2. Von-Neumann-Rechner und Prozessortechniken

2.1 Einleitung

In der vorherigen Kurseinheit (KE) haben wir uns mit der Technik von PCs aus der speziellen Sicht eines Anwenders oder PC-Nutzers beschäftigt. In den folgenden beiden Kurseinheiten wollen wir uns etwas allgemeiner mit den Computersystemen zugrunde liegenden Prinzipien und Strukturen beschäftigen. Dabei steht insbesondere die logische Organisation solcher Systeme, d.h. wie laufen die ausgeführten Operationen intern ab und welche Komponenten kommen dabei in welcher Form zum Einsatz, im Vordergrund. Hardwarerealisierungen, d.h. die zugrunde liegende elektronische Schaltungstechnik, wollen wir hingegen nicht betrachten, da diese für Wirtschaftinformatiker, für die hauptsächlich die Begriffe und Phänomene der Computerwelt wichtig sind, weniger von Bedeutung sind.

Als Einstieg in die Thematik wollen wir das ursprünglich allen Digitalrechnern zugrunde liegende Modell des von-Neumann-Rechners mit seinen Komponenten und deren Funktionen betrachten. Es folgt die Einführung der wichtigsten Charakteristika von Befehlssatzarchitekturen. Die Befehlssatzarchitektur beschreibt die Sicht des Programmierers auf den Rechner und wird daher auch als Programmiermodell bezeichnet. Danach werden wir auf die Unterschiede zwischen CISC (Complex Instruction Set Computer) und RISC (Reduced Instruction Set Computer) eingehen. Weiter werden wir die Umsetzung des von-Neumann-Prinzips durch Befehls-Pipelining behandeln. Dabei werden wir die auftretenden Pipeline-Konflikte sowie deren Lösungen betrachten. Der Rest dieser KE beschäftigt sich mit dem Skalar- und dem Superskalarprozessor. Dabei werden wir insbesondere Maßnahmen zur Beschleunigung dieser Prozessoren kennenlernen. Dazu gehört die effiziente Befehlsbereitstellung sowie Sprungvorhersagetechniken und spekulative Ausführung.

2.2 Das Prinzip eines Digitalrechners

Wenn wir über einen Computer bzw. einen Rechner reden, dann meinen wir damit heute immer einen digitalen Rechner. Ein Digitalrechner oder abkürzend nur Rechner oder Computer genannt, besteht aus drei Hauptkomponenten: Hardware, Software und Firmware, die wiederum jeweils in Teilkomponenten unterteilt werde können.

Hardware umfasst alle mechanischen und elektronischen Bauelemente und Baugruppen: Gehäuse, Stromversorgungen, Integrierte Schaltungen (integrated circuits – ICs) bestehend aus Platinen, Transistoren, usw.

Software umfasst alle Programme, die auf dem Rechner ablaufen. Dazu gehören das Betriebssystem, das die geordnete Bearbeitung aller Programme und die Zuteilung der erforderlichen Betriebsmittel sicherstellen muss, die Hilfsprogramme zur Erstellung von Programmen, wie Editoren, Übersetzer, Interpreter, Lader, usw. und schließlich die Anwenderprogramme.

Firmware nimmt eine Mittelstellung zwischen Hardware und Software ein. Sie besteht einerseits aus den Mikroprogrammen, die in Festwert- oder Schreib-/Lese-Speichern untergebracht sind und den "Befehlssatz" des Rechners, die sog. Maschinenbefehle, realisieren. Andererseits wird dazu häufig jedes Programm gezählt, das in einem Festwertspeicher abgelegt ist.

Aus Hardware-Sicht ist ein Rechner ein digitalelektronisches System, in das Daten als binär codierte Informationen eingegeben, dort gespeichert und verarbeitet werden. Das Ergebnis der Bearbeitung wird dann in unterschiedlichster Form ausgegeben, z.B. in Form binär kodierter Zeichen oder als

elektrische Signale zur Steuerung bestimmter Prozesse. Die Verarbeitung der Daten läuft nach einem vorgegebenen Programm ab, das zunächst selbst als binär codierte Information eingegeben und gespeichert werden muss. Es besteht aus einer Folge von (Maschinen-)Befehlen, die die jeweils nächste Operation und die dazu benötigten Operanden bestimmen.

Zur Eingabe von Daten und Programmen sowie zur Ausgabe verarbeiteter Daten und zur internen Kommunikation kommen zwei Prinzipien zur Anwendung. In heutigen PCs wird hauptsächlich das Prinzip der Punkt-zu-Punkt-Verbindung, die auch als Direktverbindung bezeichnet werden, eingesetzt. Dabei handelt es sich um eine direkte, unmittelbare Verbindung zwischen zwei Kommunikationsteilnehmern A und B. Verbunden werden interne Komponenten wie beispielsweise Prozessor und Grafikkarte oder periphere Geräte wie Massenspeicher, Drucker, Bildschirme und Tastaturen usw. Busse hingegen können mehr als zwei Teilnehmer miteinander verbinden. Sie werden heute im wesentlichen zur internen Verbindung zwischen Prozessor und Hauptspeicher (HSP) und zum prozessorinternen Datenaustausch eingesetzt. Ein Bussystem besteht in der Regel aus Adress-, Daten- und Steuerleitungen.

- Unter einer Adresse versteht man in der Digitaltechnik eine binär codierte Information zur eindeutigen Auswahl (Adressierung) eines bestimmten Speicherwortes, eines Registers oder einer Schnittstelle als Quelle oder Ziel eines Datentransportes.
- Datenleitungen übertragen die zu verarbeitenden Daten. Sie sind in der Regel bidirektional, d.h. Daten können zu einer Komponente hin oder von ihr weg transportiert werden.
- Auf Steuerleitungen werden für den Transport benötigte Informationen von einer Bussteuereinheit übertragen.

2.2.1 Das Modell des von-Neumann-Rechners

Dem überwiegenden Teil aller Rechner liegt nach wie vor ein Maschinenmodell zugrunde, das vom Mathematiker **J. von Neumann** bereits in den 1950-er entworfen wurde. Seine wesentlichen Komponenten sind:

- **Prozessor**, der Programme ausführt und dabei Daten verarbeitet. Er besteht aus einem **Steuerwerk** (auch Leitwerk genannt) und einem **Rechenwerk**;
- **Arbeitsspeicher** zur Ablage für Programme und Daten;
- **Ein-/Ausgabewerk** für die Kommunikation mit der Umwelt.

Prozessoren werden auch als **CPUs** (*central processing unit*) bezeichnet, und der Arbeitsspeicher heißt meistens auch **Hauptspeicher**. Rechenwerke sind auch als **ALUs** (*arithmetic logical unit*) oder Operationswerke bekannt, und statt Ein-/Ausgabewerken bzw. Geräten spricht man auch von der Peripherie.

Das Steuerwerk übernimmt die Koordination aller Abläufe innerhalb des Rechners. Es holt die Maschinenbefehle aus dem Hauptspeicher, interpretiert sie und setzt sie unter Berücksichtigung von Statusinformation in Steuerinformation für andere Komponenten um. Das Rechenwerk führt einfache mathematische Operationen aus. Der Arbeits- oder Hauptspeicher dient zur Ablage der Daten und Programme in Form von Bitfolgen. Er besteht aus Speicherzellen fester Wortlänge, die über eine Adresse jeweils einzeln angesprochen werden können.

Charakteristisch für von Neumann-Rechner sind neben der schon angegebenen Gliederung die folgenden Prinzipien:

- Der Speicher besteht aus einzelnen, gleich großen Speicherzellen, welche mit fortlaufenden Adressen versehen sind.
- Der Inhalt einer Speicherzelle wird über ihre Adresse angesprochen.
- Der Rechner verarbeitet Gruppen von Bits fester Länge.

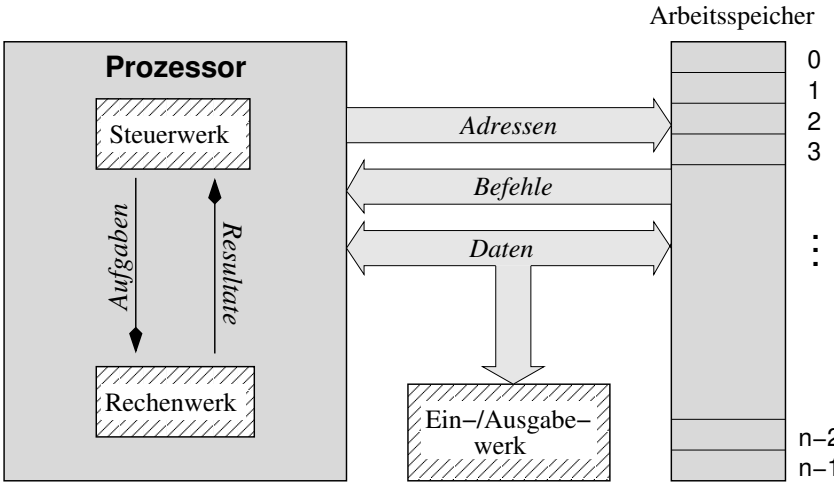


Abbildung 2.1: Modell des von-Neumann-Rechners

- Programme und Daten liegen in einem einheitlichen Speicher.
- Zur Bearbeitung von Aufgaben werden diese in Form von Programmen eingegeben und im Speicher abgelegt.
- Programme bestehen aus einzelnen Befehlen, die der Prozessor sequentiell abarbeitet.
- Die Verarbeitung von Befehlen geschieht normalerweise in der Reihenfolge ihrer Ablage im Speicher.
- Die normale Verarbeitungsreihenfolge der Befehle kann durch bedingte und unbedingte Sprungbefehle verändert werden. Hierdurch sind Verzweigungen im Ablauf des Programms in Abhängigkeit von Daten möglich.

Das hervorstechende Merkmal ist die Steuerung des Prozessors durch im Speicher abgelegte Programme. Diese **Speicherprogrammierbarkeit** erlaubt es, den Prozessor universell für die Lösung sehr unterschiedlicher Probleme anzuwenden, ohne die Struktur des Rechners zu verändern.

Eine Konsequenz der Gleichbehandlung von Daten und Programmen ist es, dass Programme selbst als Eingabe- oder Ausgabedaten anderer Programme auftreten können. In diesem Sinne ist auch die Bezeichnung „Daten“ an den Verbindungspfeilen von CPU und den Ein-/Ausgabewerken in Abb. 2.1 zu verstehen. Zu den ein- bzw. ausgegebenen Daten gehören auch Programme, die aber durch den Prozessor nicht direkt ausgeführt, sondern erst im Speicher abgelegt und dann ausgeführt werden.

Wie wir im Verlauf dieses Kurses noch sehen werden, weichen moderne Rechner in einigen Punkten von den oben genannten Prinzipien ab. Dennoch folgt nach wie vor das Groß aller Rechner vielen dieser Prinzipien, so dass man sie als „von-Neumann-Rechnerarchitekturen“ bezeichnet. Im nächsten Abschnitt wollen wir den von-Neumann-Rechner mit seiner Umgebung aus logischer Sicht etwas genauer betrachten und dabei die Funktion der Elemente erläutern.

2.2.2 Die Umgebung eines von-Neumann-Rechners

Wenn wir der Einfachheit halber annehmen, dass wir höhere Programmiersprachen als geeignetes Mittel zur Beschreibung von algorithmischen Problemlösungen einsetzen wollen, so reduziert sich die Aufgabe des Rechners auf die Ausführung solcher Hochsprachenprogramme. Leider sind diese aber zu komplex als dass sich mit der heutigen Technologie Prozessoren mit einem vernünftigen Kosten/Leistungsverhältnis herstellen ließen, welche z.B. Java-Programme direkt ausführen könnten. Daher stellt sich die Frage nach geeigneten Maschinensprachen, die effizient durch Hardware ausführbar sind.

Das von Neumann-Modell gibt bereits einige Hinweise, wie eine solche Sprache aussehen sollte. Zum einen bestehen die auszuführenden Programme dieses Modells aus Mengen von Befehlen, welche sequentiell ausgeführt werden. Konzeptuell bedeutet dies, dass sich der Prozessor wie ein Automat verhält, wobei die Eingaben aus dem gerade abzuarbeitenden Befehl bestehen. Der Zustand des Prozessors ist durch die Werte aller Speicherzellen innerhalb des Prozessors bestimmt, und die Ausgabe besteht aus den Steuersignalen, welche an die E/A-Geräte und den Speicher übergeben werden.

Es bleibt die Frage nach sinnvollen Befehlssätzen, welche einerseits einen Beitrag zur Implementation von höheren Programmiersprachen leisten, andererseits aber effizient in Hardware implementiert werden können. Um die Antworten auf diese Frage verstehen zu können, ist es nötig, die Umgebung des Prozessors genauer zu betrachten. Diese besteht aus verschiedenen Speichereinheiten und den Ein-/Ausgabegeräten, welche über sog. Busse mit der CPU verbunden sind. Abb. 2.2 gibt eine vereinfachte Sicht eines solchen Systems, wobei Bausteine, die den Busverkehr regeln, nicht dargestellt werden.

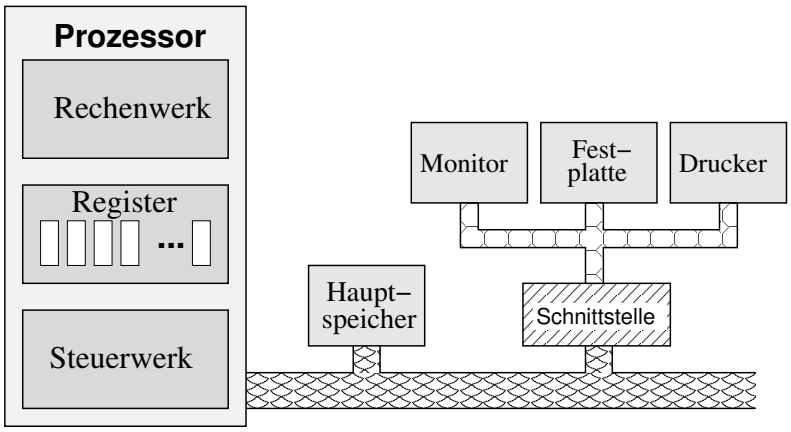


Abbildung 2.2: Rechner mit Bussystem

Im Folgenden wollen wir die Grundlagen von Speichern und der Ein-/Ausgabe soweit zusammenstellen, wie es zum groben Verständnis der Arbeit eines Prozessors nötig ist. Eine weitergehende Behandlung von Speichern findet in der nächsten Kurseinheit statt. Die Funktion eines Speichers besteht darin, Informationen unter bestimmten Adresse abzulegen um sie später wieder abrufen zu können. Diese **Adressen** verweisen auf **Speicherzellen**, die kleinste adressierbare Einheit des jeweiligen Speichers. Bei vielen Rechnern können die einzelnen Speicherzellen jeweils ein Byte (= 8 Bit) Information aufnehmen, man spricht dementsprechend von Byte-adressierten Speichern. Bei manchen neueren Architekturen ist nicht einmal der Zugriff auf einzelne Bytes mehr möglich, sondern es werden stets mehrerer Bytes auf einmal transportiert.

Ein abstraktes Modell des Speichers

Mathematisch-abstrakt gesehen ist ein Speicher eine Abbildung von Adressen auf Bitgruppen. Der Definitionsbereich dieser Funktion, also die Menge aller möglichen Adressen, heißt **Adressraum**. Das Lesen einer Speicherzelle ist einfach die Anwendung dieser Funktion auf die Adresse der Speicherzelle. Die Operation des Schreibens einer Speicherzelle mit Adresse a bewirkt eine Ersetzung des Funktionswertes der Adresse a, wobei die Werte aller anderen Adressen unverändert bleiben.

Die **Speicherkapazität** eines Speichers, also die Größe der speicherbaren Informationsmenge, ergibt sich als das Produkt der Anzahl der Speicherzellen und der Kapazität jeder einzelnen Zelle. Gängige Maßeinheiten sind Bit und Byte sowie Größen, die sich durch Vorstellen der Präfixe K(ilo) $= 2^{10}$, M(ega) $= 2^{20}$ und G(iga) $= 2^{30}$ hieraus ableiten lassen.

Aufgabe 2.1 *Adresslänge*
Ein Speicher mit 4 Gigabyte Kapazität sei in Speicherzellen zu je 8 Byte aufgeteilt. Wie lang ist jede Adresse mindestens? ◇

Speicherklassen

Neben den strukturellen Eigenschaften eines Speichers sind in der Praxis weitere technische Eigenschaften wichtig:

- die **Zugriffszeit**, also die Zeit, die für das Lesen oder Schreiben einer Speicherzelle benötigt wird;
- die **Zykluszeit**, also die Zeit, die zwischen zwei aufeinander folgenden Schreib- oder Lesezugriffen liegen muss;
- die **Zugriffsmethode**, die Art des Zugriffs, z.B.
  - sequentieller Zugriff, wie bei einem Magnetband, oder
  - wahlfreier Zugriff auf einzelne Speicherzellen, wie bei elektronischen Speichermedien (**RAM**, *random access memory*).

Weiterhin ist nach der Dauerhaftigkeit des Speichers zu unterscheiden, also ob der Speicher auch nach dem Ausschalten oder Ausfall der Versorgungsspannung seinen Wert behält. Gilt dies, so spricht man von einem **Festwertspeicher**.

Schließlich stellt sich die Frage, inwieweit der Inhalt von Speicherzellen verändert werden kann. Mit der Bezeichnung RAM-Speicher impliziert man, dass der Speicher sowohl les- als auch schreibbar ist. Da hingegen behalten irreversible Festwertspeicher (**ROM**, *read only memory*) ihren einmal eingespeicherten Wert bei und sind daher besonders geeignet, grundlegende Mikroprogramme oder Teile des Betriebssystems aufzunehmen. Geschieht die Festlegung der Speicherinhalte nicht direkt bei der Erzeugung des Chips, sondern später elektrisch mit Hilfe von speziellen Programmiergeräten, so spricht man von programmierbaren Bausteinen, den **PROMs** (*programmable read only memory*). Bei **EPROM**- und **EEPROM**-Chips lassen sich die Speicherinhalte mit Hilfe von optischen bzw. elektrischen Verfahren wieder löschen, so dass die Chips danach wieder neu programmiert werden können. Im Vergleich zu RAM-Chips benötigt das Löschen und Schreiben von EEPROMs allerdings um einen Faktor von 10^3 mehr Zeit. Der Lesezugriff auf die verschiedenen Chips ist hingegen vergleichbar schnell.

Aus Sicht des Prozessors und seines Befehlssatzes unterscheiden wir grundsätzlich drei Klassen von Speichern:

- Register, die sehr schnell sind und in begrenzter Anzahl prozessorintern vorliegen;
- einen schnellen großen Hauptspeicher;
- externe Speicher mit sehr großer Kapazität, die aber relativ langsam sind.

Beim **Hauptspeicher** (auch Arbeitsspeicher genannt) handelt es sich um den Speicher, welcher die im von Neumann-Modell beschriebenen Speicherfunktionen erfüllt, d.h. aus ihm holt der Prozessor die auszuführenden Befehle nebst ihren Daten. Als **Register** bezeichnen wir spezielle interne Speicherzellen, auf die sehr schnell zugegriffen werden kann. Sie dienen beispielsweise zur Aufnahme der Operanden bei arithmetischen Operationen.

Extrem große Datenmengen werden am besten auf großen Hintergrundspeichern untergebracht. Der Prozessor hat keinen direkten Zugriff auf diese Speicher, so dass benötigte Daten vor der Verarbeitung erst in den Hauptspeicher eingelesen werden müssen. Der Datenaustausch des Prozessors mit externen Speichern vollzieht sich daher ähnlich wie mit einem Ein-/Ausgabegerät und wird dementsprechend mit zur E/A gezählt.

