die abgeschlossenen Befehle. Es sei daran erinnert, dass dieses Beispiel sowohl das Starten als auch das Abschließen der Befehle in der Programmreihenfolge (In-Order) verlangt. Deshalb lässt sich Befehl $k + 1$ erst nach Befehl k starten und Befehl $k + 1$ erst abschließen (d.h. das Ergebnis in das Zielregister schreiben), nachdem Befehl k abgeschlossen wurde. Auf die übrigen 16 Spalten gehen wir später ein.

Z	B	Decodiert	A	R	Gelesene Register							Geschriebene Register									
					0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	
1	1	R3=R0 * R1	1		1	1								1							
	2	R4=R0 + R2	2		2	1	1							1	1						
2	3	R5=R0 + R1	3		3	2	1							1	1	1					
	4	R6=R1 + R4	—		3	2	1							1	1	1					
3					3	2	1							1	1	1					
4					1	2	1	1						1	1						
					2	1	1							1							
					3																
5					4		1		1						1						
	5	R7=R1 * R2	5		2	1		1						1	1						
6	6	R1=R0 - R2	—		2	1		1						1	1						
7					4		1	1							1						
8					5																
9					6		1		1					1							
	7	R3=R3 * R1	—		1		1							1							
10					1		1							1							
11					6																
12					7		1		1					1							
	8	R1=R4 + R4	—		1		1							1							
13					1		1							1							
14					1		1							1							
15					7																
16					8				2					1							
17									2					1							
18					8																

Tabelle 4.12: Abläufe in einer superskalaren CPU mit In-Order-Starten und In-Order-Abschließen

Nach dem Decodieren eines Befehls muss die Decodiereinheit entscheiden, ob er sich sofort starten lässt oder nicht. Dafür muss sie den Status aller Register kennen. Braucht beispielsweise der aktuelle Befehl ein Register, dessen Wert noch nicht berechnet wurde, kann der Befehl nicht ausgegeben werden und die CPU muss anhalten.

Die Registerverwendung protokollieren wir mit einem sogenannten **Scoreboard** (wörtlich „Anzeigetafel bei Spielen“), das erstmals im Supercomputer CDC 6600 zu finden ist. Das Scoreboard besitzt für jedes Register einen kleinen Zähler, der angibt, wie oft das betreffende Register als Quelle in den aktuell ausgeführten Befehlen erscheint. Wenn maximal 15 Befehle gleichzeitig ausgeführt werden, genügt ein 4-Bit-Zähler. Beim Starten eines Befehls werden die Scoreboard-Einträge für seine Operandenregister inkrementiert, beim Abschließen eines Befehls dekrementiert.

Außerdem besitzt das Scoreboard Zähler, um die als Ziele verwendeten Register zu protokollieren. Da jeweils nur eine Schreiboperation zulässig ist, können diese Zähler 1 Bit breit sein. Die 16 rechten Spalten in ►Tabelle 4.12 zeigen das Scoreboard.

Bei realen Computern protokolliert das Scoreboard auch die Verwendung von Funktionseinheiten, um zu vermeiden, dass ein Befehl an eine nicht verfügbare Funktionseinheit ausgegeben wird. Der Einfachheit halber gehen wir hier davon aus, dass immer eine Funktionseinheit verfügbar ist, und zeigen deshalb die Funktionseinheiten nicht auf dem Scoreboard an.

Die erste Zeile von Tabelle 4.12 zeigt Befehl 1 (kurz B1), der R0 mit R1 multipliziert und das Ergebnis in R3 ablegt. Da noch keines dieser Register in Gebrauch ist, wird der Befehl ausgegeben und das Scoreboard aktualisiert. Es zeigt jetzt, dass R0 und R1 gelesen und R3 geschrieben wird. Bis zum Ende von B1 kann kein nachfolgender Befehl in eines dieser Register schreiben oder R3 lesen. Dieser Befehl führt eine Multiplikation aus und wird damit am Ende von Zyklus 4 fertiggestellt. Die in jeder Zeile angegebenen Scoreboard-Werte entsprechen dem Zustand nach der Befehlausgabe der betreffenden Zeile. Für leere Einträge gilt 0.