Virtuelle Adressräume

Sehen wir einmal von der Ein-/Ausgabe ab, so sind aus Sicht des Prozessors in unserem Modell nur zwei Adressräume zu unterscheiden, nämlich Register und Hauptspeicher. Prinzipiell könnten diese sogar in einem Adressraum vereinigt werden, indem man die Adressen so strukturiert, dass aus ihnen hervorgeht, ob sie sich auf ein Register oder den Hauptspeicher beziehen. Diese Betrachtung der Vereinigung von Adressräumen macht allerdings noch einmal klar, dass es aus rein logischer Sicht keine

Rolle spielt, was beim Zugriff auf eine Adresse physikalisch geschieht. Sieht man vom Zeitfaktor ab, so ist es für die Abarbeitung eines Programmes auf einem Prozessor unerheblich, ob der zu einer Adresse gehörende Wert z.B. von einem Speicherchip oder von einer Festplatte geholt wird. Man sagt daher, dass die vom Prozessor verwendeten Hauptspeicheradressen einen **virtuellen** (logischen) **Adressraum** bilden. Diese virtuellen Adressen sind von den Adressen zu unterscheiden, unter denen sich ein Wert letztendlich auf einem physikalisch vorhandenen Speicher lokalisieren lässt, wie z.B. Zylinder, Spur und Position auf einer Festplatte. Die Umsetzung der logischen Adressen in physikalische Adressen zählt zu den Aufgaben der virtuellen Speicherverwaltung innerhalb Betriebssystems und wird in der folgenden KE behandelt.

Bytes und Wörter

Viele Rechner unterstützen zusätzlich zur Byte-adressierung auch die Einteilung des Speichers in Gruppen von mehreren Byte, den sog. **Wörtern**, sowie ihren Abkömmlingen, den *Halb-* oder *Lang-*Wörtern mit ihrer jeweils halben bzw. doppelten Bitlänge im Verhältnis zu einem Wort. Entsprechend kann man dann bei verschiedenen Befehlen angeben, ob sie sich auf Bytes oder andere Bitgruppen beziehen.

Bussysteme

Die Kommunikation zwischen den verschiedenen Bestandteilen eines Rechensystems erfolgt in erster Linie über Bündel von Leitungen, die sog. **Busse**. Ein Bus ist dadurch charakterisiert, dass er zu einem Zeitpunkt maximal eine Verbindung vom Typ „Ein Sender sendet an *n* Empfänger“ ermöglicht. Mit anderen Worten, der Bus wird jeweils maximal einer der angeschlossenen Einheiten zugeteilt, die dann senden darf. Der prinzipielle Vorteil der Verwendung von Bussen liegt im geringen Aufwand an Verdrahtung und Anschlüssen bei einem durchgehenden Leitungsbündel im Vergleich zu anderen Verbindungstopologien. Mit Hilfe von zeitlich versetzter Nutzung (Zeitmultiplex) können diese Ressourcen noch effizienter genutzt werden.

Je nach dem Verwendungszweck der transportierten Information unterscheidet man Adress-, Daten- und Steuerbusse. Die Standard-CPUs besitzen einen Satz von externen Bussen dieser Art, über die der Speicher und die Peripherie an die CPU angeschlossen sind. CPU-intern gibt es zwar viele Datenbusse, aber wenige Adressbusse, die Auswahl eines Ziel- oder Quellregisters erfolgt meistens direkt durch Schaltung der entsprechenden Leitungen. Spricht man von *dem* Daten-, Adress- oder Steuerbus, so meint man die externen Busse, welche die CPU mit den anderen Systemkomponenten verbinden. Während Daten- und Steuerbus bidirektional sind, reicht oft ein unidirektionaler **Adressbus**. Über diesen teilt die CPU dem Hauptspeicher die Adresse einer zu lesenden oder zu schreibenden Speicherzelle mit. Der gelesene oder zu schreibende Wert, sei es im Endeffekt ein Datenwert oder ein Teil eines auszuführenden Befehls, wird über den **Datenbus** transportiert. Über den **Steuerbus** sendet die CPU Steuersignale an die anderen Einheiten, oder sie empfängt Statussignale und Rückmeldungen über durchgeführte Aktionen. Adress-, Daten- und Steuerbus werden manchmal zum **Systembus** zusammengefasst.

Harvard-Architektur

Bei der Standard-von-Neumann-Architektur mit einem Hauptspeicher und einem Daten- und Adressbus kann diese Verbindung schnell zum Engpass werden. Alle Daten und Maschinenbefehle müssen über die gleiche Verbindung zwischen Prozessor und Hauptspeicher transportiert werden. Dieser Engpass wird deshalb oft auch als von-Neumann-Flaschenhals bezeichnet. Verschärft wird dieser Engpass dadurch, dass die Befehlsausführung durch den Prozessor heute wesentlich schneller abläuft als der Zugriff auf den Hauptspeicher.

Eine Verbesserung schafft die **Harvard-Architektur**, bei der Programme und Daten in zwei verschiedenen Speichern abgelegt sind, die jeweils über einen eigenen Daten-, Adress- und Steuerbus mit der CPU verbunden sind. Heute trifft man meistens Systeme, bei denen Befehle und Daten zwar in einem Hauptspeicher liegen, der größte Teil der Speicherzugriffe aber über getrennte sog. Befehls-

und Datencaches erfolgt. Diese Caches sind besonders schnelle Speicher, die zum Teil auf dem CPU-Chip integriert sind.

E/A-Adressen

Ähnlich wie beim Speicher sind auch zum Ansprechen der E/A-Geräte Kennungen (Nummern) zur eindeutigen Identifikation nötig. Diese bezeichnet man auch als Adressen, wobei statt dem Lesen oder Schreiben einer Speicherzelle nun der Austausch von Daten mit einem bestimmten Gerät steht. Bei der Wahl der Adressierung der E/A-Geräte gibt es dabei zwei Möglichkeiten. Zum einen kann man die E/A-Geräte als einen getrennten Adressraum betrachten, dessen Adressen, die **E/A-Ports**, von 0 bis $m-1$ reichen, wobei m die maximale Anzahl unterstützter E/A-Geräte ist.

Häufiger setzt man aber die **speicherorientierte Ein-/Ausgabe** *memory mapped I/O* ein, bei der ein Abschnitt der Hauptspeicheradressen, die sog. **I/O page**, für den Zugriff auf E/A-Geräte benutzt wird. Der Vorteil dieser Gleichbehandlung von Speichermedien und der E/A besteht darin, dass nur eine Sorte von Adressen über den Adressbus ausgegeben werden muss. Erforderlich ist dabei jedoch eine Einrichtung, welche anhand der Adresse unterscheiden kann, ob ein Zugriff auf den Hauptspeicher oder ein E/A-Geräte erfolgt.

2.2.3 CISC- und RISC-Architekturen

Bei der Entwicklung der Großrechner in den 1960er und 1970er Jahren hatten technologische Bedingungen wie der teure und langsame Hauptspeicher zu einer immer größeren Komplexität der Befehle und damit der Architekturen von Rechnern geführt. Um den teuren Hauptspeicher optimal zu nutzen, wurden komplexe Maschinenbefehle entworfen. Ein solcher Maschinenbefehl codierte eine mächtige und umfassende Operation mit einem einzelnen Opcode. Damit konnte auch der im Verhältnis zur Ausführungsgeschwindigkeit des Prozessors langsame Speicherzugriff überbrückt werden, denn ein Maschinenbefehl umfasste genügend Operationen, um die Zentraleinheit für mehrere, eventuell sogar mehrere Dutzend Prozessortakte mit wenigen Speicherzugriffen zu beschäftigen.

Eine auch heute noch übliche Implementierungstechnik, um den Ablauf solcher komplexen Maschinenbefehle zu steuern, ist die **Mikroprogrammierung**, bei der ein Maschinenbefehl durch eine Folge von Mikrobefehlen implementiert wird. Diese Mikrobefehle stehen in einem Mikroprogrammspeicher innerhalb des Prozessors und werden von einer Mikroprogrammsteuereinheit interpretiert. Dieser Mikroprogrammspeicher ist in der Regel als ROM ausgeführt, sodass der Prozessor zwar *mikroprogrammiert*, aber für den Anwender nicht *mikroprogrammierbar* ist.

Die Komplexität der Großrechner zeigte sich in mächtigen Maschinenbefehlen, umfangreichen Befehlssätzen, vielen Befehlsformaten, Adressierungsarten und spezialisierten Registern. Rechner mit diesen Architekturcharakteristika wurden später mit dem Akronym **CISC** (*Complex Instruction Set Computers*) bezeichnet. Solche CISC-Charakteristika zeigten sich auch bei den Intel-80x86- und den Motorola-680x0-Mikroprozessoren, die ab Ende der 1970er Jahre entstanden. Da jedoch die mittels Mikroprogrammierung realisierten Maschinenbefehle nur nacheinander, also sequentiell, ausgeführt werden konnten, war die Auslastung der einzelnen Prozessor-Funktionseinheiten schlecht. Um die Ausführung der Maschinenbefehle zu parallelisieren, entwickelte sich daher etwa 1980 ein zu CISC gegenläufiger Trend, der die Prozessorarchitekturen bis heute maßgeblich beeinflusst hat: das **RISC** (*Reduced Instruction Set Computer*) genannte Architekturkonzept.

Bei der Untersuchung von Maschinenprogrammen war beobachtet worden, dass manche Maschinenbefehle und komplexe Adressierungsarten wie sie in CISC realisiert sind, fast nie verwendet wurden. Komplexe Befehle lassen sich durch eine Folge einfacher Befehle ersetzen, komplexe Adressierungsarten entsprechend durch eine Folge einfacherer Adressierungsarten. Die vielen unterschiedlichen Adressierungsarten, Befehlsformate und -längen von CISC-Architekturen erschwerten die Codegenerierung durch den Compiler oder Übersetzer eines Programms. Das komplexe Steuerwerk, das notwendig war, um einen großen Befehlssatz in Hardware zu implementieren, benötigte viel Chip-Fläche und führte zu langen Entwicklungszeiten.

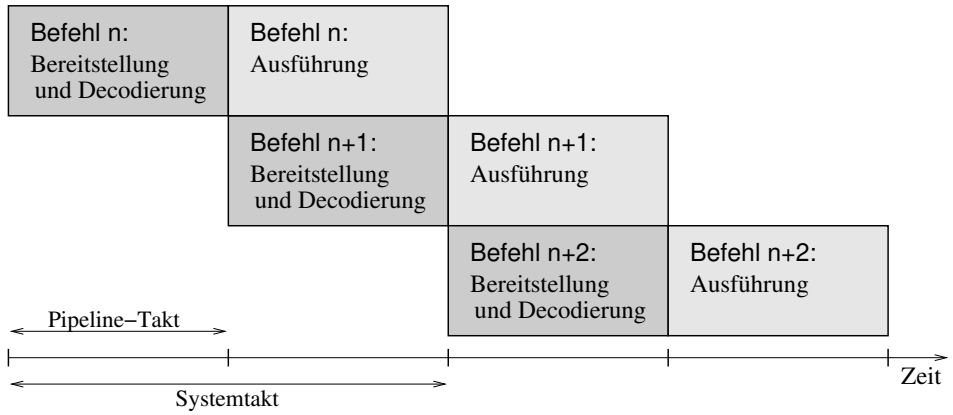


Abbildung 2.3: Überlappende Befehlsbearbeitung (Befehlspipelining)

Das RISC-Architekturkonzept wurde Ende der 1970er Jahre mit dem Ziel entwickelt, durch vereinfachte Architekturen Rechner schneller und preisgünstiger zu machen. Außerdem wurde zu diesem Zeitpunkt die Speichertechnologie billiger und erste Mikroprozessoren konnten einfacher auf Silizium-Chips implementiert werden. Die dem RISC-Architekturkonzept zugrunde liegende wichtigste Neuerung ist die Anwendung des sog. Pipelining-Prinzips. Pipelining bedeutet soviel wie Fließbandbearbeitung, die hier auf dem gleichen Prinzip beruht wie die schrittweise Fertigung von Waren in der Industrie: Der zu erledigende Auftrag wird in kleinere Teilaufträge zergliedert, die unabhängig voneinander an verschiedenen Stationen ausgeführt werden können. Jeder einzelne Befehl durchläuft nacheinander die verschiedenen, seriell angeordneten Stufen.

Charakteristisch für die Fließbandverarbeitung ist, dass mit der Bearbeitung eines neuen Auftrags nicht gewartet wird, bis der vorherige Auftrag vollständig abgearbeitet wird. Statt dessen werden in einem einheitlichen Takt der Pipeline neue Aufträge zugeführt und sich im System befindliche, noch nicht vollständig fertige Aufträge an die jeweils nachfolgende Stufe übergeben. Zur gleichen Zeit werden also mehrere Befehle im Prozessor bearbeitet, die Verarbeitungszeiten von aufeinander folgenden Befehlen überlappen sich.

Voraussetzung für die Anwendung des Pipelinings ist die Zerteilbarkeit der Maschinenbefehle in Teilaufträge sowie eine möglichst einfache und gleichförmige Gestaltung *aller* Maschinenbefehle (Opcodes). Alle Maschinenbefehle sollten so implementierbar sein, dass deren Ausführung auf eine feste Anzahl von Takten begrenzt ist, so dass im Optimalfall pro Prozessortakt die Ausführung eines Maschinenbefehls beendet wird. Durch die Implementierung mittels einer Befehlspipeline kann auf die Mikroprogrammierung verzichtet werden. Die Befehle werden in einem einzigen Pipelinetakt decodiert, im nächsten ausgeführt usw. Mikroprogrammierung und Pipelining sind jedoch keine sich gegenseitig ausschließende Konzepte. Moderne PC-Prozessoren von Intel oder AMD verbinden beide Konzepte miteinander.

Folgende Eigenschaften charakterisieren frühe RISC-Architekturen:

- Der Befehlssatz besteht aus wenigen, unbedingt notwendigen Befehlen (Anzahl ≤ 128) und Befehlsformaten (Anzahl ≤ 4) mit einer einheitlichen Befehlslänge von 32 Bit und mit nur wenigen Adressierungsarten (Anzahl ≤ 4). Damit wird die Implementierung des Steuerwerks erheblich vereinfacht und auf dem Prozessor-Chip Platz für weitere begleitende Maßnahmen geschaffen.

- Eine große Zahl von mindestens 32 allgemein frei verwendbaren Registern, das sind prozessorinterne sehr schnell beschreibbare und lesbare Speicherplätze. Man spricht daher bei RISC-Prozessoren auch von einer LOAD/STORE-Architektur.

- Der Zugriff auf den Hauptspeicher erfolgt nur über Lade-/Speicherbefehle. Alle anderen Befehle, d.h. insbesondere auch die arithmetischen Befehle, beziehen ihre Operanden aus den Registern und speichern ihre Resultate in Registern. Dieses Prinzip der Register-Register-Architektur

ist für RISC-Rechner kennzeichnend und hat sich heute bei allen neu entwickelten Prozessorarchitekturen durchgesetzt.

- Weiterhin wurde bei den frühen RISC-Rechnern die Überwachung der Befehls-Pipeline von der Hardware in die Software verlegt, d.h. Abhängigkeiten zwischen den Befehlen und bei der Benutzung der Ressourcen des Prozessors mussten bei der Codeerzeugung bedacht werden.

Die RISC-Charakteristika galten zunächst nur für den Entwurf von Prozessorarchitekturen, die keine Gleitkomma- und Multimediabefehle umfassten, da die Chip-Fläche einfach noch zu klein war, um solch komplexe Einheiten aufzunehmen. Inzwischen können natürlich auch Gleitkomma- und Multimediaeinheiten auf dem Prozessor-Chip untergebracht werden. Gleitkommabefehle benötigen nach heutiger Implementierungstechnik üblicherweise drei Takte in der Ausführungsstufe einer Befehls-Pipeline. Da die Gleitkommaeinheiten intern wiederum als Pipelines aufgebaut sind, können sie jedoch ebenfalls pro Takt ein Resultat liefern.

RISC-Prozessoren, die das Entwurfsziel von durchschnittlich *einer* Befehlsausführung pro Takt erreichen, werden als **skalare RISC-Prozessoren** bezeichnet. Doch gibt es heute keinen Grund, bei der Forderung nach *einer* Befehlsfertigstellung pro Takt stehen zu bleiben. Die Superskalartechnik ermöglicht es heute gleichzeitig mehrere Befehle von einer Reihe von Ausführungseinheiten zu verarbeiten, wodurch mehrere Befehlsausführungen pro Takt beendet werden können. Solche Prozessoren werden als **superskalare (RISC)-Prozessoren** bezeichnet, da die oben definierten RISC-Charakteristika auch heute noch weitgehend beibehalten werden.

2.2.4 Betrachtungsebenen eines Rechners

Das von-Neumann-Rechnermodell beschreibt nur eine der verschiedenen Ebenen eines Rechners, nämlich wie sich der Rechner dem Implementierer von Maschinenprogrammen darstellt. Um die Betrachtung eines Rechners zu vereinfachen, kann man verschiedene Betrachtungsebenen unterscheiden:

1. die Schaltwerksebene
2. die Mikroprogrammebene
3. die Maschinenprogrammenebene
4. die Assemblerebene
5. die Betriebssystemebene
6. die Ebene höherer Programmiersprachen wie z.B. Java, C oder C++
7. die Anwendungsebene

Die technische Realisierung der *Schaltwerksebene* erfolgt durch elektronische Baugruppen, ist also Teil der Hardware des Systems, die nicht Gegenstand dieses Kurses sein soll. Die darüber liegenden Ebenen werden überwiegend als Software realisiert. Die erste dieser Ebenen bilden die *Mikroprogramme*. Diese Ebene tritt natürlich nur bei CISC-Rechnern auf, denn bei den RISC-Rechnern wird die Pipeline in Hardware realisiert. Durch die Mikroprogrammierung werden dem Programmierer Maschinenbefehle zur Verfügung gestellt, die intern als Folgen einfacherer Befehle realisiert sind. Die Ebene der *Maschinenprogramme* ist die in Abschnitt 2.2.1 skizzierte Sicht des Rechners. Typische Maschinenbefehle sind die Addition des Inhalts zweier Speicherzellen oder bedingte Sprungbefehle. In den ersten Rechnergenerationen wurden die Befehle dieser Ebene direkt durch Hardware ausgeführt. Aus diesem historischen Grund nennt man die Befehlssätze dieser Ebene immer noch Maschinensprachen, spricht aber manchmal von „konventionellen Maschinensprachen“, um sie von den Mikroprogrammbefehlssätzen, der eigentlichen *Maschinen*sprache eines mikroprogrammierten Prozessors, zu unterscheiden.

Die durch das Steuerwerk interpretierten Maschinenbefehle liegen in binär kodierter Form, also als Bitmuster, vor. Diese Form ist zwar zur Interpretation durch Maschinen geeignet, für den Menschen aber kaum lesbar, sodass symbolische Maschinensprachen entwickelt wurden, sog. *Assemblersprachen*. Diese erlauben das Ansprechen von Befehlen und Speicherzellen mit Hilfe von Kürzeln (**Mnemonics**) und symbolischen Bezeichnungen für Konstanten und Adressen. Kennzeichnend für Assemblerprogramme ist, dass sie sich mit Hilfe eines Übersetzungsprogrammes, dem sog. **Assembler**, strukturgleich in Maschinenprogramme übersetzen lassen. Assemblersprachen sind daher im wesentlichen eine bequemere Notation von Maschinensprachen.

Bei der *Betriebssystemebene*, die Sie im parallel laufenden Kurs dieses Moduls kennen lernen, handelt es sich um eine noch weitergehende Abstraktion von der vorhandenen Hardware der jeweiligen Rechnerfamilie. Konzepte dieser Ebene im Vergleich zu den vorherigen Ebenen sind vor allem:

- virtuelle Speicher, die von dem physikalisch vorhandenen Speicher abstrahieren;
- Dateisysteme zur strukturierten Speicherung und Ein-/Ausgabe von Programmen und Daten;
- Prozesse, welche Aufträge an das Rechensystem darstellen, die teilweise gleichzeitig oder überlappend bearbeitet werden können.

In einer noch weitergehenden Abstraktion erlauben **höhere Programmiersprachen** wie Java oder C das Schreiben von Programmen, die auf unterschiedlichen Rechnern effizient ausgeführt werden können. Dies wird durch maschinenunabhängige Kontrollstrukturen, Datentypen und Operationen erreicht. Diese Sprachen stellen mithin abstrakte Berechnungsmodelle dar, in denen maschinenunabhängige Algorithmen zur Lösung von Problemen formuliert werden können.

Die oberste Ebene eines Rechensystems bilden die **Anwendungsprogramme**. Beispiele sind Textverarbeitungssysteme, Datenbanksysteme oder Programme zur Auswertung von Messwerten oder zur Steuerung eines Fahrzeugs. Solche Programme stellen also idealerweise die Lösung des Problems dar, für das der Rechner eingesetzt wird. Wegen der Diversität der Anforderungen unterscheiden sich die Anwendungsprogramme und ihre Bedienung in einem hohen Maß, obwohl sich inzwischen einige immer wiederkehrende Teilaufgaben, wie die Programmierung von Bedienoberflächen oder Datenbankfunktionen, herauskristallisiert haben.

Die dargestellte Hierarchie stellt in verschiedener Hinsicht eine Vereinfachung dar. In der Praxis kommt es durchaus vor, dass Teile der Funktionalität einer der niederen Schichten mit Hilfe einer höheren Schicht realisiert werden, z.B. sind große Teile des UNIX-Betriebssystems in C geschrieben. Weiter ließe sich über die Einführung von weiteren Ebenen diskutieren. So könnte man zwischen den höheren Programmiersprachen und den Anwendungen noch die „Programmiersprachen der vierten Generation“ einführen, welche die Erstellung bestimmter Applikationen wie Datenbanken oder graphische Bedienoberflächen stark vereinfachen, indem sie Bibliotheken mit entsprechenden Funktionen bereitstellen. Da es uns hier aber nicht um eine Klassifikation, sondern um das Verständnis geht, werden wir es bei den sieben Schichten belassen.

Im nun folgenden Kapitel wollen wir uns mit Rechnerarchitektur beschäftigen. Die Rechnerarchitektur soll eine optimale Grundlage für die Ausführung der Anwendungen bilden. Dabei überdeckt die Rechnerarchitektur im wesentlichen die zuvor bezeichneten Schichten 1 bis 3, wobei die Grenzen nicht exakt festzulegen sind. So verfügen moderne Prozessoren zum Beispiel über eine Einrichtung zur Unterstützung des Betriebssystems auf der Ebene 5 bei der Verwaltung des Speichers.

2.3 Rechnerarchitektur

Das sog. Programmiermodell eines Prozessors stellt die oberste Ebene der Rechnerarchitektur dar. Das Programmiermodell beschreibt die Sicht eines Systemprogrammierers auf den Prozessor und hat daher entscheidenden Einfluss auf die Leistung und insbesondere die Einsatzmöglichkeiten eines Rechners. Dazu beinhaltet das Programmiermodell alle notwendigen Details, um ablauffähige Maschinenprogramme für den betreffenden Prozessor zu erstellen. Natürlich muss auch der Compiler, der die Hochsprachenprogramme in Maschinensprache übersetzen soll, dieses Programmiermodell kennen. Da das Programmiermodell im Wesentlichen durch den Befehlssatz des Prozessors festgelegt ist, spricht man von der *Befehlssatzarchitektur* (Instruction Set Architectur – ISA) oder einfach auch nur von der *Architektur* eines Prozessors. Der Begriff Befehlssatzarchitektur und der etwas allgemeiner klingende Begriff Architektur werden also synonym verwendet. Die Befehlssatzarchitektur beschreibt nur das für den Anwender sichtbare äußere Verhalten des Prozessors, sie macht keine Aussage über seine Implementierung, die entweder durch die *logische Organisation* oder durch die tatsächliche *technologische Realisierung* beschrieben werden kann (Abb. 2.4).

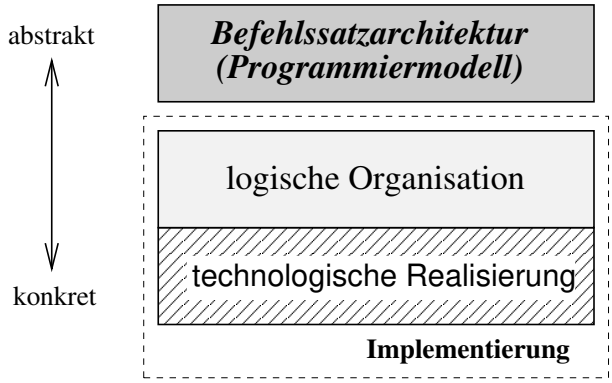


Abbildung 2.4: Ebenen der Rechnerarchitektur

Ähnlich wie ein Architekt zunächst mit dem Bauherrn die gewünschten Eigenschaften eines Bauwerks festlegt und diese dann optimal auf die spätere Nutzung abstimmt (z.B. Anzahl, Größe, Ausstattung und Anordnung der benötigten Räume), geht auch ein *Rechnerdesigner* oder Systemingenieur systematisch an den Entwurf eines Prozessors. Zunächst muss die Befehlssatzarchitektur des Prozessors festgelegt werden. Sie beschreibt die für die spätere Anwendung benötigten prozessorinternen Speichermöglichkeiten, Operationen und Datenformate. Aus dieser Architekturbeschreibung leitet der Rechnerdesigner dann eine logische Struktur zur Implementierung ab, die auch als *Mikroarchitektur* bezeichnet wird. Dabei legt er fest, welche Funktionseinheiten (z.B. Register(sätze), ALUs, Multiplexer usw.) benutzt werden sollen, welche Datenpfade (data path) zwischen den Funktionseinheiten vorhanden sein sollen und wie alle diese Komponenten koordiniert werden (control path). Die Umsetzung dieser logischen Organisation in einer bestimmten Hardware-Technologie bezeichnet man als technologische Realisierung (in Analogie zu einem Bauwerk wären dies die verwendeten Baumaterialien).

2.3.1 Prozessorarchitektur, Mikroarchitektur und Programmiermodell

Eine **Prozessorarchitektur**, die auch als die **Befehlssatz-Architektur** (ISA) oder das **Programmiermodell** eines Prozessors bezeichnet wird, definiert die Grenze zwischen Hardware und Software. Sie umfasst den für den Systemprogrammierer und für den Compiler sichtbaren Teil des Prozessors. Zur Befehlssatz-Architektur gehören neben dem Befehlssatz (Menge der verfügbaren Befehle) das Befehlsformat, die Adressierungsarten, das System der Unterbrechungen und das Speichermodell, das sind die Register und der Adressraumaufbau. Eine Prozessorarchitektur betrifft jedoch keine Details der Hardware und der technischen Ausführung eines Prozessors, sondern nur sein äußeres Erschei-

nungsbild. Die internen Vorgänge werden ausgeklammert.

Eine **Mikroarchitektur** (entsprechend dem englischen Begriff *microarchitecture*) bezeichnet die Implementierung einer Prozessorarchitektur in einer speziellen Verkörperung der Architektur – also in einem Mikroprozessor. Dazu gehören die Hardware-Struktur und der Entwurf der Kontroll- und Datenpfade, sowie die Art und Stufenzahl des Befehls-Pipelinings und der Grad der Verwendung der Superskalartechnik. Der Begriff Superskalartechnik bezeichnet eine bestimmte Art der Pipelinetechnologie, die wir im Folgenden noch genauer kennen lernen werden. Zur Mikroarchitektur gehören weiter die Art und Anzahl der internen Ausführungseinheiten eines Mikroprozessors sowie Einsatz und Organisation von Primär-Cache-Speichern. Primär-Cache-Speicher ist sehr schneller On-Board-Speicherplatz, der neben dem Hauptspeicher häufig benötigte Daten zusätzlich zwischenspeichert.

Die Mikroarchitektur definiert also die logische Organisation eines Prozessors. Diese Eigenschaften werden von der Befehlssatzarchitektur nicht erfasst. Systemprogrammierer und optimierende Compiler benötigen jedoch auch die Kenntnis von Mikroarchitektureigenschaften, um effizienten Code für einen speziellen Mikroprozessor zu erzeugen.

Architektur- und Implementierungstechniken werden beide im Folgenden als **Prozessortechniken** bezeichnet. Die Architektur macht die Benutzerprogramme von den Mikroprozessoren, auf denen sie ausgeführt werden, unabhängig. Alle Mikroprozessoren, die derselben Architekturspezifikation folgen, sind binär kompatibel zueinander. Die Implementierungstechniken zeigen sich bei den unterschiedlichen Verarbeitungsgeschwindigkeiten.

Man spricht von einer **Prozessorfamilie**, wenn alle Prozessoren die gleiche Basisarchitektur haben, wobei häufig die neueren oder die komplexeren Prozessoren der Familie die Architekturspezifikation erweitern. In einem solchen Fall ist nur eine Abwärtskompatibilität mit den älteren bzw. einfacheren Prozessoren der Familie gegeben, d.h. der Objektcode der älteren läuft auf den neueren Prozessoren, jedoch nicht umgekehrt.

Die Programmierersicht (Programmiermodell) eines Prozessors lässt sich durch Beantworten der folgenden fünf Fragen einführen:

- Wie werden Daten repräsentiert?
- Wo werden die Daten gespeichert?
- Welche Operationen (Befehle) können auf den Daten ausgeführt werden?
- Wie werden die Befehle codiert?
- Wie wird auf die Operanden zugegriffen?

Die Antworten auf diese Fragen definieren die Prozessorarchitektur bzw. das Programmiermodell eines Prozessors. Wir wollen diese Eigenschaften im Einzelnen in den folgenden Abschnitten genauer betrachten.

2.3.2 Datenformate

Wie in den höheren Programmiersprachen können auch in maschinennahen Sprachen die Daten verschiedenen Datenformaten zugeordnet werden. Diese Datenformate legen fest, wie die Daten rechnerintern repräsentiert werden. Bei der Übertragung der Daten zwischen Speicher und prozessorinternen Registern kann die Größe der übertragenen Datenportion mit dem Maschinenbefehl festgelegt werden. Übliche Datenlängen sind Byte (8 Bits), Halbwort (16 Bits), Wort (32 Bits) und Doppelwort (64 Bits). In der Assemblerschreibweise werden diese Datengrößen oft durch die Buchstaben B, H, W und D abgekürzt, wie z.B. in dem Befehl MOVB, der die Übertragung eines Byte beschreibt. Bei Mikrocontrollern – das sind aus einem Mikroprozessor und verschiedenen Schnittstelleneinheiten bestehende vollständige Mikrorechner auf einem einzigen Chip – werden auch die Definitionen Wort (16 Bits), Doppelwort (32 Bits) und Quadwort (64 Bits) angewendet.

Ein 32-Bit-Datenwort reflektiert die Sicht eines 32-Bit-Prozessors bzw. ein 16-Bit-Datenwort reflektiert die Sicht eines 16-Bit-Prozessors oder -Mikrocontrollers. Wir sprechen von einem *n*-Bit-Prozessor, wenn die allgemeinen Register *n* Bit breit sind. Da diese Register häufig auch für die Speicheradressierung verwendet werden, ist dann meist auch die Breite der effektiven Adresse *n* Bit. Dies ist heute insbesondere bei den PC-Prozessoren der Intel Pentium-Familie der Fall. Workstation-Prozessoren, wie die Sun UltraSPARC-Prozessoren und die Itanium-Prozessoren der Firma Intel, sind 64-Bit-Prozessoren. Bei Mikrocontrollern sind oft auch 8-Bit- und 16-Bit-Prozessorkerne üblich.

Die Befehlssätze unterstützen jedoch auch Datenformate wie zum Beispiel Einzelbit-, Ganzzahl- (*integer*), Gleitkomma- und Multimediaformate, die mit den Datenformaten in Hochsprachen enger verwandt sind. Die Einzelbitdatenformate können dabei alle Datenlängen von 8 Bit bis hin zu 256 Bit aufweisen. Das Besondere dabei ist, dass einzelne Bits eines solchen Worts manipuliert werden können.

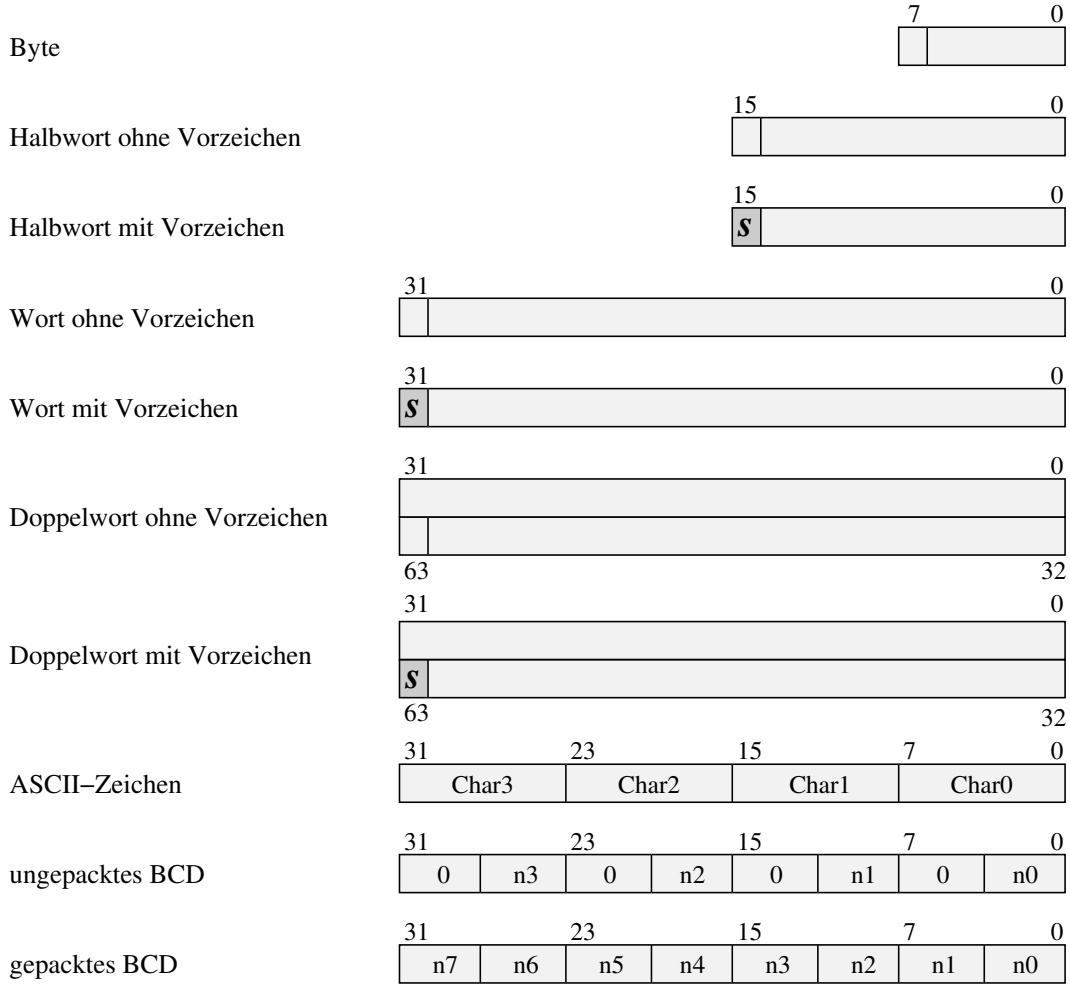


Abbildung 2.5: Ganzzahlendatenformate

Ganzzahldatenformate (Abb. 2.5) sind unterschiedlich lang und können mit Vorzeichen (*signed*) oder ohne Vorzeichen (*unsigned*) definiert sein. Gelegentlich findet man auch gepackte (*packed*) und ungepackte (*unpacked*) BCD-Zahlen (*Binary Coded Decimal*) und ASCII-Zeichen (*American Standard Code for Information Interchange*). Eine BCD-Zahl codiert die Ziffern 0 bis 9 als Dualzahlen in vier Bits. Das gepackte BCD-Format codiert zwei BCD-Zahlen pro Byte, also acht BCD-Zahlen pro 32-Bit-Wort. Das ungepackte BCD-Format codiert eine BCD-Zahl an den vier niederwertigen Bitpositionen eines Bytes, also nur vier BCD-Zahlen pro 32-Bit-Wort. Der ASCII-Code belegt ein Byte pro Zeichen, so dass vier ASCII-codierte Zeichen in einem 32-Bit-Wort untergebracht werden.

Die Gleitkommadatenformate (Abb. 2.6) werden durch den IEEE 754-1985-Standard definiert und unterscheiden Gleitkommazahlen mit einfacher (32 Bit) oder doppelter (64 Bit) Genauigkeit. Das

Format mit erweiterter Genauigkeit umfasst 80 Bit und kann herstellergebunden variieren. Beispielsweise verwenden die Intel Pentium-Prozessoren intern ein solches erweitertes Format mit 80 Bit breiten Gleitkommazahlen.

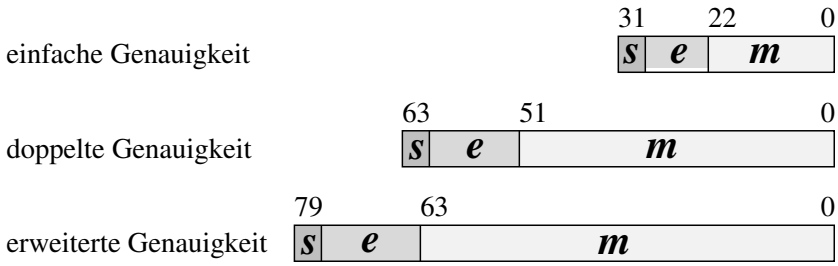


Abbildung 2.6: Gleitkomma-Datenformate

Eine Gleitkommazahl f wird nach dem IEEE-Standard wie folgt dargestellt:

$$F = (-1)^s \cdot 1.m \cdot 2^{e-b}$$

Dabei steht s für das Vorzeichenbit (0 für positiv, 1 für negativ), e für den verschobenen (*biased*) Exponenten, b für die Verschiebung (*bias*) und m für die Mantisse oder den Signifikanten. Die führende Eins in der obigen Gleichung ist implizit vorhanden und benötigt kein Bit im Mantissenfeld (Abb. 2.6). Für die Mantisse m gilt:

$$m = .m_1 m_2 \ldots m_p$$

mit $p = 23$ für die einfache, $p = 52$ für die doppelte und $p = 63$ für die erweiterte Genauigkeit (in diesem Fall wird die führende Eins der Mantisse mit dargestellt). Die Verschiebung b ist definiert als:

$$b = 2^{ne-1} - 1$$

wobei ne die Anzahl der Exponentenbits (8 bei einfacher, 11 bei doppelter und 15 bei erweiterter Genauigkeit) bedeutet. Der Wert des Exponenten E ergibt sich aus der Gleichung:

$$E = e - b = e - (2^{ne-1} - 1)$$

Multimediadatenformate definieren 64 oder 128 Bit breite Wörter. Man unterscheidet zwei Arten von Multimediadatenformaten (Abb. 2.7): Die bitfeldorientierten Formate unterstützen Operationen auf Pixeldarstellungen, wie sie für die Videocodierung oder -decodierung benötigt werden. Die graphikorientierten Formate unterstützen komplexe graphische Datenverarbeitungsoperationen. Die Multimediadatenformate für die bitfeldorientierten Formate sind in 8 oder 16 Bit breite Teilfelder zur Repräsentation jeweils eines Pixels aufgeteilt. Die graphikorientierten Formate beinhalten zwei bis vier einfach genaue Gleitkommazahlen.