Superskalare Prozessoren können mehrere Befehle pro Taktzyklus starten. Abhängigkeiten müssen hierbei eingehalten werden.

Da die superskalare Maschine unseres Beispiels zwei Befehle pro Zyklus starten kann, löst sie in Zyklus 1 einen zweiten Befehl (B2) aus. Er addiert R0 zu R2 und speichert das Ergebnis in R4. Anhand folgender Regeln lässt sich feststellen, ob dieser Befehl ausgegeben werden kann:

- 1 Nicht ausgeben, falls ein Operand geschrieben wird (RAW-Abhängigkeit).
- 2 Nicht ausgeben, falls das Ergebnisregister gelesen wird (WAR-Abhängigkeit).
- 3 Nicht ausgeben, falls in das Ergebnisregister geschrieben wird (WAW-Abhängigkeit).

Bereits kennengelernt haben Sie RAW-Abhängigkeiten, die auftreten, wenn ein Befehl als Quelle ein Ergebnis benötigt, das ein vorheriger Befehl noch nicht geliefert hat. Die beiden anderen Abhängigkeiten sind nicht so gravierend und betreffen im Wesentlichen Ressourcenkonflikte. Bei einer **WAR-Abhängigkeit** (Write After Read) versucht ein Befehl, ein Register zu überschreiben, das ein vorheriger Befehl möglicherweise noch liest. Bei der **WAW-Abhängigkeit** (Write After Write) verhält es sich ähnlich. Diese beiden Abhängigkeiten lassen sich oftmals vermeiden, wenn man die Ergebnisse des zweiten Befehls an einer anderen Stelle (eventuell temporär) speichern lässt. Liegt keine der genannten drei Abhängigkeiten vor und ist die benötigte Funktionseinheit verfügbar, dann wird der Befehl ausgegeben. In diesem Fall verwendet B2 ein Register (R0), das von

einem anhängigen Befehl gelesen wird. Diese Überlappung ist aber zulässig und folglich wird B2 ausgegeben. Das Gleiche gilt für B3 in Zyklus 2.

Nun kommen wir zum Befehl B4, der R4 verwenden muss. Wie aus Zeile 3 hervorgeht, wird R4 leider gerade geschrieben. Hier liegt eine RAW-Abhängigkeit vor, sodass die Decodiereinheit anhält, bis R4 wieder verfügbar ist. Während des Stillstands bezieht sie auch keine Befehle von der Abrufeinheit. Ist der interne Puffer der Abrufeinheit gefüllt, hört sie mit dem Prefetching auf.

Es sei betont, dass beim nächsten Befehl in der Programmreihenfolge, B5, keine Konflikte mit einem der anstehenden Befehle auftreten. Man könnte ihn decodieren und starten, wenn da nicht die Design-Forderung wäre, die Befehle in der Programmreihenfolge zu starten.

Sehen wir uns nun an, was während Zyklus 3 passiert. B2 wird als Additionsbefehl (zwei Zyklen) am Ende von Zyklus 3 fertig sein. Leider lässt er sich nicht abschließen (und somit R4 für B4 freimachen). Dieses Design verlangt nämlich auch das In-Order-Abschließen. Warum? Was könnte wohl passieren, wenn man jetzt in R4 speichert und es als verfügbar markiert?

Die Antwort ist subtil, aber wichtig. Nehmen wir nur einmal an, Befehle könnten Out-of-Order abgeschlossen werden. Sollte es dann zu einem Interrupt kommen, wäre es sehr schwierig, den Zustand der Maschine für die spätere Wiederherstellung zu speichern. Insbesondere könnte man überhaupt nicht mehr sagen, dass alle Befehle bis zu einer gewissen Adresse ausgeführt seien und alle darüber hinausgehenden Befehle nicht. Diese erwünschte Eigenschaft einer CPU bezeichnet man als **präzisen Interrupt** (Precise Interrupt), siehe auch in [Moudgil und Vassiliadis, 1996]. Durch das Out-of-Order-Abschließen von Befehlen werden Interrupts ungenau, weshalb einige Maschinen das In-Order-Abschließen von Befehlen verlangen.

Doch zurück zu unserem Beispiel: Am Ende von Zyklus 4 können alle drei anhängigen Befehle abgeschlossen werden, sodass schließlich in Zyklus 5 der Befehl B4 zusammen mit dem gerade decodierten Befehl B5 ausgelöst werden kann. Bei jedem Abschließen eines Befehls muss die Decodiereinheit prüfen, ob ein angehaltener Befehl vorhanden ist, der jetzt ausgegeben werden kann.

In Zyklus 6 kommt B6 zum Stillstand, weil er in R1 schreiben muss, R1 aber belegt ist. Schließlich wird der Befehl in Zyklus 9 gestartet. Die gesamte Sequenz von acht Befehlen beansprucht aufgrund vieler Abhängigkeiten 18 Zyklen zur Fertigstellung, obwohl die Hardware in jedem Zyklus zwei Befehle starten kann. Beachten Sie aber, dass die Spalte A in Abbildung 4.12 von oben nach unten gelesen zeigt, dass alle Befehle in der Programmreihenfolge gestartet wurden. Analog geht aus Spalte R hervor, dass alle Befehle auch in der Programmreihenfolge abgeschlossen wurden.

Nun betrachten wir ein alternatives Design: die **Out-of-Order-Ausführung**. Bei diesem Design können Befehle von der Programmreihenfolge abweichend gestartet und ebenso abgeschlossen werden. ►Tabelle 4.13 zeigt dieselbe Sequenz von acht Befehlen, wobei dieses Mal aber das Out-of-Order-Starten und das Out-Of-Order-Abschließen erlaubt sind.

Bei der **Out-of-Order-Ausführung** können Befehle abweichend von der Programmreihenfolge bearbeitet werden, wenn alle Abhängigkeiten von früheren Befehlen erfüllt sind. Dies kann zu einer besseren Auslastung der Funktionseinheiten führen.

Z	B	Decodiert	A	R	Gelesene Register							Geschriebene Register								
					0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7
1	1	R3=R0 * R1	1		1	1								1						
	2	R4=R0 + R2	2		2	1	1								1	1				
2	3	R5=R0 + R1	3		3	2	1								1	1	1			
	4	R6=R1 + R4	—		3	2	1								1	1	1			
3	5	R7=R1 * R2	5		3	3	2								1	1	1		1	
	6	S1=R0 - R2	6		4	3	3								1	1	1		1	
					2	3	3	2							1	1	1		1	
4			4		3	4	2	1							1	1	1	1		
	7	R3=R3 * S1	—		3	4	2	1							1	1	1	1		
8		S2=R4 + R4	8		3	4	2	3							1	1	1	1		
					1	2	3	2	3						1	1	1			
					3	1	2	2	3						1	1				
5				6	2	1	3							1			1	1		
6			7		2	1	1	3						1	1		1	1		
				4	1	1	1	2						1	1			1		
				5		1	2							1	1					
				8		1								1						
7						1								1						
8							1							1						
9								7												

Tabelle 4.13: Abläufe in einer superskalaren CPU mit Out-of-Order-Starten und Out-of-Order-Abschließen

Der erste Unterschied ergibt sich in Zyklus 3. Obwohl B4 angehalten hat, können wir B5 decodieren und ausgeben, weil er mit keinem anstehenden Befehl kollidiert. Allerdings bewirkt das Überspringen von Befehlen ein neues Problem. Angenommen, B5 verwendet einen Operanden, den der übersprungene Befehl B4 berechnet. Aus dem angegebenen Scoreboard lässt sich das nicht erkennen. Folglich müssen wir das Scoreboard erweitern, um die von den übersprungenen Befehlen durchgeführten Speicheroperationen zu erfassen. Dazu fügen wir eine zweite Bitmap mit einem Bit für jedes Register hinzu, um die von den angehaltenen Befehlen ausgeführten Speichervorgänge zu protokollieren. (Diese Zähler sind in Tabelle 4.13 nicht dargestellt.) Die Regel für das Starten von Befehlen ist nun zu erweitern, damit kein Befehl ausgegeben wird, dessen Operand laut Scheduler (Befehlsplaner) von einem Befehl zu speichern ist, der eigentlich vorher an der Reihe war, aber übersprungen wurde.