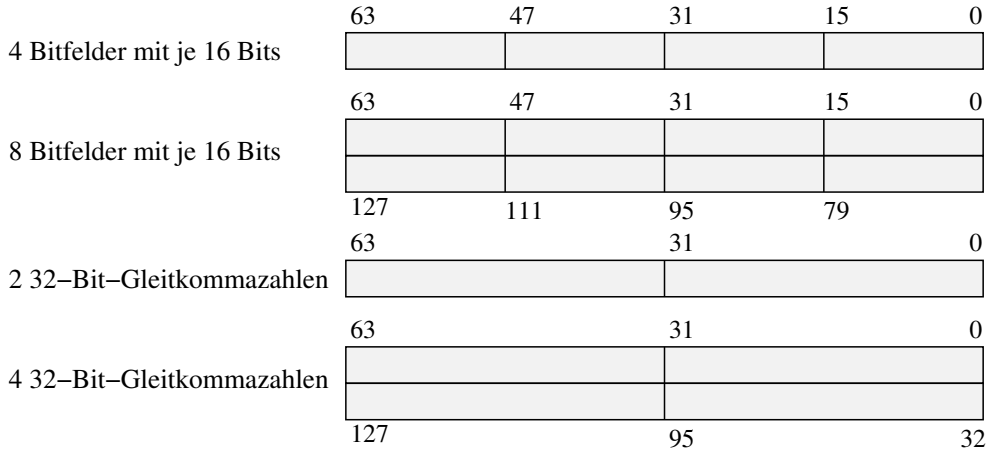


Abbildung 2.7: Beispiele bitfeld- und graphikorientierter Multimediadatenformate

2.3.3 Adressraumorganisation

Die Adressraumorganisation bestimmt, wo die Daten gespeichert werden. Jeder Prozessor enthält eine kleine Anzahl von **Registern**. Register sind sehr schnelle interne Speicherplätzen, auf die in einem Taktzyklus zugegriffen werden kann. Bei den heute üblichen Pipeline-Prozessoren wird in der Operandenholphase der Befehls-Pipeline auf die Registeroperanden zugegriffen und in der Resultatspeicherphase in Ergebnis-Register zurück gespeichert. Diese Register können **allgemeine Register** (auch Universalregister oder Allzweckregister genannt), **Multimediaregister**, **Gleitkommaregister** oder **Spezialregister** (Befehlszähler, Statusregister etc.) sein. Heutige Mikroprozessoren verwenden meist 32 allgemeine Register R0,R1,...,R31 von je 32 oder 64 Bit und zusätzlich 32 Gleitkommaregister F0,...,F31 von je 64 oder 80 Bit. Oft ist außerdem das Universalregister R0 fest mit dem Wert 0 verdrahtet (dies wird im Folgenden auch so angenommen). Die Multimediaregister stehen für sich oder sind mit den Gleitkommaregistern identisch. All diese für den Programmierer sichtbaren und frei verwendbaren Register werden als **Architekturregister** bezeichnet, da sie im Programmiermodell, der „Prozessorarchitektur“ sichtbar sind. Im Gegensatz dazu sind die auf heutigen Mikroprozessoren oft vorhandenen physikalischen Register oder Umbenennungspufferregister (vgl. Abschnitt 2.6.1) nicht in der Architektur sichtbar.

Um Daten zu speichern werden mehrere Adressräume unterschieden. Diese sind neben den Registern und Spezialregistern insbesondere Speicheradressräume für

- den Laufzeitstapel (*run-time stack)*, insbesondere zur Ablage von Rücksprungadressen aus Unterprogrammen und Unterbrechungsroutinen,
- den *Heap*, einen Speicherbereich für dynamisch zur Programmlaufzeit erzeugte Datenobjekte,
- die Ein-/Ausgabe- und Steuerdaten.

Abgesehen von den Registern werden alle anderen Adressräume meist auf einen einzigen, durchgehend adressierten Adressraum abgebildet. Dieser kann **Byte-adressierbar** sein, d.h. jedes Byte in einem Speicherwort kann einzeln adressiert werden. Oft sind heutige Prozessoren jedoch **wortadressierbar**, so dass nur 16-, 32- oder 64-Bit-Wörter direkt adressiert werden können. In der Regel muss deshalb der Zugriff auf Speicherwörter **ausgerichtet** (*aligned*) sein. Ein Zugriff auf ein Speicherwort mit einer Länge von n Bytes ab der Speicheradresse A heißt ausgerichtet, wenn $A \; modulo \; n = 0$ gilt, d.h. wenn der Rest der ganzzahligen Division von A durch n den Wert 0 ergibt.

Ein 64-Bit-Wort umfasst 8 Bytes und benötigt auf einem Byte-adressierbaren Prozessor 8 Speicheradressen. Für die Speicherung im Hauptspeicher unterscheidet man zwei Arten der Byteanordnung innerhalb eines Wortes (Abb. 2.8):

- Das **Big-Endian-Format** („*most significant byte first*“) speichert von links nach rechts, d.h. die Adresse des Speicherwortes ist die Adresse des höchstwertigen Bytes des Speicherwortes.
- Das **Little-Endian-Format** („*least significant byte first*“) speichert von rechts nach links, d.h. die Adresse eines Speicherwortes ist die Adresse des niedrigstwertigen Bytes des Speicherwortes.

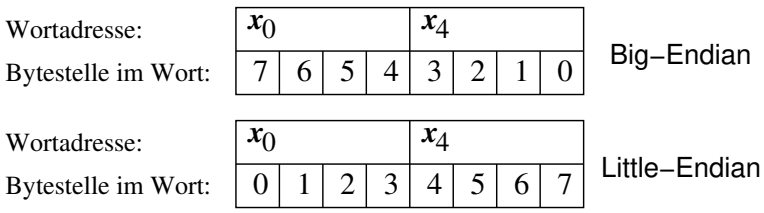


Abbildung 2.8: Big-Endian- und Little-Endian-Format

Für die Assemblerprogrammierung ist diese Unterscheidung häufig irrelevant, da beim Laden eines Operanden in ein Register die Maschine den zu ladenden Wert so anordnet, wie man es erwartet, nämlich die höchstwertigen Stellen links (Big-Endian-Format). Dies geschieht auch für das Little-Endian-Format, das bei einigen Prozessorarchitekturen, insbesondere den Intel-Prozessoren, aus Kompatibilitätsgründen mit älteren Prozessoren weiter angewandt wird. Beachten muss man das Byteanordnungsformat eines Prozessors insbesondere beim direkten Zugriff auf einen Speicherplatz als Byte oder Wort oder bei der Arbeit mit einem *Debugger*, also einem Software-Werkzeug[1] zum Finden von Fehlern in Programmen. Die Bytereihenfolge wird ein Problem, wenn Daten zwischen zwei Rechnern verschiedener Architektur ausgetauscht werden. Heute setzt sich insbesondere durch die Bedeutung von Rechnernetzen das Big-Endian-Format durch, das häufig auch als Netzwerk-Format bezeichnet wird.

2.3.4 Befehlssatz

Der **Befehlssatz** (*instruction set*) definiert, welche Operationen auf den Daten ausgeführt werden können. Er legt daher die Grundoperationen eines Prozessors fest. Man kann die folgenden **Befehlsarten** unterscheiden:

Datenbewegungsbefehle (*data movement*) übertragen Daten von einer Speicherstelle zu einer anderen. Falls es einen separaten Ein-/Ausgabeadressraum gibt, so gehören hierzu auch die Ein-/Ausgabebefehle. Auch die Stapelspeicherbefehle PUSH und POP fallen, sofern vorhanden, in diese Kategorie.

Arithmetisch-logische Befehle (*Integer arithmetic and logical*) können Ein-, Zwei- oder Dreioperandenbefehle sein. Prozessoren nutzen meist verschiedene Befehle für verschiedene Datenformate ihrer Operanden. Meist werden durch den Befehlsopcode arithmetische Befehle mit oder ohne Vorzeichen unterschieden. Beispiele arithmetischer Operationen sind Addieren ohne/mit Übertrag, Subtrahieren ohne/mit Übertrag, Inkrementieren und Dekrementieren, Multiplizieren ohne/mit Vorzeichen, Dividieren ohne/mit Vorzeichen und Komplementieren im Zweierkomplement. Beispiele logischer Operationen sind die bitweise Negation-, UND-, ODER- und ANTIVALENZ-Operationen.

Schiebe- und Rotationsbefehle (*shift, rotate*) schieben die Bits eines Wortes um eine Anzahl von Stellen entweder nach links oder nach rechts bzw. rotieren die Bits nach links oder rechts (Abb. 2.9).

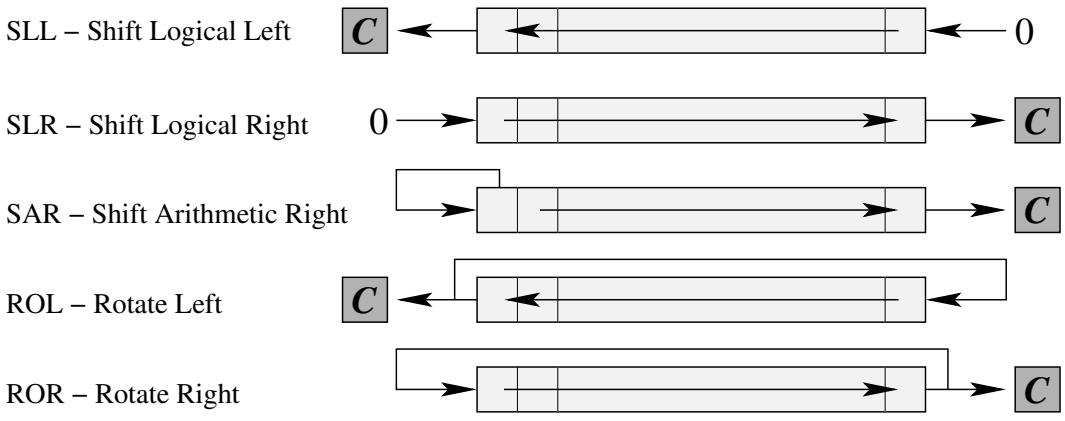


Abbildung 2.9: Einige Schiebe- und Rotationsbefehle

Beim Schieben gehen die herausfallenden Bits verloren, beim Rotieren werden diese auf der anderen Seite wieder eingefügt. Beispiele von Schiebe- und Rotationsoperationen sind das Linksschieben, Rechtsschieben, Linksrotieren ohne Übertragsbit, Linksrotieren durchs Übertragsbit, Rechtsrotieren ohne Übertragsbit und das Rechtsrotieren durchs Übertragsbit. Dem arithmetischen Linksschieben entspricht die Multiplikation mit 2. Dem arithmetischen Rechtsschieben entspricht die ganzzahlige Division durch 2. Dies gilt jedoch nur für positive Zahlen. Bei negativen Zahlen in **Zweierkomple-**

[1] Beim Entwurf von Steuerungssystemen enthält der Debugger meist auch noch eine Hardware-Komponente.

mentdarstellung muss zur Vorzeichenerhaltung das höchstwertige Bit in sich selbst zurückgeführt werden. Daraus ergibt sich der Unterschied des logischen und arithmetischen Rechtsschiebens. Beim Rotieren wird ein Register als geschlossene Bitkette betrachtet. Ein sog. Übertragsbit (*carry flag*) im Prozessorstatusregister kann wahlweise mitbenutzt oder als zusätzliches Bit einbezogen werden.

Multimediabefehle (*multimedia instructions*) führen taktsynchron dieselbe Operation auf mehreren Teiloperanden innerhalb eines Operanden aus (Abb. 2.10). Man unterscheidet zwei Arten von Multimediabefehlen: bitfeldorientierte und graphikorientierte Multimediabefehle.

Bei den *bitfeldorientierten Multimediabefehlen* repräsentieren die Teiloperanden Pixel. Typische Operationen sind Vergleiche, logische Operationen, Schiebe- und Rotationsoperationen, Packen und Entpacken von Teiloperanden in oder aus Gesamtoperanden sowie arithmetische Operationen auf den Teiloperanden entsprechend einer Saturationsarithmetik: Bei einer solchen Arithmetik werden Zahlbereichsüberschreitungen auf die höchstwertige bzw. niederstwertige Zahl abgebildet. Dadurch wird z.B. verhindert, dass die Summe zweier positiver Zahlen negativ wird.

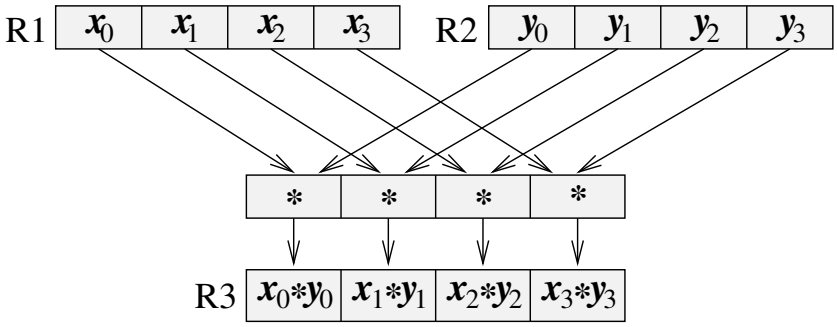


Abbildung 2.10: Grundprinzip einer Multimediaoperation

Bei den *graphikorientierten Multimediabefehlen* repräsentieren die Teiloperanden einfach genaue Gleitkommazahlen, also zwei 32-Bit-Gleitkommazahlen in einem 64-Bit-Wort bzw. vier 32-Bit-Gleitkommazahlen in einem 128-Bit-Wort. Die Multimediaoperationen führen dieselbe Gleitkommaoperation auf allen Teiloperanden aus.

Gleitkommabefehle (*floating-point instructions*) repräsentieren arithmetische Operationen und Vergleichsoperationen, aber auch zum Teil komplexe Operationen wie Quadratwurzelbildung oder transzendente Funktionen auf Gleitkommazahlen.

Programmsteuerbefehle (*control transfer instructions*) sind alle Befehle, die den Programmablauf direkt ändern, also die bedingten und unbedingten Sprungbefehle, Unterprogrammaufruf und -rückkehr sowie Unterbrechungsaufruf und -rückkehr.

Systemsteuerbefehle (*system control instructions*) erlauben es in manchen Befehlssätzen, direkten Einfluss auf Prozessor- oder Systemkomponenten wie z.B. den Daten-Cache-Speicher oder die Speicherverwaltungseinheit zu nehmen. Weiterhin gehören der HALT-Befehl zum Anhalten des Prozessors und Befehle zur Verwaltung der elektrischen Leistungsaufnahme zu dieser Befehlsgruppe, die üblicherweise nur vom Betriebssystem genutzt werden dürfen.

Synchronisationsbefehle ermöglichen es, Synchronisationsoperationen zur Prozess- und Unterbrechungsbehandlung durch das Betriebssystem zu implementieren. (Unter einem Prozess versteht man dabei ein in Ausführung befindliches oder ausführbereites Programm mit seinen Daten, also den Konstanten und Variablen mit ihren aktuellen Werten.) Wesentlich ist dabei, dass bestimmte, eigentlich sonst nur durch mehrere Befehle implementierbare Synchronisationsoperationen ohne Unterbrechung (auch als „atomar“ bezeichnet) ablaufen müssen. Ein Beispiel ist der *swap*-Befehl, der als atomare Operation einen Speicherwert mit einem Registerwert vertauscht. Noch komplexer ist der TAS-Befehl (*test and set*), der als atomare Operation einen Speicherwert liest, diesen auf Null testet, ggf. ein Bedingungsbit im Prozessorstatuswort setzt und einen bestimmten Wert zurückspeichert. Die Ausführung als atomare Operation verhindert, dass z.B. ein weiterer Prozessor zwischenzeitlich dieselbe Speicherzelle liest und dort noch den alten, unveränderten Wert vorfindet.

2.3.5 Befehlsformate

Das **Befehlsformat** (*instruction format*) definiert, wie die Befehle codiert sind. Eine Befehlscodierung beginnt mit dem Opcode (Operation Code), der den Befehl selbst festlegt. In Abhängigkeit vom Opcode werden weitere Felder im Befehlsformat benötigt. Diese sind für die arithmetisch-logischen Befehle die Adressfelder, um Quell- und Zieloperanden zu spezifizieren, und für die Lade-/Speicherbefehle die Quell- und Zieladressangaben. Bei Programmsteuerbefehlen wird der als nächstes auszuführende Befehl adressiert.

Je nach der Art, wie die arithmetisch-logischen Befehle ihre Operanden adressieren, unterscheidet man vier Klassen von Befehlssätzen:

- Das **Dreiadressformat** *(3-address instruction format)* besteht aus dem Opcode, zwei Quell- (im Folgenden Src1, Scr2) und einem Zieloperandenbezeichner (Dest):

OPCODE	DEST	SRC1	SCR2

- Das **Zweiadressformat** *(2-address instruction format)* besteht aus dem Opcode, einem Quell- und einem Quell-/Zieloperandenbezeichner, d.h. ein Operandenbezeichner bezeichnet einen Quell- und gleichzeitig den Zieloperanden:

OPCODE	DEST/SRC1	SRC2

- Das **Einadressformat** *(1-address instruction format)* besteht aus dem Opcode und einem Quelloperandenbezeichner:

OPCODE	SRC

  Dabei wird ein im Prozessor ausgezeichnetes Register, das sogenannte Akkumulatorregister, implizit adressiert. Dieses Register enthält immer einen Quelloperanden und nimmt das Ergebnis auf.

- Das **Nulladressformat** *(0-address instruction format)* besteht nur aus dem Opcode:

OPCODE

  Voraussetzung für die Verwendung eines Nulladressformats ist eine sog. Stackarchitektur, die wir gleich noch beschreiben.

Die Adressformate hängen eng mit der folgenden Klassifizierung von Befehlssatzarchitekturen zusammen:

- Arithmetisch-logische Befehle sind meist **Dreiadressbefehle**, die zwei Operandenregister und ein Zielregister angeben. Falls nur die Lade- und Speicherbefehle Daten zwischen dem Hauptspeicher (bzw. Cache-Speicher) und den Registern transportieren, spricht man von einer **Lade-/Speicherarchitektur** (*load/store architecture*). Da arithmetische und logische Operationen nur mit Operanden aus Registern ausgeführt und die Ergebnisse auch nur in Register gespeichert werden können, spricht man auch von einer **Register-Register-Architektur** .
- Analog kann man von einer **Register-Speicher-Architektur** sprechen, wenn in arithmetisch-logischen Befehlen mindestens einer der Operandenbezeichner ein Register bzw. einen Speicherplatz im Hauptspeicher adressiert. Falls gar keine Register existieren, so muss jeder Operandenbezeichner eine Speicheradresse sein und man kann von einer **Speicher-Speicher-Architektur** sprechen. (Der einzige uns bekannte Prozessor diesen Typs war der TI 9900 der Firma Texas Instruments.) Die ersten Mikroprozessoren besaßen ein sog. **Akkumulatorregister**, das bei arithmetisch-logischen Befehlen immer implizit eine Quelle und das Ziel darstellte, so dass **Einadressbefehle** genügten. Solche Akkumulatorarchitekturen sind gelegentlich noch bei einfachen Mikrocontrollern und Digitalen Signalprozessoren (DSPs) zu finden.

- Man kann sogar mit **Nulladressbefehlen** auskommen: Dies geschieht bei den sog. **Stackarchitekturen** oder **Kellerarchitekturen**, welche ihre Operandenregister als Stapel (*stack*) verwalten. Eine zweistellige Operation verknüpft die beiden obersten Stackeinträge miteinander, löscht beide Inhalte vom Registerstapel und speichert das Resultat auf dem obersten Register des Stapels wieder ab.

Tabelle 2.1 zeigt, wie ein einfaches Zweizeilenprogramm, dass aus einer Additions- und einer Subtraktions-Operation besteht, in Pseudo-Assemblerbefehlen für die vier Befehlssatz-Architekturen dargestellt werden kann. Die Syntax der Assemblerbefehle schreibt vor, dass nach dem Opcode erst der Zieloperandenbezeichner und dann der oder die Quelloperandenbezeichner stehen. Manche andere Assemblernotationen verfahren genau umgekehrt und verlangen, dass der oder die Quelloperandenbezeichner vor dem Zieloperandenbezeichner stehen müssen.

Register-Register	**Register-Speicher**	**Akkumulator**	**Stapel**
LOAD REG1,A	LOAD REG1,A	LOAD A	PUSH B
LOAD REG2,B	ADD REG1,B	ADD B	PUSH A
ADD REG3,REG1,REG2	STORE C,REG1	STORE C	ADD
STORE C,REG3			POP C
LOAD REG1,C	SUB REG1,B	SUB B	PUSH B
LOAD REG2,B	STORE D,REG1	STORE D	PUSH C
SUB REG3,REG1,REG2			SUB
STORE D,REG3			POP D

Tabelle 2.1: Programm aus den zwei Operationen C=A+B; D=C-B in den vier Befehlsformatsarten (REGI steht für nicht genauer festgelegtes Register)

In der Regel kann bei heutigen Mikroprozessoren jeder Registerbefehl auf jedes beliebige Register gleichermaßen zugreifen. Bei älteren Prozessoren war dies jedoch keineswegs der Fall. Noch beim Intel i8086 gab es beliebig viele Anomalien beim Registerzugriff. Beispiele für Stackarchitekturen sind die von der Java Virtual Machine hergeleiteten Java-Prozessoren, die Verwaltung der Gleitkommaregister der Intel 8087- bis 80387-Gleitkomma-Coprozessoren sowie der 4-Bit-Mikroprozessor ATAM862 der Firma Atmel.

Die Befehlscodierung kann eine feste oder eine variable Befehlslänge festlegen. Um die Codierung zu vereinfachen, nutzen RISC-Befehlssätze meist ein Dreiadressformat mit einer festen Befehlslänge von 32 Bit. CISC-Befehlssätze dagegen nutzen Register-Speicher-Befehle und benötigen dafür meist variable Befehlslängen. Um den Speicherbedarf zu minimieren hat man früher bei CISC-Befehlssätzen mit variablen Opcodelängen gearbeitet. Hierzu wurden häufig benutzten Befehlen kurze Opcodes zugeordnet. Variable Befehlslängen finden sich auch in Stackmaschinen wie beispielsweise bei den Java-Prozessoren.

2.3.6 Adressierungsarten

Die **Adressierungsarten** definieren, wie auf die Daten zugegriffen wird. Sie bestimmen die verschiedenen Möglichkeiten, wie eine Operanden- oder eine Sprungzieladresse in dem Prozessor berechnet werden kann. Eine Adressierungsart kann eine im Befehlswort stehende Konstante, ein Register oder einen Speicherplatz im Hauptspeicher spezifizieren.

Wenn ein Speicherplatz im Hauptspeicher bezeichnet wird, so heißt die durch die Adressierungsart spezifizierte Speicheradresse die **effektive Adresse**. Eine effektive Adresse entsteht im Prozessor nach Ausführung der Adressberechnung. In modernen Prozessoren, die eine virtuelle Speicherverwaltung anwenden, wird die effektive Adresse als sog. logische Adresse weiteren Speicherverwaltungsoperationen in einer Speicherverwaltungseinheit (*Memory Management Unit – MMU*) unterworfen,

um letztendlich eine physikalische Adresse zu erzeugen, mit der dann auf den Hauptspeicher zugegriffen wird. Im Folgenden betrachten wir zunächst nur die Erzeugung einer effektiven Adresse aus den Angaben in einem Maschinenbefehl.

Neben den „expliziten“ Adressierungsarten kann die Operandenadressierung auch „implizit“ bereits in der Architektur oder durch den Opcode des Befehls festgelegt sein. In der oben erwähnten Stackarchitektur sind z.B. die beiden Quelloperanden und der Zieloperand einer arithmetisch-logischen Operation implizit als die beiden bzw. der oberste Stackeintrag festgelegt. Ähnlich ist bei einer Akkumulatorarchitektur das Akkumulatorregister bereits implizit als ein Quell- und als Zielregister der arithmetisch-logischen Operationen fest vorgegeben. Bei einer Register-Speicher-Architektur ist meist das eine Operandenregister fest vorgegeben, während der zweite Operand und das Ziel explizit adressiert werden müssen.

Durch den Opcode eines Befehls wird bei Spezialbefehlen (Programm- oder Systemsteuerbefehle) häufig ein besonderes Register als Quelle und/oder Ziel adressiert. Weiterhin wird in vielen Befehlssätzen im Opcode festgelegt, ob ein Bit des Prozessorstatusregisters mit verwendet wird.

Bei den im Folgenden aufgeführten expliziten Adressierungsarten können drei Klassen unterschieden werden:

1. die Klasse der Register- und unmittelbaren Adressierung,
2. die Klasse der einstufigen und
3. die Klasse der zweistufigen Speicheradressierungen.

Bei der Registeradressierung steht der Operand in einem Register und bei der unmittelbaren Adressierung steht der Operand, das zu verarbeitende Datum, direkt im Befehlswort. In beiden Fällen sind weder eine Adressberechnung noch ein Speicherzugriff nötig.

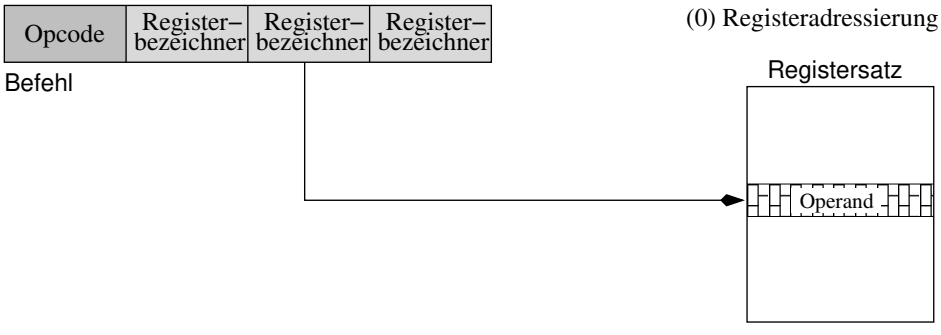
Bei der einstufigen Speicheradressierung steht der Operand im Speicher, und für die effektive Adresse ist nur *eine* Adressberechnung notwendig. Diese Adressberechnung kann einen oder mehrere Registerinhalte sowie einen im Befehl stehenden Verschiebewert oder einen Skalierungsfaktor, jedoch keinen weiteren Speicherinhalt betreffen.

Wenig gebräuchlich sind die zweistufigen Speicheradressierungen, bei denen mit einem Teilergebnis der Adressberechnung wiederum auf den Speicher zugegriffen wird, um einen weiteren Datenwert für die Adressberechnung zu holen. Es ist somit ein doppelter Speicherzugriff notwendig, bevor der Operand zur Verfügung steht.

Im Folgenden zeigen wir eine Auswahl von **Datenadressierungsarten**, die in heutigen Mikroprozessoren und Mikrocontrollern Verwendung finden. Abgesehen von der Register- und der unmittelbaren Adressierung sind dies allesamt einstufige Speicheradressierungsarten.

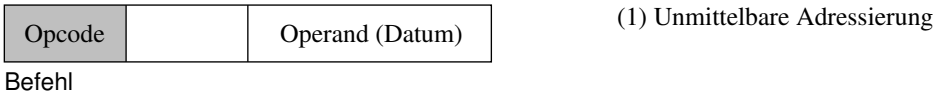
(0) **Registeradressierung**

Bei der Registeradressierung (*register*) steht der Operand direkt in einem Register.



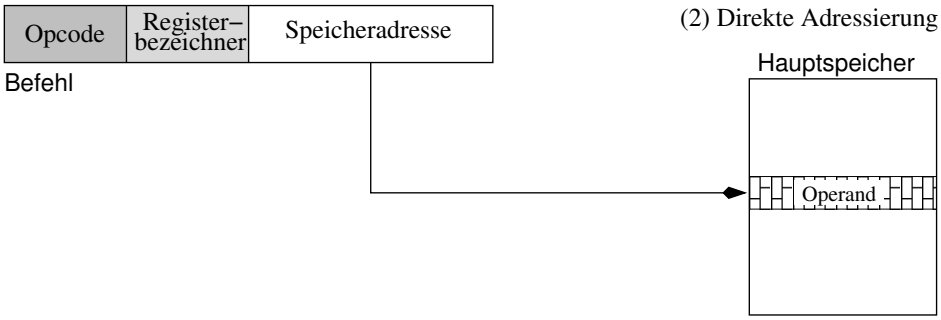
(1) **Unmittelbare Adressierung**

Bei der unmittelbaren Adressierung (*immediate*) steht der Operand als Konstante direkt im Befehlswort.



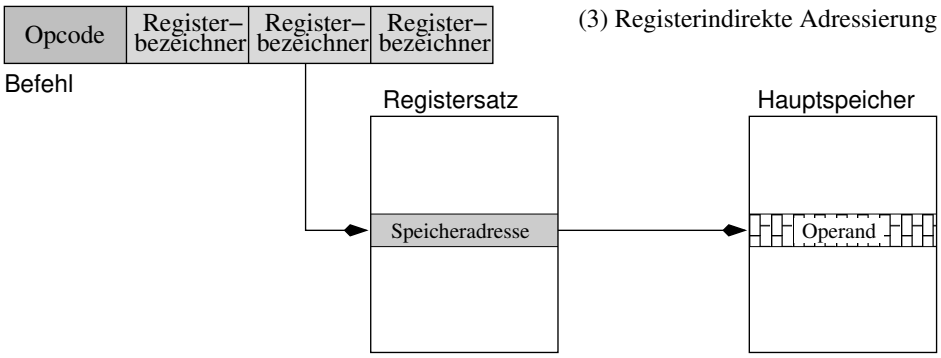
(2) **Direkte** oder **Absolute Adressierung**

Bei der direkten oder absoluten Adressierung (*direct, absolute*) steht die Adresse eines Speicheroperanden im Befehlswort.



(3) **Registerindirekte Adressierung**

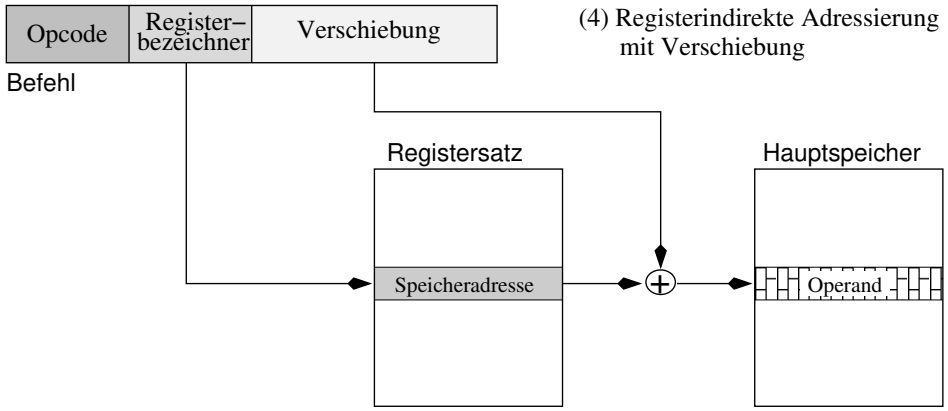
Bei der registerindirekten Adressierung (*register indirect* oder *register deferred*) steht die Operandenadresse in einem Register. Das Register dient als Zeiger auf eine Speicheradresse.



Als Spezialfälle der registerindirekten Adressierung können die **registerindirekten Adressierungen mit Autoinkrement/Autodekrement** (*autoincrement/autodecrement*) betrachtet werden. Diese arbeiten wie die registerindirekte Adressierung, aber inkrementieren bzw. dekrementieren den Registerinhalt. Bei dieser registerindirekten Adressierung wird also die „Speicheradresse“ vor oder nach dem Benutzen dieser Adresse um die Länge des adressierten Operanden inkrementiert oder dekrementiert. Dementsprechend unterscheidet man die registerindirekte Adressierung mit **Präinkrement**, mit **Postinkrement**, mit **Prädekrement** und mit **Postdekrement** (üblich sind Postinkrement und Prädekrement). Diese Adressierungsarten sind für den Zugriff zu Feldern (*arrays*) in Schleifen nützlich. Der Registerinhalt zeigt auf den Anfang oder das letzte Element eines Feldes und jeder Zugriff erhöht oder erniedrigt den Registerinhalt um die Länge eines Feldelements.

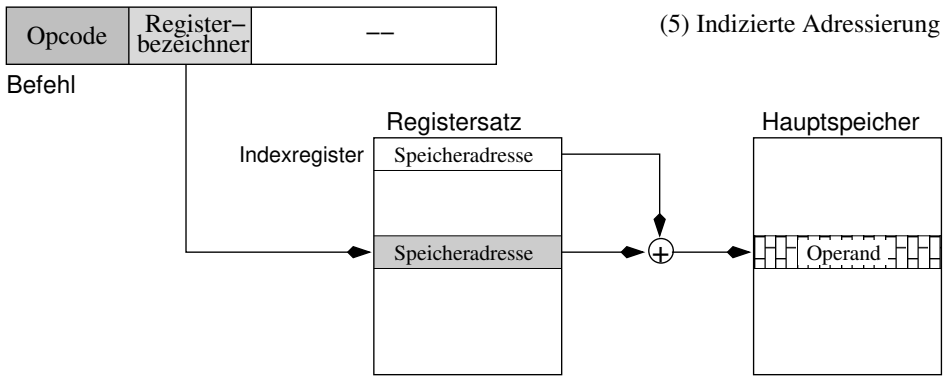
(4) **Registerindirekte Adressierung mit Verschiebung**

Die registerindirekte Adressierung mit Verschiebung (*displacement*, *register indirect with displacement* oder *based*) errechnet die effektive Adresse eines Operanden als Summe eines Registerwerts und des Verschiebewerts (*displacement*), d.h. eines konstanten Werts, der im Befehl steht.



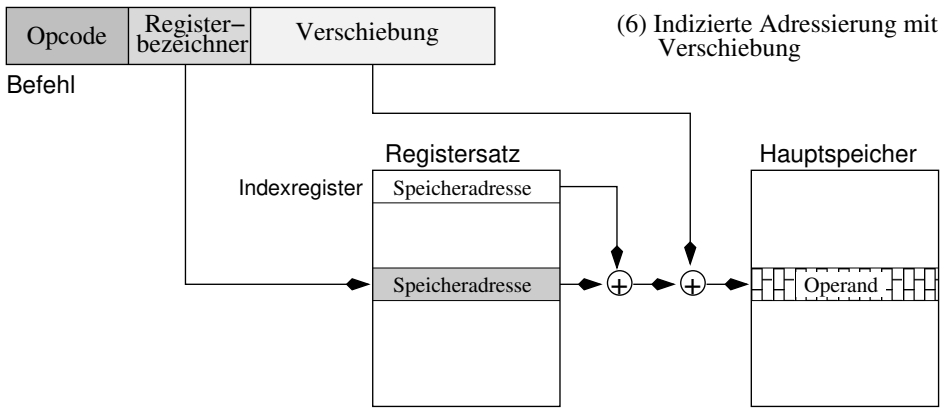
(5) **Indizierte Adressierung**

Die indizierte Adressierung (*indirect indexed*) errechnet die effektive Adresse als Summe eines Registerinhalts und eines weiteren Registers, das bei manchen Prozessoren als spezielles Indexregister vorliegt. Damit können Datenstrukturen beliebiger Größe und mit beliebigem Abstand durchlaufen werden. Angewendet wird die indizierte Adressierung auch beim Zugriff auf Tabellen, wobei der Index erst zur Laufzeit ermittelt wird.



(6) **Indizierte Adressierung mit Verschiebung**

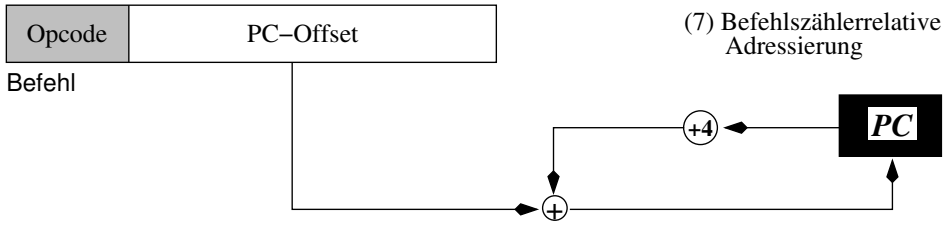
Die indizierte Adressierung mit Verschiebung (*indirect indexed with displacement*) ähnelt der indizierten Adressierung, allerdings wird zur Summe der beiden Registerwerte noch ein im Befehl stehender Verschiebewert (*displacement*) hinzuaddiert.



Zur Änderung des Befehlszählregisters (*Program Counter* – PC) durch Programmsteuerbefehle (bedingte oder unbedingte Sprünge sowie Unterprogrammaufruf und -rücksprung) sind nur zwei **Befehlsadressierungsarten** üblich:

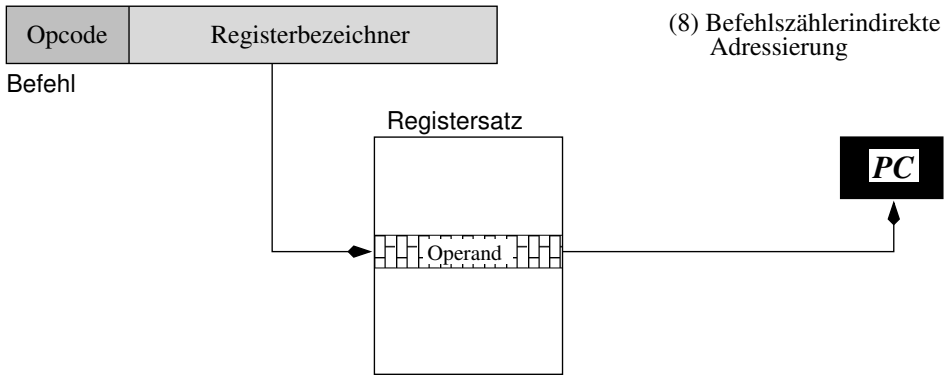
(7) **Befehlszählerrelative Adressierung**

Der befehlszählerrelative Modus (*PC-relative*) addiert einen Verschiebewert (*displacement*, *PC-offset*) zum Inhalt des Befehlszählerregisters bzw. häufig auch zum Inhalt des inkrementierten Befehlszählers PC+4, denn dieser wird bei Architekturen mit 32-Bit-Befehlsformat meist automatisch um vier erhöht. Die Sprungzieladressen sind häufig in der Nähe des augenblicklichen Befehlszählerwertes, so dass nur wenige Bits für den Verschiebewert im Befehl benötigt werden.



(8) **Befehlszählerindirekte Adressierung**

Der befehlszählerindirekte Modus (*PC-indirect*) lädt den neuen Befehlszähler aus einem allgemeinen Register. Das Register dient als Zeiger auf eine Speicheradresse, bei der im Programmablauf fortgefahren wird.



Die Tabelle 2.2 fasst die verschiedenen Adressierungsarten nochmals zusammen. In der Tabelle bezeichnet `Mem`[R2] den Inhalt des Speicherplatzes, dessen Adresse durch den Inhalt des Registers R2 gegeben ist; *const*, *displ* können Dezimal-, Hexadezimal-, Oktal- oder Binärzahlen sein; *step* bezeichnet die Feldelementbreite und `inst_step` bezeichnet die Befehlsschrittweite in Abhängigkeit von der Befehlswortbreite, z.B. vier bei Vierbyte-Befehlswörtern.

Darüber hinaus gibt es eine ganze Anzahl weiterer Adressierungsarten. Wir wollen aber damit die Betrachtung des Programmiermodells bzw. der Prozessorarchitektur abschließen und uns im nächsten Abschnitt der Mikroarchitektur zuwenden.