Sehen Sie sich noch einmal B6, B7 und B8 in Tabelle 4.12 an. Hier ist zu erkennen, dass B6 einen Wert in R1 berechnet, auf den B7 zurückgreift. Allerdings ist auch festzustellen, dass der Wert niemals verwendet wird, weil B8 das Register R1 überschreibt. Es gibt keinen triftigen Grund, in R1 das Ergebnis von B6 abzulegen. Zudem stellt R1 als Zwischenregister noch eine unglückliche Wahl dar, obwohl es für einen Compiler oder Programmierer, der an die sequenzielle Ausführung von Befehlen ohne überlappende Befehle gewöhnt ist, absolut sinnvoll erscheint.

In Tabelle 4.13 führen wir eine neue Technik zur Lösung dieses Problems ein: die **Registerumbenennung** (Register Renaming). Die intelligente Dekodiereinheit ändert die Verwendung von R1 in B6 (Zyklus 3) und B7 (Zyklus 4) auf ein geheimes Register S1 ab, das für den Programmierer nicht sichtbar ist. Nun lässt sich B6 gleichzeitig mit B5 starten. Moderne CPUs verfügen oft über Dutzende geheimer Register für die Registerumbenennung. Mit dieser Technik kann man oft WAR- und WAW-Abhängigkeiten vermeiden.

Bei B8 greifen wir erneut auf die Registerumbenennung zurück. Diesmal wird R1 in S2 umbenannt, sodass sich die Addition starten lässt, bevor R1 frei ist, d.h. am Ende von Zyklus 6. Stellt sich heraus, dass das Ergebnis diesmal wirklich in R1 stehen muss, kann man immer noch den Inhalt von S2 rechtzeitig dorthin kopieren. Noch besser ist es, wenn alle späteren Befehle, die auf das Register zugreifen, ihre Quellen in das Register umbenennen lassen, wo es tatsächlich gespeichert ist. In jedem Fall kommt die B8-Addition auf diese Weise früher zum Zuge.

Bei vielen echten Maschinen ist die Umbenennung tief in die Organisation der Register eingebettet. Es gibt dort viele geheime Register und eine Tabelle, die für den Programmierer sichtbaren Register auf die geheimen Register abbildet. Somit findet sich das echte Register, das beispielsweise für R0 benutzt wird, in Eintrag 0 dieser Abbildungstabelle. Auf diese Weise gibt es kein echtes Register R0, sondern lediglich eine Bindung zwischen dem Namen R0 und einem der geheimen Register. Diese Bindung ändert sich häufig während der Ausführung, um Abhängigkeiten zu vermeiden.

Liest man die vierte Spalte der Tabelle 4.13 von oben nach unten, erkennt man, dass die Befehle nicht in der Programmreihenfolge gestartet wurden. Und sie wurden auch nicht in der Programmreihenfolge abgeschlossen. Die Schlussfolgerung aus diesem Beispiel ist einfach: Durch Out-of-Order-Ausführung und Registerumbenennung konnten wir die Berechnung fast doppelt so schnell machen.

Die **Registerumbenennung**, also die Ersetzung eines Registers im Befehl durch ein Notizregister, beseitigt Abhängigkeiten zwischen den Befehlen und führt zu einer effizienteren Fließbandverarbeitung.

4.5.4 Spekulative Ausführung

Im vorherigen Abschnitt haben wir das Konzept der Umordnung von Befehlen zur Verbesserung der Leistung eingeführt. Wenn auch nicht explizit erwähnt, ist es dabei um die Umordnung innerhalb eines einzigen Basisblocks gegangen. Nun ist es an der Zeit, diesen Punkt eingehender zu untersuchen.

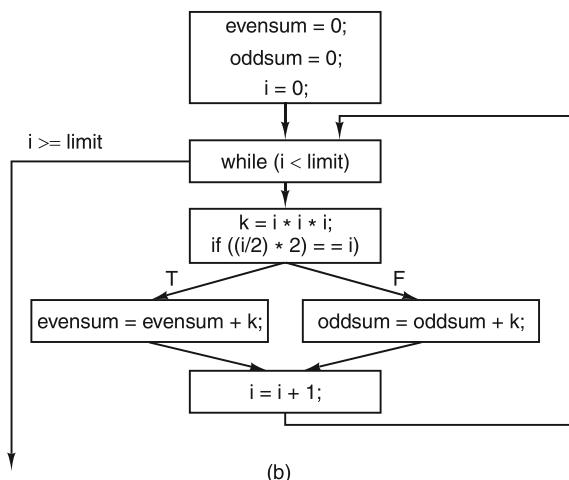
Computerprogramme kann man in **Basisblöcke** aufteilen, wobei jeder Basisblock aus einer linearen Codesequenz mit einem Eintrittspunkt oben und einem Austrittspunkt unten besteht. Ein Basisblock enthält keinerlei Steuerstrukturen (z.B. if- oder while-Anweisungen), sodass seine Übersetzung in Maschinensprache ohne Verzweigungen vor sich geht. Die Basisblöcke werden durch Steueranweisungen miteinander verbunden.

Ein Programm in dieser Form lässt sich wie in ►Abbildung 4.30 als gerichteter Graph darstellen. Hier berechnen wir die Summe der Kubikzahlen von geraden und ungeraden Ganzahlen bis zu einer gewissen Grenze und akkumulieren sie in *evensum* bzw. *oddsum*. Innerhalb eines jeden Basisblocks funktionieren die Umordnungstechniken des vorherigen Abschnitts.

```

evensum = 0;
oddsum = 0;
i = 0;
while (i < limit) {
    k = i * i * i;
    if (((i/2) * 2) == i)
        evensum = evensum + k;
    else
        oddsum = oddsum + k;
    i = i + 1;
}
    
```

(a)



(b)

Abbildung 4.30: (a) Ein Programmfragment und (b) der entsprechende Kontrollflussgraph

Problematisch ist, dass die meisten Basisblöcke kurz sind und nicht genügend Ansatzpunkte für eine effiziente Parallelität bieten. Folglich gehen wir im nächsten Schritt dazu über, die Umordnung über Basisblockgrenzen hinweg zuzulassen und so zu versuchen, die maximale Zahl von abgeschlossenen Befehlen pro Takt zu erreichen. Die größten Gewinne lassen sich erzielen, wenn man eine potenziell langsame Operation im Graphen nach oben verschieben kann, um sie früh zu starten. Dies kann ein LOAD-Befehl, eine Gleitkommaoperation oder gar der Beginn einer langen Abhängigkeitskette sein. Das Verschieben von Code nach oben über einen Sprung nennt man **Hoisting** (Anheben).

Stellen Sie sich vor, dass in Abbildung 4.30 alle Variablen in Registern gespeichert werden, außer *evensum* und *oddsum* (weil nicht genügend Register vorhanden sind). Es scheint dann sinnvoll zu sein, ihre LOAD-Befehle vor der Berechnung von *k* über die Schleife zu verschieben, damit sie früh starten, sodass die Werte bei Bedarf verfügbar sind. Natürlich wird in jeder Iteration nur einer dieser LOAD-Befehle gebraucht, der andere ist überflüssig. Wenn aber Cache und Speicher fließbandorientiert arbeiten und das Starten weiterer Befehle möglich ist, lohnt es sich dennoch so vorzugehen. Die Ausführung von Code, noch bevor bekannt ist, ob er überhaupt gebraucht wird, nennt man **spekulativer Ausführung** (Speculative Execution). Diese Technik setzt die Unterstützung von Compiler und Hardware sowie einige Architekturerweiterungen voraus. In den meisten Fällen übersteigt die Umordnung von Befehlen über mehrere Basisblockgrenzen hinweg die Fähigkeit der Hardware, sodass der Compiler die Befehle explizit verschieben muss.

Die spekulativen Ausführung wirft einige interessante Probleme auf. Zum einen ist es wichtig, dass die spekulativen Befehle keine unwiderruflichen Ergebnisse erzeugen, weil sich später herausstellen kann, dass sie nicht hätten ausgeführt werden sollen.