Aufgabe 2.2 *Registerindirekte Adressierung*
Welche einfacheren Adressierungsarten lassen sich aus der registerindirekten Adressierung mit Verschiebung zusammensetzen? ◇

Adressierungsart	**Beispielbefehl**	**Bedeutung**
Register	LOAD R1,R2	R1 ← R2
Unmittelbar	LOAD R1,`const`	R1 ← `const`
Direkt, Absolut	LOAD R1,(`const`)	R1 ← `Mem[const]`
Registerindirekt	LOAD R1,(R2)	R1 ← `Mem`[R2]
Postinkrement	LOAD R1,(R2)+	R1 ← `Mem[R2]`, R2 ← R2+`step`
Prädekrement	LOAD R1,–(R2)	R2 ← R2-`step`, R1 ← `Mem`[R2]
Registerindirekt mit Verschiebung	LOAD R1,`displ`(R2)	R1 ← `Mem`[`displ`+R2]
Indiziert	LOAD R1,(R2,R3)	R1 ← `Mem`[R2+R3]
Indiziert mit Verschiebung	LOAD R1,`displ`(R2,R3)	R1 ← `Mem`[`displ`+R2+R3]
Befehlszählerrelativ	BRANCH `displ`	PC←PC+`inst_step`+`displ` (falls Sprung genommen) PC ← PC+`inst_step` (sonst)
Befehlszählerindirekt	BRANCH R2	PC ← R2 (falls Sprung genommen) PC ← PC+`inst_step` (sonst)

Tabelle 2.2: Tabelle der Adressierungsarten

2.4 Die Mikroarchitektur

In Abschnitt 2.2.1 haben wir die Struktur des von-Neumann-Rechners kennen gelernt. Der Prozessor des von-Neumann-Rechners besteht aus einem Steuerwerk und einem Rechenwerk. Das **Steuerwerk** übernimmt die Steuerung aller Abläufe. Dazu interpretiert es die Maschinenbefehle und setzt sie unter Berücksichtigung von Statusinformation in Steuersignale für andere Komponenten um. Ein Befehlszähler (*Program Counter, PC*) innerhalb des Steuerwerks enthält die Speicheradresse des als nächstes auszuführenden Maschinenbefehls, sofern der vorangegangene Befehl kein Sprungbefehl ist. Das **Rechenwerk** führt die Befehle bzw. Operationen aus, bei denen es sich nicht um Ein-/Ausgabe-, Sprung- oder Lade-/Speicherbefehle handelt.

Die Maschinenbefehle bestehen in der Regel aus zwei Teilen, einem Operationscode (Opcode) und einem oder mehreren Operanden. Bei der Ausführung eines Maschinenbefehls durch den von-Neumann-Rechner wird der Operationscode auf den Operanden angewendet. Je nach Operationscode werden die Operanden dabei entweder direkt als Daten oder als Speicheradressen von Daten interpretiert. Da beim von-Neumann-Rechner Programmcode und Daten im selben Speicher stehen, kann eine Speicherzelle einen Datenwert oder einen Maschinenbefehl enthalten. Wie der Inhalt einer Speicherzelle interpretiert wird, hängt also allein vom Operationscode des Maschinenbefehls ab.

Aufgrund des Operationsprinzips des von-Neumann-Rechners werden die Maschinenbefehle in sequenzieller Reihenfolge ausgeführt. Dabei kann der nächste Befehl erst ausgeführt werden sobald die Ausführung des vorhergehenden Befehls abgeschlossen ist. Der von-Neumann-Rechner arbeitet in zwei Phasen:

- In der Befehlsbereitstellungs- und Decodierphase wird, adressiert durch den Befehlszähler, der Inhalt einer Speicherzelle in den Prozessor geholt und als Befehl interpretiert.
- In der Ausführungsphase werden die Inhalte von einer oder mehreren Speicherzellen bereitgestellt und entsprechend der Vorschrift durch den Opcode verarbeitet.

Heutige superskalare Prozessoren sind von ihrer Architektur her durch das von-Neumann-Prinzip geprägt, auch wenn in ihren Mikroarchitekturen Konzepte angewendet werden, die weit über das des von-Neumann-Prinzips hinausgehen. Größtes Hemmnis ist das Operationsprinzip, das eine rein sequentielle Ausführung vorschreibt. Dies erzwingt nach außen die sequentielle Semantik beizubehalten, obwohl heutige Superskalarprozessoren intern die Befehle hoch parallel und vielfach überlappt ausführen.

Zu diesem Zweck verwenden heutige Prozessoren in der Implementierung sowohl die Mikroprogrammierung als auch das Pipelining. Eine klare Trennung in CISC- und RISC-Prozessoren ist kaum noch möglich. CISC-Prozessoren arbeiten mit Mikroprogrammierung und lassen dabei die Mikrobefehle auf sehr schnellen Pipelines ausführen. Dabei werden komplexe Befehle durch die Mikroprogrammierung in kleine einheitliche Mikrobefehle aufgeteilt, die dann wieder sehr gut auf Pipelines ausgeführt werden können. Pipelining wird damit zum zentralen Organsisationsprinzip heutiger Prozessoren und wird daher auch Gegenstand der nächsten Abschnitte sein.

2.4.1 Überlappende Befehlsbearbeitung beim von-Neumann-Rechner

Ein von-Neumann-Rechner benötigt zum Ausführen eines Befehls die zwei grundlegenden Phasen Befehlsbereitstellung und Ausführung. In der einfachsten Implementierung ist jede der beiden Phasen abwechselnd aktiv, da die beiden Phasen sequentiell zueinander ausgeführt werden. Jede der beiden Phasen kann je nach Implementierung und Komplexität des Befehls wiederum eine oder mehrere Takte in Anspruch nehmen. Für die Durchführung eines Programms mit n Befehlen von je k Takten werden $n \cdot k$ Takte benötigt.

Weicht man diese reine Sequentialität auf, dann könnten diese beiden Phasen auch wie in Abb. 2.3 dargestellt überlappend zueinander ausgeführt werden. Während der eine Befehl sich in seiner Aus-

führungsphase befindet, wird bereits der nach Ausführungsreihenfolge nächste Befehl aus dem Speicher geholt und decodiert. Bei dieser Art der Befehlsverarbeitung sind aber nun zwei Prozesse (repräsentiert durch die beiden Phasen) gleichzeitig aktiv und benötigen möglicherweise konkurrierend Ressourcen. Die Organisation dieser überlappenden Befehlsbearbeitung ist dadurch erheblich komplizierter, denn dazu müssen Ressourcenkonflikte bedacht und gelöst werden.

Ein Ressourcenkonflikt tritt ein, wenn beide Phasen gleichzeitig dieselbe Ressource benötigen. Zum Beispiel muss in der Befehlsbereitstellungsphase über die entsprechende Verbindungseinrichtung (Daten- und Adressbus) auf den Speicher zugegriffen werden, um den Befehl zu holen. Gleichzeitig kann jedoch auch in der Ausführungsphase ein Speicherzugriff zum Holen eines Operanden notwendig sein. Falls Daten und Befehle im gleichen Speicher stehen (von-Neumann-Architektur im Gegensatz zur Harvard-Architektur) und zu einem Zeitpunkt nur ein Speicherzugriff erfolgen kann, so muss dieser Zugriffskonflikt hardwaremäßig erkannt und gelöst werden.

Ein weiteres Problem entsteht bei Programmsteuerbefehlen, die den Programmfluss ändern. In der Ausführungsphase eines Sprungbefehls, in der die Sprungzieladresse berechnet werden muss, wird bereits der im Speicher an der nachfolgenden Adresse stehende Befehl geholt. Dies ist im Falle eines unbedingten Sprungs jedoch nicht der richtige Befehl. Dieser Fall muss von der Hardware erkannt werden und entweder das Holen des nachfolgenden Befehls zurückgestellt oder der bereits geholte Befehl wieder gelöscht werden.

Idealerweise sollten beide Phasen etwa gleich viele Takte benötigen und keinerlei Konflikte hervorrufen. Dann kann unter Vernachlässigung der Start- und der Endphase der Verarbeitung die Verarbeitungsgeschwindigkeit gegenüber der Implementierung ohne Überlappung im Idealfall verdoppelt werden.

2.4.2 Pipeline-Prinzip

Die konsequente Fortführung der im letzten Abschnitt beschriebenen überlappenden Verarbeitung bildet bei heutigen Mikroprozessoren das **Pipelining** – die „Fließband-Bearbeitung‘, welche die Ausführungsgeschwindigkeit von Befehlen in ähnlicher Weise beschleunigt, wie z.B. die Herstellung von Produkten des täglichen Lebens.

Unter dem Begriff **Pipelining** versteht man die Zerlegung eines Verarbeitungsauftrages (im Folgenden eine Maschinenoperation) in mehrere Teilverarbeitungsschritte, die dann von hintereinander geschalteten Verarbeitungseinheiten taktsynchron bearbeitet werden, wobei jede Verarbeitungseinheit genau einen speziellen Teilverarbeitungsschritt ausführt. Die Gesamtheit dieser Verarbeitungseinheiten nennt man eine **Pipeline**. Pipelining ist eine Implementierungstechnik (Mikroarchitekturtechnik), bei der mehrere Befehle überlappt ausgeführt werden. Jede Stufe der Pipeline heißt **Pipeline-Stufe** oder **Pipeline-Segment**. Die einzelnen Pipeline-Stufen bestehen aus digitalen Logik-Schaltnetzen, welche die Funktion der Stufe realisieren.

Die Pipeline-Stufen werden durch getaktete **Pipeline-Register** getrennt, welche die jeweiligen Zwischenergebnisse der einzelnen Stufen aufnehmen bzw. weiterleiten (Abb. 2.11). Pipeline-Register sind Pipeline-interne Pufferspeicherregister für diesen speziellen Zweck der Pipelinezwischenspeicherung, die im Gegensatz zu den Registern des *Registersatzes* des Prozessors nicht beliebig lesbar oder beschreibbar sind. Sie gehören nicht zu den Architekturregistern und sind damit in der Befehlssatzarchitektur nicht sichtbar. Alle Pipeline-Register werden vom selben Taktsignal gesteuert, sie arbeiten also synchron. Gleichzeitig bilden die Pipeline-Register eine logische und zeitliche Trennung der einzelnen Pipeline-Stufen. Erst wenn alle Signale die Gatter einer Pipeline-Stufe durchlaufen haben und sich in den Pipeline-Registern ein stabiler Zustand eingestellt hat, kann der nächste Takt der Pipeline erfolgen.

Die Schaltnetze in den Pipeline-Stufen S_j besitzen u.U. unterschiedliche Verzögerungszeiten und die Pipeline-Register eine feste Verzögerungszeit. Die benötigte **Länge eines Taktzyklus** (**Taktperiode**) eines Pipeline-Prozessors wird bestimmt durch die Summe aus der Verzögerungszeit der Pipeline-

Register und der Verzögerungszeit der langsamsten Pipeline-Stufe.

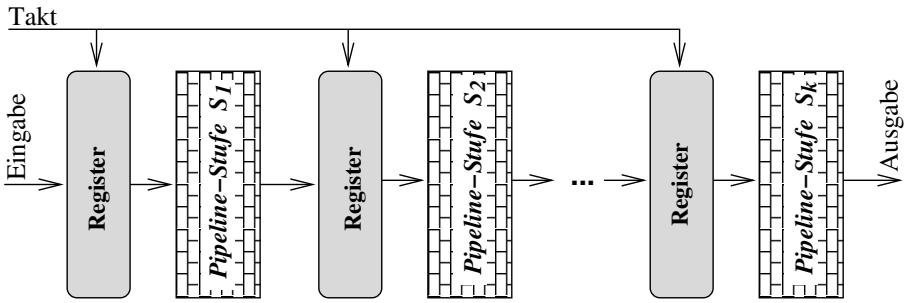


Abbildung 2.11: Ausführung einer Pipeline mit Stufen und Registern

Ein **Pipeline-Maschinentakt** ist die Zeit, die benötigt wird, um einen Befehl eine Stufe weiter durch die Pipeline zu schieben. Sobald die Pipeline „aufgefüllt“ ist, kann unabhängig von der Stufenzahl des Pipeline-Prozessors ein Ergebnis pro Taktzyklus berechnet werden. Idealerweise wird ein Befehl in einer ***k*-stufigen Pipeline** in k Takten verarbeitet. Wird in jedem Takt ein neuer Befehl geladen, dann werden zu jedem Zeitpunkt unter idealen Bedingungen k Befehle gleichzeitig behandelt und jeder Befehl benötigt k Takte bis zum Verlassen der Pipeline.

Man definiert die **Latenz** als die Zeit, die ein Befehl benötigt, um alle k Pipeline-Stufen zu durchlaufen. Der **Durchsatz** einer Pipeline wird definiert als die Anzahl der Befehle, die eine Pipeline pro Takt verlassen können. Dieser Wert spiegelt die Rechenleistung einer Pipeline wider. Im Gegensatz zu $(n \cdot k)$ Takten eines hypothetischen Prozessors ohne Pipeline dauert die Ausführung von n Befehlen in einer k-stufigen Pipeline $k+n-1$ Takte (unter der Annahme idealer Bedingungen mit einer Latenz von k Takten und einem Durchsatz von 1). Dabei werden k Taktzyklen benötigt, um die Pipeline aufzufüllen bzw. den ersten Verarbeitungsauftrag auszuführen, und $(n-1)$ Taktzyklen, um die restlichen $(n-1)$ Verarbeitungsaufträge (Befehlsbearbeitungen) durchzuführen.

Daraus ergibt sich eine **Beschleunigung** (*Speedup S*) von

$$S = \frac{n \cdot k}{k+n-1} = \frac{k}{k/n+1-1/n}$$

Ist die Anzahl der in die Pipeline gegebenen Befehle sehr groß, so ist die Beschleunigung näherungsweise gleich der Anzahl k der Pipeline-Stufen.

Durch die mittels der überlappenden Verarbeitung gewonnene Parallelität kann die Verarbeitungsgeschwindigkeit eines Prozessors beschleunigt werden, wenn die Anzahl der Pipeline-Stufen erhöht wird. Außerdem wird durch eine höhere Stufenzahl jede einzelne Pipeline-Stufe weniger komplex. Damit wird die Gattertiefe des Schaltnetzes, das die Pipeline-Stufe implementiert, geringer und dies wiederum reduziert die Signallaufzeit, sodass die Ausgangswerte der Stufen schneller an den Pipeline-Registern ankommen. Vom Prinzip her kann deshalb eine lange Pipeline schneller getaktet werden als eine kurze. Dem steht jedoch die erheblich komplexere Verwaltung gegenüber, die durch die zahlreicher auftretenden Pipeline-Konflikte benötigt wird.

Bei einer **Befehls-Pipeline** (*instruction pipeline*) wird die Bearbeitung eines Maschinenbefehls in verschiedene Phasen unterteilt. Aufeinander folgende Maschinenbefehle werden jeweils um einen Takt versetzt im Pipelining-Verfahren verarbeitet. Befehls-Pipelining ist eines der wichtigsten Merkmale moderner Prozessoren. Bei einfachen RISC-Prozessoren bestand das Entwurfsziel darin, einen durchschnittlichen CPI-Wert (cycles per instruction, Taktzyklen pro Befehl) zu erhalten, der möglichst nahe bei 1 liegt. Heutige Hochleistungsprozessoren kombinieren das Befehls-Pipelining mit weiteren Mikroarchitekturtechniken wie der Superskalartechnik, der VLIW- und der EPIC-Technik, um bis zu sechs Befehle oder mehr pro Takt ausführen zu können.

2.4.3 Befehls-Pipelining

Voraussetzung für ein effizientes Befehls-Pipelining ist die Zerlegbarkeit der Befehlsausführung in einzelne Phasen, aus denen sich die Zahl und Art der Pipeline-Stufen ergibt. Die zentrale Frage lautet also: Wie teilen wir die Verarbeitung oder Ausführung eines Befehls in Phasen auf? Als Erweiterung des zweistufigen Konzepts aus Abschnitt 2.4.1 kann man eine Befehlsverarbeitung beispielsweise folgendermaßen feiner unterteilen:

- Befehl bereitstellen,
- Befehl decodieren,
- Operanden (in den Pipeline-Registern) vor der ALU bereitstellen,
- Operation auf der ALU ausführen und
- das Resultat zurückschreiben.

Da alle diese Phasen als Pipeline-Stufen etwa gleich lang dauern sollten, gelten die folgenden Randbedingungen:

- Die Befehlsbereitstellung sollte möglichst immer in einem Takt erfolgen. Das kann nur unter der Annahme eines sog. Code-Cache-Speichers (näheres siehe KE3) auf dem Prozessor-Chip geschehen. Falls der Befehl aus dem Speicher geholt werden muss, wird die Pipeline-Verarbeitung für einige Takte unterbrochen.
- Vorteilhaft für die Befehlsdecodierung ist ein einheitliches Befehlsformat und eine geringe Komplexität der Befehle. Das sind Eigenschaften, die für die RISC-Architekturen als Ziele formuliert wurden. Falls die Befehle unterschiedlich lang sind, muss erst die Länge des Befehls durch einen aufwändigen Decodierschritt erkannt werden.
- Vorteilhaft für die Operandenbereitstellung ist eine sog. Register-Register-Architektur (ebenfalls eine Eigenschaft, die RISC-Architekturen auftritt). Bei einer Register-Register-Architektur können die Operanden der arithmetisch-logischen Befehle nur aus den Registern des Registersatzes in die Pipeline-Register bezogen werden. Falls die Operanden sich noch nicht im Registersatz befinden, müssen sie vorher aus dem Speicher dort hin geladen werden. Falls Speicheroperanden ebenfalls direkt zugelassen sind, wie es bei CISC-Architekturen der Fall ist, so dauert das Laden der Operanden eventuell mehrere Takte und der Pipeline-Fluss muss unterbrochen werden.
- Für Befehlsformate einheitlicher Länge und Befehle mit geringer Komplexität ist die Befehlsdecodierung so einfach, dass sie mit der Operandenbereitstellung aus den Registern eines Registersatzes zu einer Stufe zusammengefasst werden kann. In diesem Fall kann ein Befehl in der ersten Takthälfte dieser Phase decodiert und die Universal- bzw. Architektur-Register der Operanden können angesprochen werden. In der zweiten Takthälfte werden dann die Operanden aus den Universal-Registern in die Pipeline-Register übertragen.
- Die Ausführungsphase kann für einfache arithmetisch-logische Operationen in einem Takt durchlaufen werden. Komplexere Operationen wie die Division oder die Gleitkommaoperationen benötigen mehrere Takte, was die Organisation der Pipeline erschwert.
- Bei Lade- und Speicheroperationen muss erst die effektive Adresse berechnet werden, bevor auf den Speicher zugegriffen werden kann. Der Speicherzugriff ist wiederum besonders schnell, wenn das Speicherwort aus dem Daten-Cache-Speicher (näheres siehe KE3) gelesen oder dorthin geschrieben werden kann. Im Falle eines Daten-Cache-Fehlzugriffs oder bei Prozessoren ohne Daten-Cache dauert der Speicherzugriff mehrere Takte, in denen die anderen Pipeline-Stufen leer laufen. In der nachfolgend betrachteten DLX-Pipeline wird deshalb nach der Ausführungsphase, in der die effektive Adresse berechnet wird, eine zusätzliche Speicherzugriffsphase eingeführt, in der der Zugriff auf den Daten-Cache-Speicher durchgeführt wird.

- Das Rückschreiben des Resultatwertes in ein Register des Registersatzes kann in einem Takt oder sogar einem Halbtakt der Pipeline geschehen. Das Rückschreiben in den Speicher nach einer arithmetisch-logischen Operation, wie es durch Speicheradressierungen von CISC-Prozessoren möglich ist, dauert länger und erschwert die Pipeline-Organisation.