Beim Code in Abbildung 4.30 ist es sinnvoll, *evensum* und *oddsum* abzurufen, und es ist auch richtig, die Addition auszuführen, sobald *k* verfügbar ist (selbst vor der `if`-Anweisung), doch es wäre verkehrt, die Ergebnisse in den Speicher zurückzuschreiben. Um in komplizierten Codesequenzen zu verhindern, dass spekulativer Code vorschnell – und vielleicht unerwünscht – Register überschreibt, benennt man üblicherweise alle vom spekulativen Code verwendeten Zielregister um. Auf diese Weise werden lediglich Notizregister modifiziert, sodass kein Problem entsteht, wenn der Code letztendlich nicht mehr erforderlich ist. Andernfalls werden die Arbeitsregister in die echten Zielregister kopiert. Wie man sich vorstellen kann, ist ein Scoreboard, das alle diese Dinge verwalten muss, recht komplex. Mit entsprechender Hardware lässt sich das aber bewerkstelligen.

Codestücke können zur Optimierung **spekulativ** ausgeführt werden. Ausnahmen und schreibende Zugriffe müssen bis zur endgültigen Entscheidung aufgeschoben werden.

Allerdings führt spekulativer Code zu einem anderen Problem, das sich nicht durch Registerumbenennung lösen lässt. Was passiert, wenn ein spekulativ ausgeführter Befehl eine Ausnahme (Exception) verursacht? Ein schmerzliches, aber nicht verhängnisvolles Beispiel ist ein `LOAD`-Befehl, der einen Cache-Zugriffsfehler verursacht, wenn die Maschine mit einer großen Cachezeilengröße (z.B. 256 Byte) arbeitet und ihr Speicher weit langsamer als die CPU und der Cache ist. Stoppt ein unbedingt erforderlicher `LOAD`-Befehl, so gerät die Maschine über viele Zyklen hinweg in Stillstand, während die Cachezeile geladen wird. Allerdings ist es kontraproduktiv, die Maschine anzuhalten, nur um ein Wort abzurufen, das sich später als gar nicht benötigt entpuppt. Zu viele solcher „Optimierungen“ machen die CPU langsamer, als wenn sie überhaupt keine hätte. (Falls die Maschine mit einem virtuellen Speicher arbeitet, den *Kapitel 6* behandelt, könnte ein spekulativer `LOAD`-Befehl sogar einen Seitenfehler verursachen. Daraufhin ist eine Festplattenoperation erforderlich, um die benötigte Seite einzulesen. Seitenfehler können sich verheerend auf die Leistung auswirken. Deshalb ist es so wichtig, sie zu vermeiden.)

Viele moderne Maschinen verfügen über einen speziellen `SPECULATIVE-LOAD`-Befehl, der das Wort aus dem Cache zu lesen versucht und, wenn er es dort nicht findet, einfach aufgibt. Steht der Wert im Cache und wird er tatsächlich gebraucht, kann der Prozessor ihn verwenden. Befindet sich der Wert nicht im Cache, muss die Hardware ihn unverzüglich herbeischaffen. Stellt sich heraus, dass der Wert nicht benötigt wird, dann hat sich durch den Cache-Zugriffsfehler auch kein Nachteil ergeben.

Eine weit schlimmere Situation lässt sich mit der folgenden Anweisung veranschaulichen:

```
if (x > 0) z = y/x;
```

Hier sind *x*, *y* und *z* Gleitkommavariablen. Angenommen, alle Variablen werden vorab in Register eingelesen und die (langsame) Gleitkommadivision wird über den `if`-Test angehoben. Nun ist *x* leider 0 und die resultierende unzulässige Division durch null beendet das Programm. Unter dem Strich ist festzustellen, dass die Spekulation ein korrektes Programm zum Absturz gebracht hat. Noch schlimmer ist, dass der Programmierer ausdrücklich Code geschrieben hat, um diese Situation zu verhindern, und das Unglück ist trotzdem passiert. Sicherlich gehört das nicht zu den Dingen, die Glücksgefühle beim Programmierer aufkommen lassen.

Als mögliche Lösung kann man spezielle Versionen für Befehle vorsehen, die gegebenenfalls Ausnahmen auslösen. Zusätzlich fügt man jedem Register ein sogenanntes **Poison Bit** („Giftbit“) hinzu. Scheitert ein spezieller spekulativer Befehl, bewirkt er keinen Trap, sondern setzt das Poison Bit im Ergebnisregister. Wenn später ein normaler Befehl auf dieses Register zugreift, tritt der Trap auf (wie er sollte). Wird allerdings das Ergebnis niemals verwendet, wird das Poison Bit schließlich gelöscht und es entsteht kein Schaden.