2.4.4 Die DLX-Pipeline

In diesem Abschnitt wollen wir ein Beispiel für eine Befehlsausführung mit Pipelining betrachten. Die sog. DLX-Pipeline ist eine sehr einfache hypothetische Befehls-Pipeline mit den in Abb. 2.12 dargestellten fünf Stufen. Die überlappende Ausführung führt zu einer fünfstufigen Pipeline mit den folgenden Phasen der Befehlsausführung:

- **Befehlsbereitstellungs**- oder **IF-Phase** (*Instruction Fetch*): Der Befehl, der durch den Befehlszähler adressiert ist, wird aus dem Cache-Speicher (*genauer: Befehls-Cache*) in einen Befehlspuffer geladen. Der Befehlszähler PC wird weitergeschaltet.
- **Decodier**- und Operandenbereitstellungsphase oder **ID-Phase** (*Instruction Decode/Register Fetch*): Aus dem Operationscode des Maschinenbefehls werden Pipeline-interne Steuersignale generiert. Die Operanden werden aus Registern bereitgestellt.
- **Ausführungs**- oder **EX-Phase** (*Execute/Address Calculation*): Die Operation wird von der ALU auf den Operanden ausgeführt. Bei Lade-/Speicherbefehlen berechnet die ALU die effektive Adresse.
- **Speicherzugriffs**- oder **MEM-Phase** (*Memory Access*): Der Speicherzugriff auf den Cache (*genauer: Daten-Cache*) wird durchgeführt.
- **Resultatspeicher**- oder **WB-Phase** (*Write Back*): Das Ergebnis wird in ein Register geschrieben.

In der ersten Phase wird ein Befehl von einer Befehlsbereitstellungseinheit geladen. Wenn dieser Vorgang beendet ist, wird der Befehl zur Decodiereinheit weitergereicht. Während die zweite Einheit mit ihrer Aufgabe beschäftigt ist, wird von der ersten Einheit bereits der nächste Befehl geladen.

Im Idealfall bearbeitet diese fünfstufige Pipeline fünf aufeinander folgende Befehle gleichzeitig, jedoch befinden sich die Befehle jeweils in einer unterschiedlichen Phase der Befehlsausführung. Die

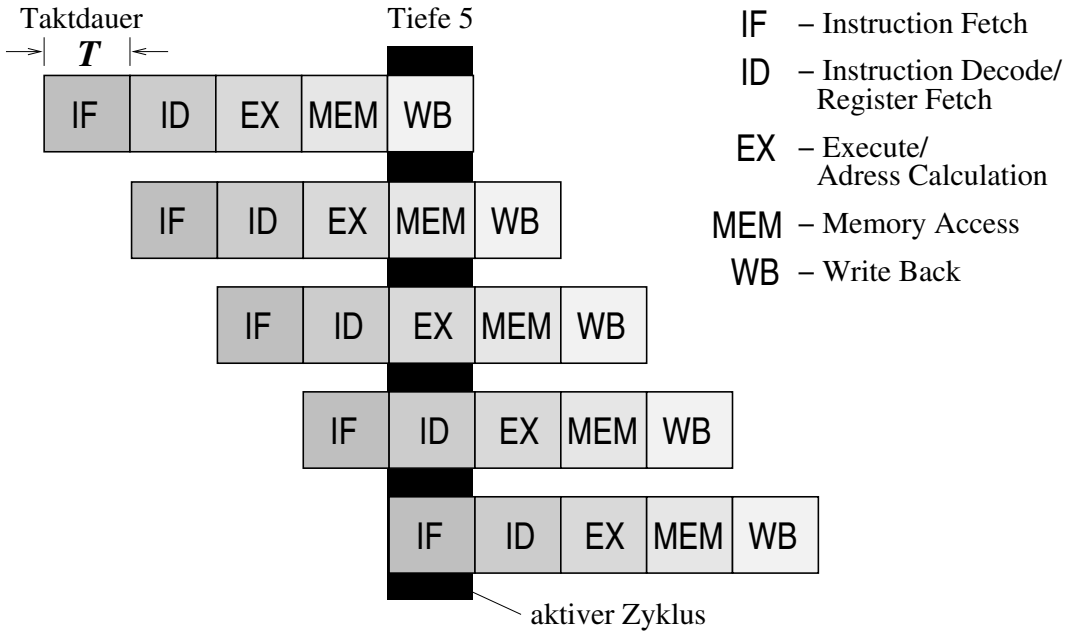


Abbildung 2.12: Stufen der DLX-Befehlspipeline

RISC-Architektur des DLX-Prozessors führt die meisten Befehle in einem Takt aus. Ausnahmen sind die Gleitkommabefehle. Somit beendet im Idealfall jede Pipeline-Stufe ihre Ausführung innerhalb eines Takts und es wird auch nach jedem Taktzyklus ein Resultat erzeugt.

Im Folgenden wollen wir die Befehlsausführung innerhalb der fünf Stufen der DLX-Pipeline etwas genauer beschreiben.

- **IF**: In der *IF-Stufe* wird der Befehl, auf den der Befehlszähler zeigt, aus dem Code-Cache in das Befehlsregister geholt. Der Befehlszähler wird entweder erhöht oder im Falle eines vorangegangenen Sprungbefehls auf die Zieladresse gesetzt um auf den nächsten auszuführenden Befehl im Speicher zu zeigen.
- **ID**: In der *ID-Stufe* wird der im Befehlsregister stehende Befehl decodiert. Danach wird abhängig vom Befehlstyp eine der folgenden Aktionen ausgeführt:
  Bei einem Speicherzugriff (Lade- oder Speicherbefehl) wird die ALU zur Berechnung der effektiven Adresse eingesetzt. Zu diesem Zweck werden in dieser Stufe in Abhängigkeit von der Adressierungsart die dazu notwendigen Werte in die ALU-Eingaberegister geladen. Beispielsweise wird bei der registerindirekten Adressierung mit Verschiebung der Inhalt eines Architekturregisters in ein Eingaberegister geladen und der Wert der Verschiebung (*displacement*) aus dem decodierten Befehl in ein Weiteres.
  Bei einem Verzweigungsbefehl wird die ALU zur Berechnung des Sprungziels eingesetzt. Auch hier werden wieder die dazu notwendigen Werte in die ALU-Eingaberegister geladen. Zum Beispiel wird bei einem unbedingten Sprung mit befehlszählerrelativer Adressierung der Inhalt des Befehlszählers in ein Eingaberegister geladen und die relative Adressangabe aus dem decodierten Befehl in das Andere.
  Im Falle der Ausführung eines Rechenbefehls (arithmetisch-logischer Befehl) werden die Operandenwerte aus den Architektur-Registern in die Eingaberegister der ALU kopiert.
- **EX**: In der *EX-Stufe* erhält die arithmetisch-logische Einheit ihre Operanden aus den ALU-Eingaberegistern. Die Inhalte der Eingaberregister hängen vom Befehlstyp ab. Die ALU führt die dem Befehlstyp entsprechende Operation auf den Operanden aus dem ersten und dem zweiten ALU-Eingaberegister aus und legt das Ergebnis im ALU-Ausgaberegister ab.
- **MEM**: Die *MEM-Stufe* wird nur für Lade- und Speicherbefehle benötigt.
  Im Falle eines Rechenbefehls wird daher der Wert des ALU-Ausgaberegisters in ein Ergebnisregister der MEM-Stufe übertragen.
  Im Falle eines Ladebefehls wird das Speicherwort so, wie es vom ALU-Ausgaberegister adressiert wird, aus dem Daten-Cache gelesen und in einem weiteren Ergebnisregister der MEM-Stufe abgelegt.
  Im Falle eines Speicherbefehls wird das zu speichernde Datum in den Daten-Cache geschrieben, wobei der Inhalt des ALU-Ausgaberegisters als Adresse benutzt wird.
  Im Falle eines Sprungbefehls mit einem genommenen Sprung wird das Befehlszähler-Register (PC) durch den Inhalt des ALU-Ausgaberegisters ersetzt.
- **WB**: In der *WB-Stufe* werden die Inhalte der Ergebnisregister der MEM-Stufe in die Architekturregister übertragen, wenn dies der zu verarbeitende Befehlstyp erfordert.

In Abb. 2.11 wird nur der Datenfluss durch die Pipeline-Stufen gezeigt. Die Steuerungsinformation, die während der ID-Stufe aus dem Opcode generiert wurde, fließt durch die nachfolgenden Pipeline-Stufen und steuert damit die Funktion der ALU.

Alle Pipeline-Stufen benutzen unterschiedliche Ressourcen, die ihnen zugeordnet sind. Deshalb werden zum Beispiel, nachdem ein Befehl zur ID geliefert wurde, die von der IF genutzten Ressourcen frei und zum Holen des nächsten Befehls benutzt. Idealerweise wird in jedem Takt ein neuer Befehl geholt und an die ID-Stufe weitergeleitet. Die Taktzeit wird durch den „kritischen Pfad“ vorgegeben, das bedeutet durch die langsamste Pipeline-Stufe. Ideale Bedingungen bedeuten, dass die Pipeline immer mit aufeinander folgenden Befehlen bzw. deren Befehlsausführungen gefüllt sein muss.

Leider gibt es mehrere potentielle Probleme, die eine reibungslose Befehlsausführung in der Pipeline stören können. Wenn zum Beispiel sowohl der Befehls- als auch der Daten-Cache nachgeladen werden müssen und nur ein Kanal zum Speicher (*memory port*) existiert, so erzeugt ein Ladebefehl in der MEM-Stufe einen Speicher-Lesekonflikt zwischen der IF- und der MEM-Stufe. In diesem Fall muss die Pipeline einen der Befehle anhalten, bis der benötigte Speicherkanal zum Nachladen des zweiten Caches wieder verfügbar ist. Auch gehen wir davon aus, dass der Registersatz mit zwei Lesekanälen und einem Schreibkanal ausgestattet ist, so dass gleichzeitig sowohl in der ID-Stufe zwei Operanden aus den Registern gelesen werden können als auch in der WB-Stufe ein Resultat in ein Register geschrieben werden kann. Trotzdem können bestimmte Hemmnisse die reibungslose Ausführung in einer Pipeline stören und zu den sog. Pipeline-Konflikten führen. Diese Pipeline-Konflikte wollen wir im nächsten Abschnitt ausführlicher betrachten und Möglichkeiten, wie man sie eliminiert oder zumindest ihre Auswirkung mindert, diskutieren.

2.5 Pipeline-Konflikte und deren Lösung

Als **Pipeline-Konflikt** bezeichnet man die Unterbrechung des taktsynchronen Durchlaufs der Befehle durch die einzelnen Stufen der Befehls-Pipeline. Pipeline-Konflikte werden einerseits durch Daten- und Steuerflussabhängigkeiten im Programm hervorgerufen. Dabei benötigt ein Befehl ein Resultat der Ausführung eines vorhergehenden Befehls. Andererseits können Pipeline-Konflikte durch die vorübergehende Nichtverfügbarkeit von Ressourcen (Ausführungseinheiten, Registern etc.), die durch andere Einheiten belegt sind, entstehen. Diese Abhängigkeiten können, falls sie nicht erkannt und behandelt werden, zu fehlerhaften Datenzuweisungen führen. Die Situationen, die zu Pipeline-Konflikten führen können, werden auch als **Pipeline-Hemmnisse** (*pipeline hazards*) bezeichnet.

Grundsätzlich werden drei Arten von Pipeline-Konflikten unterschieden:

- **Datenkonflikte** treten auf, wenn ein Operand in der Pipeline (noch) nicht verfügbar ist oder das Register bzw. der Speicherplatz, in den ein Resultat geschrieben werden soll, noch nicht zur Verfügung steht. Datenkonflikte werden durch Datenabhängigkeiten im Befehlsstrom erzeugt.
- **Struktur**- oder **Ressourcenkonflikte** treten auf, wenn zwei Pipeline-Stufen dieselbe Ressource benötigen, auf diese aber nur einmal zugegriffen werden kann (vgl. den oben beschriebenen Speicher-Lesekonflikt).
- **Steuerflusskonflikte** treten bei Programmsteuerbefehlen auf, wenn in der Befehlsbereitstellungsphase die Zieladresse des als nächstes auszuführenden Befehls noch nicht berechnet ist bzw. im Falle eines bedingten Sprunges noch nicht klar ist, ob überhaupt gesprungen wird.

2.5.1 Datenkonflikte

Wir betrachten zwei in einem Programm aufeinander folgende Befehle (*Instructions*) I_1 und I_2, wobei I_1 vor I_2 ausgeführt werden soll. Zwischen diesen Befehlen können verschiedene Arten von **Datenabhängigkeiten** bestehen, die **Datenkonflikte** zwischen zwei Befehlen verursachen können, wenn die beiden Befehle I_1 und I_2 so nahe beieinander sind, dass ihre Überlappung innerhalb der Pipeline ihre Zugriffsreihenfolge auf ein Register oder einen Speicherplatz im Hauptspeicher verändern würde.

- Es besteht eine **echte Datenabhängigkeit** (*true dependence*) von Befehl I_1 zu Befehl I_2, wenn I_1 seine Ausgabe in ein Register *REG* (oder in den Speicher) schreibt, das von I_2 als Eingabe gelesen wird. Eine echte Datenabhängigkeit kann also einen **Lese-nach-Schreibe-Konflikt** *(Read After Write – RAW)* verursachen.

- Es besteht eine **Gegenabhängigkeit** (*anti dependence)* von I_1 zu I_2, falls I_1 Daten von einem Register *REG* (oder einer Speicherstelle) liest, das anschließend von I_2 überschrieben wird. Durch eine Gegenabhängigkeit kann also ein **Schreibe-nach-Lese-Konflikt** *(Write After Read – WAR)* verursacht werden.

- Es besteht eine **Ausgabeabhängigkeit** (*output dependence*) von I_2 zu I_1, wenn beide in das gleiche Register *REG* (oder eine Speicherstelle) schreiben und I_2 sein Ergebnis nach I_1 schreibt. Eine Ausgabeabhängigkeit kann zu einem **Schreibe-nach-Schreibe-Konflikt** *(Write After Write – WAW)* führen.

Darstellung von Abhängigkeiten

Die drei Arten von *Datenabhängigkeiten* können in einem sog. **Präzedenzgraphen**, der auch als **Abhängigkeitsgraph** bezeichnet wird, dargestellt werden. Der Graph besitzt soviele Knoten wie Befehle betrachtet werden. In diesem Graphen besteht eine Kante oder gerichtete Verbindung von einem Knoten I_i nach I_j, wenn der Befehl I_j vom Befehl I_i abhängig ist. Daneben können solche Abhängigkeiten auch in Form einer Relation beschrieben werden. Diese Präzedenz- oder Abhängigkeitsrelation wird allgemein wie folgt definiert:

Ein Ereignis F heisst ***abhängig*** *von einem anderen Ereignis E, sofern F nicht stattfinden kann, ohne dass sich vorher E ereignet hat. Wir schreiben in diesem Fall* $E <\cdot F$.

$$\begin{aligned} E <\cdot F &= \text{„Ereignis } F \text{ ist abhängig von Ereignis } E\text{“} \\ &= \text{„Ereignis } E \text{ findet stets vor Ereignis } F \text{ statt“} \end{aligned}$$

Präzedenzrelationen sind strenge partielle Ordnungen, d.h. sie sind:

transitiv:	$(A <\cdot B) \wedge (B <\cdot C) \Rightarrow A <\cdot C$
irreflexiv:	es gibt kein Ereignis A mit $A <\cdot A$
und **asymmetrisch**:	$A <\cdot B \Rightarrow \neg(B <\cdot A)$

Irreflexivität schließt aus, dass es ein Ereignis gibt, welches bereits stattgefunden haben muss, bevor es stattfinden kann (Schleifenfreiheit).

Nebenläufigkeit

Mit der Definition der Präzedenz kann man leicht eine gegensätzliche Eigenschaft definieren, die Nebenläufigkeit. Zwei verschiedene Ereignisse A und B, für die in einem Prozess keine Reihenfolge festgelegt ist, für die also weder $A <\cdot B$ noch $B <\cdot A$ gilt, heißen *nebenläufig* (concurrent), und wir schreiben in diesem Fall:

$$A \; co \; B$$

Nebenläufige Ereignisse eines Prozesses können zeitgleich, überlappend oder in einer beliebigen Reihenfolge stattfinden, da kein ursächlicher Zusammenhang zwischen ihnen besteht. Prozesse ohne Nebenläufigkeit heissen *sequentielle Prozesse*. Die Präzedenzrelation bildet in diesem Fall eine strenge *totale* Ordnung, das heisst für jedes Paar E, F von Ereignissen gilt genau eine der folgenden drei Beziehungen

$$A <\cdot B \text{ oder } A = B \text{ oder } B <\cdot A$$

Grundsätzlich können natürlich alle vorher beschriebenen Pipeline-Konflikte durch solche Präzedenzen bzw. Abhängigkeiten formal beschrieben werden. Eine Abhängikeit auf einer k-stufigen Pipeline kann allgemein folgendermaßen ausgedrückt werden:

$$A_i(l) <\cdot A_j(m)$$

Dabei bezeichne $A_i(l)$ für $l \in \{1,\ldots,k\}$ die Ausführung des l-ten Teilschrittes des Befehls A_i. Da die verschiedenen Teilschritte eines jeden Befehls der Reihe nach ausgeführt werden müssen, gilt natürlich $A_i(l) <\cdot A_i(m)$ für $l, m \in \{1,\ldots,k\}$ mit $l < m$.

Beispiel 2.1 *Darstellung der Abhängigkeiten in einem Präzedenzgraphen*
Wir betrachten die folgende Drei-Adress-Befehlsfolge:

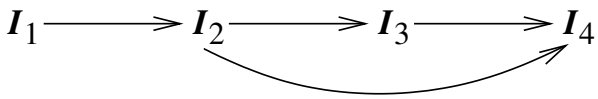
	Befehl		**Bedeutung**
I_1	ADDI	R1,R2,#2	R1=R2+2
I_2	ADD	R4,R1,R3	R4=R1+R3
I_3	MUL	R3,R5,#3	R3=R5*3
I_4	MUL	R3,R6,#3	R3=R6*3

In dieser Befehlsfolge bestehen folgenden Abhängigkeiten:

- eine echte Datenabhängigkeit von I_1 nach I_2, da I_2 den Wert oder Inhalt von Register R1 benutzt, der erst in Befehl I_1 berechnet wird;
- eine Gegenabhängigkeit von I_2 nach I_3, da I_2 den Wert von Register R3 benutzt, bevor Register R3 in Befehl I_3 einen neuen Wert zugewiesen bekommt;
- eine Ausgabeabhängigkeit von Befehl I_3 nach Befehl I_4, da Befehl I_3 und Befehl I_4 beide dem Register R3 neue Werte zuweisen.

- eine Gegenabhängigkeit von I_2 nach I_4, da I_2 den Wert von Register R3 benutzt, bevor Register R3 in Befehl I_4 einen neuen Wert zugewiesen bekommt.

Die Darstellung dieser vier Abhängigkeiten in einem Präzedenzgraphen ergibt:



Die Darstellung durch eine Relation ergibt:

$$<\cdot = \{ (I_1, I_2), (I_2, I_3), (I_3, I_4), (I_2, I_4) \}$$

Die ersten drei Abhängigkeiten in der Relation erzeugen Datenkonflikte in der Reihenfolge RAW, WAR, WAW. Man erkennt, dass es sich bei der vierten Abhängigkeit (I_2, I_4) um eine transitive Abhängigkeit handelt, da $(I_2 <\cdot I_3) \wedge (I_3 <\cdot I_4) \Rightarrow I_2 <\cdot I_4$ gilt. Solche transitiven Abhängigkeiten müssen nicht explizit behandelt werden, da dies implizit durch zwei andere Abhängigkeiten bereits geschieht. □

Gegenabhängigkeiten und Ausgabeabhängigkeiten werden häufig auch **falsche Datenabhängigkeiten** oder entsprechend dem englischen Begriff *Name Dependency* **Namensabhängigkeiten** genannt. Diese Arten von Datenabhängigkeiten sind nicht problemimmanent, sondern werden durch die Mehrfachnutzung von Speicherplätzen (in Registern oder im Arbeitsspeicher) hervorgerufen. Sie können durch Registerumbenennungen entfernt werden.

Echte oder wahre Datenabhängigkeiten werden häufig auch einfach als Datenabhängigkeiten bezeichnet. Echte Datenabhängigkeiten repräsentieren den Datenfluss durch ein Programm.

Aufgabe 2.3 *Daten- und Steuerflussabhängigkeiten*
Es sei folgende Programmsequenz gegeben:

	Befehl		**Bedeutung**
I_1	ADDI	R1,R2,#2	R1=R2+2
I_2	SUB	R4,R1,R3	R4=R1-R3
I_3	SGE	R7,R4,R0	R4 $\geq$ 0?, Status in R7
I_4	BNEZ	R7,I7	wenn ja, gehe zu I_7
I_5	MUL	R3,R5,R6	R3=R5*R6
I_6	JMP	I8	Springe nach I8
I_7	ADDI	R3,R3,#2	R3=R3+2
I_8	ADDI	R4,R4,#1	R4=R4+1

Bestimmen Sie alle Daten- und Steuerflussabhängigkeiten in dieser Programmsequenz.
Stellen Sie die Datenabhängigkeiten in einem Abhängigkeitsgraphen dar. ◇

2.5.2 Lösung von Datenkonflikten

Datenkonflikte werden zwar durch Datenabhängigkeiten hervorgerufen, sind jedoch auch wesentlich von der Pipeline-Struktur bestimmt. Wenn die datenabhängigen Befehle weit genug voneinander entfernt sind, so wird kein Datenkonflikt ausgelöst. Wie weit die Befehle voneinander entfernt sein müssen, hängt jedoch von der Pipeline-Struktur ab. Für einfache skalare Pipelines wie unsere DLX-Pipeline sind nur Lese-nach-Schreibe-Konflikte von Bedeutung. Für Superskalarprozessoren,

die mehr als einen Befehl pro Takt in die Pipeline aufnehmen, müssen jedoch auch Schreibe-nach-Lese- und Schreibe-nach-Schreibe-Konflikte beachtet werden.

Schreibe-nach-Lese-Konflikte (WAR) können nur dann in einer Pipeline auftreten, wenn die Befehle einander bereits *vor* der Operandenbereitstellung überholen können. Das heißt, ein Schreibe-nach-Lese-Konflikt tritt auf, wenn ein nachfolgender Befehl bereits sein Resultat in ein Register schreibt, bevor der in Programmreihenfolge vorherige Befehl den Registerinhalt als Operanden liest. Dieser Fall ist in unserer einfachen Pipeline ausgeschlossen, er muss jedoch für die Pipelines heutiger Superskalarprozessoren bedacht sein.

Schreibe-nach-Schreibe-Konflikte (WAW) treten nur in Pipelines auf, die in mehr als einer Stufe auf ein Register (oder einen Speicherplatz) schreiben können, oder die es einem Befehl erlauben, in der Pipeline-Verarbeitung fortzufahren, obwohl ein vorhergehender Befehl angehalten worden ist. In der einfachen DLX-Pipeline ist das erstere nicht möglich, da nur in der WB-Stufe auf die Register geschrieben werden kann. Das „Überholen“ von Befehlen ist in dieser Pipeline ebenfalls nicht möglich.

Deshalb können in unserer einfachen Pipeline nur Lese-nach-Schreibe-Konflikte auftreten, die sich wie folgt auswirken können: Betrachten wir eine Folge von zwei Register-Register-Befehlen I_1 und I_2, bei denen I_2 von I_1 datenabhängig ist und I_1 vor I_2 in der Pipeline ausgeführt wird. Nehmen wir an, dass das Ergebnis von I_1 zu I_2 über das Register REG transferiert wird. Es tritt kein Problem auf, wenn die beiden Befehle ohne Pipeline ausgeführt werden. In einer Ausführung mit Pipelining liest I_2 jedoch während der ID-Stufe den Inhalt von REG. Falls I_2 unmittelbar nach I_1 in der Pipeline ausgeführt wird, dann ist zu diesem Zeitpunkt I_1 immer noch in der EX-Stufe und wird das Ergebnis in seiner WB-Stufe zwei Takte später nach REG schreiben. Deshalb liest, wenn nichts unternommen wird, I_2 in seiner ID-Stufe den alten Wert von REG.

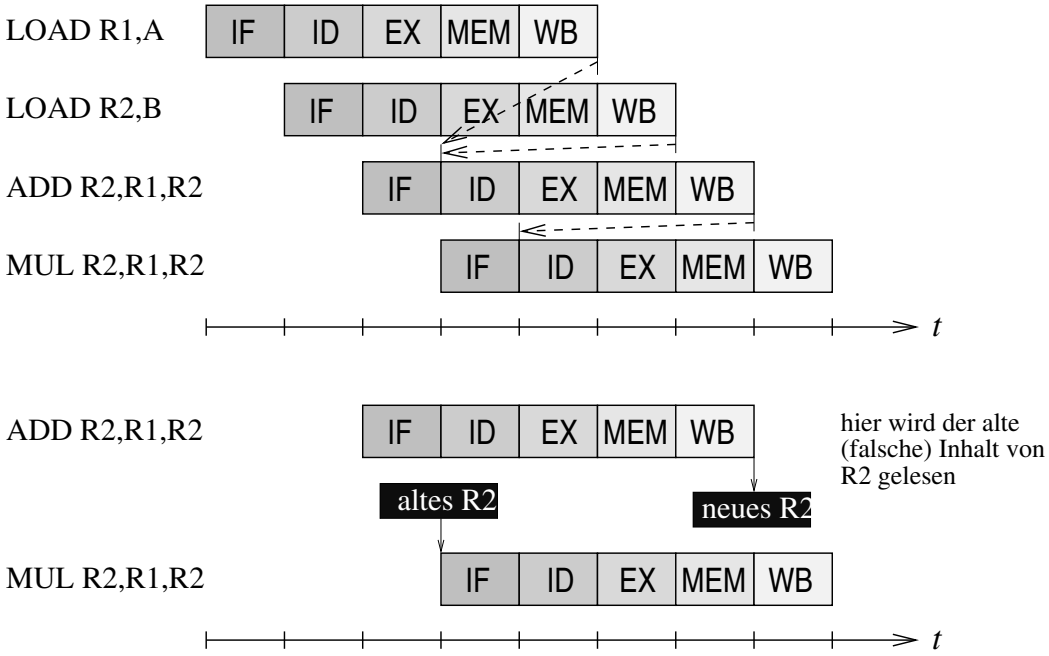


Abbildung 2.13: Datenkonflikte in der DLX-Befehlspipeline

Abb. 2.13 zeigt am Beispiel einer Befehlsfolge Lese-nach-Schreibe-Konflikte (RAW) in der DLX-Pipeline. A und B stehen symbolisch dabei für eine Speicheradresse. Vor der Addition müssen beide Operanden des ADD-Befehls erst mit Hilfe zweier Lade-Befehle in die Register R1 und R2 geladen werden. Nach der Addition der beiden Register R1 und R2 wird das Ergebnis der Addition, dass im Register R2 abgelegt ist, vom nachfolgenden MUL-Befehl als Operand benötigt. Die auftretenden (echten) Datenabhängigkeiten sind in der Abbildung so eingezeichnet, wie sie in der Pipeline-Verarbeitung der Befehle auftreten würden (unechte Datenabhängigkeiten sind in der DLX-Pipeline ohne Auswirkungen und wurden deshalb weggelassen). Die Tatsache, dass die Pfeile für den Daten-

fluss in die Rückrichtung der Zeitachse deuten, demonstriert, dass durch die Datenabhängigkeiten Datenkonflikte hervorgerufen werden, die bei Nichtbehandlung zu falschen Ergebnissen führen.

Datenkonflikte in der Pipeline müssen also erkannt und behandelt werden, sonst wird keine korrekte Programmausführung erzielt. Zur Behandlung der Datenkonflikte in der Pipeline gibt es zwei grundsätzliche Möglichkeiten der Software- und der Hardware-Lösung.

Software-Lösung

Die Nichtbehandlung der Datenkonflikte durch die Hardware hat zur Folge, dass die Pipeline-Organisation in der Architektur offengelegt und die Datenkonflikte durch Software (durch den Compiler oder den Assemblerprogrammierer) behandelt werden müssen. Dieser Fall trifft für unsere einfache DLX-Pipeline zu. Eine korrekte Ausführung direkt aufeinander folgender und voneinander datenabhängiger Befehle ist nicht gewährleistet, sondern muss im Maschinenprogrammablauf bedacht werden. Diese einfachste Form einer Pipeline-Organisation war in den ersten RISC-Prozessoren üblich, ist jedoch wegen der höchst fehleranfälligen Programmierung heute obsolet.

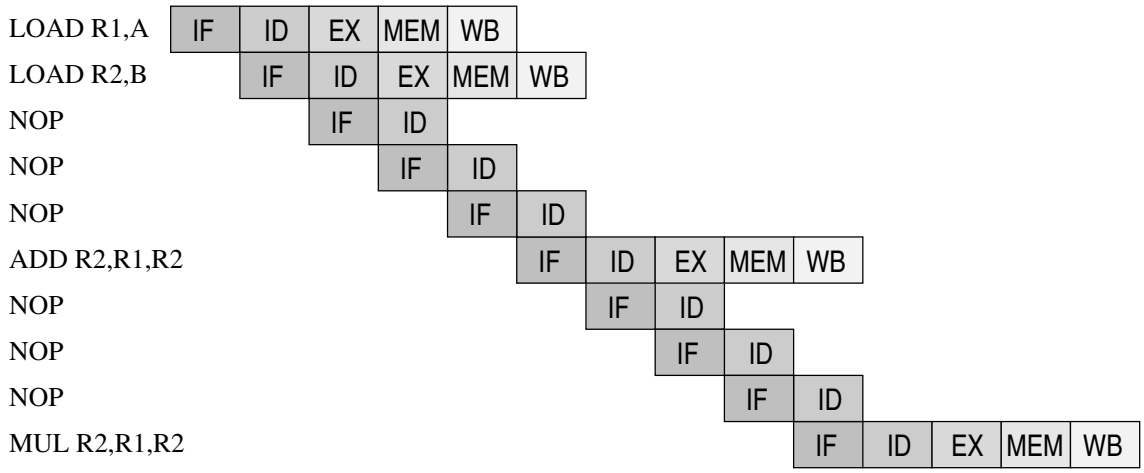


Abbildung 2.14: Eliminierung von Datenkonflikten durch NOPS

Falls die Pipeline nicht in der Lage ist, Konflikte mittels Hardware zu erkennen, muss der Compiler die Ausführung der Pipeline steuern. Das kann durch Einstreuen von Leerbefehlen (zum Beispiel NOP-Befehle, *no operation*) nach jedem datenabhängigen Befehl geschehen, der einen Pipeline-Konflikt verursachen würde. Die Anzahl der notwendigen Leerbefehle, deren Ausführung einem Stillstand der Pipeline gleichkommt, kann häufig durch den Compiler reduziert werden. Der Compiler kann den Programmcode mit dem Ziel neu anordnen, solche Leerbefehle möglichst zu eliminieren. Diese Methode wird **Befehlsanordnung** (*instruction scheduling* oder *pipeline scheduling*) genannt. Eine solche compilerbasierte Befehlsanordnung wurde bereits bei den ersten RISC-Prozessoren angewendet. Im Beispiel 2.14 sollten also wenn möglich zwei andere Befehle, die keinen neuen Datenkonflikt hervorrufen, zwischen dem ADD- und dem MUL-Befehl eingefügt werden.

Beschleunigte DLX-Pipeline

In der Literatur über die DLX-Pipeline wird häufig eine zusätzliche Annahme zur Beschleunigung der DLX-Pipeline gemacht. Dazu wird für die WB-Phase angenommen, dass das Zurückschreiben der Ergebnisregister in die Architektur-Register in der ersten Hälfte der WB-Phase abgeschlossen ist. Weiter wird für die ID-Phase angenommen, dass die Decodierung des Befehls in der ersten Hälfte der ID-Phase stattfindet und die Operanden in der zweiten Hälfte der ID-Phase aus den Architektur-Registern in die ALU-Eingaberegister geladen werden. Als Folge dieser Annahme kann die ID-Phase eines nachfolgenden datenabhängigen Befehls gleichzeitig mit der WB-Phase des ergebniserzeugenden Befehls ausgeführt werden. Damit werden statt der drei Leerbefehle im Standardfall nur noch zwei Leerbefehle benötigt.

Hardware-Lösung

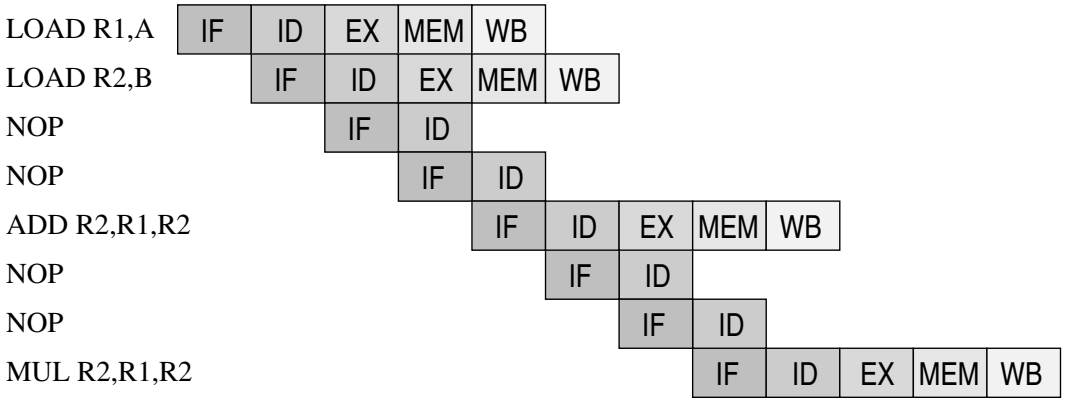


Abbildung 2.15: Beschleunigte DLX-Pipeline

Die Hardware-Lösung erfordert das Erkennen der Datenkonflikte durch die Hardware und deren automatisches Behandeln. Dies ist heute Stand der Technik. Dabei unterscheiden wir im wesentlichen die im Folgenden dargestellten drei Hardware-Lösungen für das Datenkonflikt-Problem.

1. Leerlauf der Pipeline

Die einfachste Art, mit solchen Datenkonflikten umzugehen ist es, den auf I_1 folgenden datenabhängigen Befehl I_2 in der Pipeline für drei bzw. zwei Takte anzuhalten. Hardware-Erkennung von Pipeline-Konflikten und deren Behandlung durch Anhalten der Pipeline wird als **Pipeline-Sperrung** (*interlocking*) oder **Pipeline-Leerlauf** (*stalling*) bezeichnet. Durch das Stoppen der Befehlsausführungen entstehen sog. **Pipeline-Blasen** (*pipeline bubbles*), die den gleichen Effekt wie Leerbefehle hervorrufen, nämlich, die Ausführungsgeschwindigkeit deutlich herabzusetzen. Im Beispiel der Abb. 2.16 erzeugt das Anhalten zwei Pipeline-Blasen, wenn wir annehmen, dass das Zurückschreiben in der ersten Hälfte der WB-Stufe abgeschlossen wird.

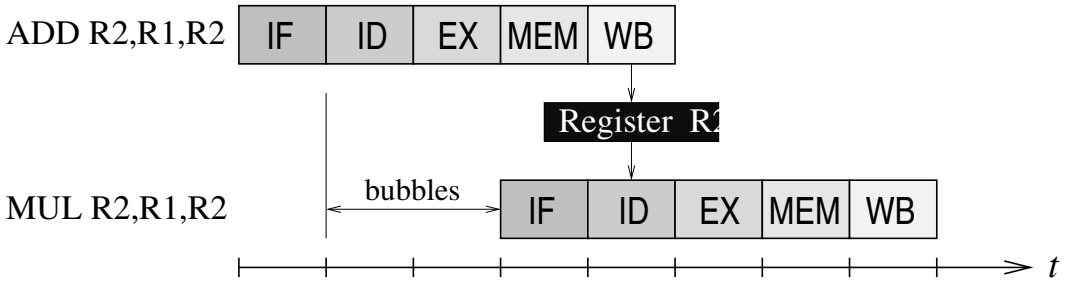


Abbildung 2.16: Datenkonflikt: Hardware-Lösung durch Interlocking

2. Forwarding

Eine bessere Lösung als der Leerlauf ist das Forwarding, das allerdings etwas mehr Hardware-Aufwand erfordert und auch als *Bypass* bezeichnet wird. Wenn ein Datenkonflikt erkannt wird, so sorgt eine Hardwareschaltung dafür, dass der betreffende Operand nicht erst aus dem Architektur-Register, sondern direkt aus dem ALU-Ausgaberegister der vorherigen Operation in das ALU-Eingaberegister übertragen wird. Übertragen auf unsere DLX-Pipeline mit der Annahme, dass das Zurückschreiben in der ersten Hälfte der WB-Stufe abgeschlossen wird, bedeutet dies, dass der nachfolgende datenabhängige Befehl nicht warten muss, bis das Ergebnis des davor stehenden Befehls während der WB-Phase in das Architektur-Register geschrieben wird. Betrachtet man wieder das Beispiel mit dem ADD- und MUL-Befehl, so können die beiden unmittelbar nacheinander in der Pipeline begonnen werden (Abb. 2.17). Das Ergebnis des ADD-Befehls im ALU-Ausgaberegister der EX-Stufe wird sofort zurück zur Eingabe der ALU in der EX-Stufe als Operand für den MUL-Befehl weitergeleitet. Im betrachteten Fall, in dem beide Befehle vom Register-Register-Typ sind, beseitigt das Forwarding alle Leertakte.

3. Forwarding mit Interlocking

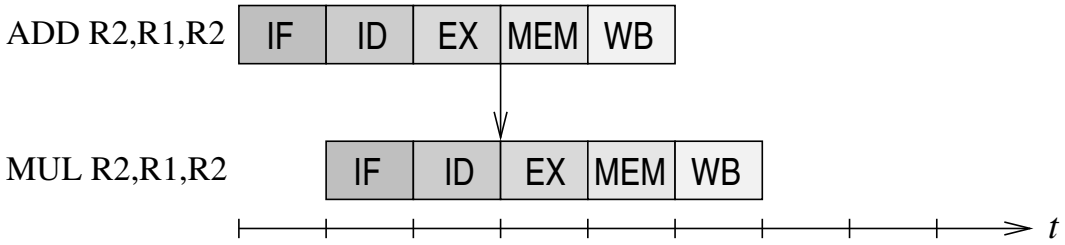


Abbildung 2.17: Datenkonflikt: Hardware-Lösung durch Forwarding

Leider löst Forwarding nicht alle möglichen Datenkonflikte auf. Wenn der Befehl I_1 ein Ladebefehl ist, dann wäre ein Forwarding vom ALU-Ausgaberegister der EX-Stufe falsch, da die EX-Stufe nicht den zu ladenden Wert erzeugt, sondern nur die effektive Speicheradresse ins ALU-Ausgaberegister schreibt. Angenommen, I_2 sei vom Ladebefehl I_1 datenabhängig, dann muss I_2 angehalten werden, bis die von I_1 geladenen Daten im Ladewertregister der MEM-Stufe verfügbar werden. Eine Lösung ist das Forwarding vom Ladewertregister der MEM-Stufe zum ALU-Eingaberegister der EX-Stufe. Dies beseitigt einen der zwei Leertakte, der zweite Leertakt kann jedoch nicht vermieden werden (siehe Beispiel in Abb. 2.18).

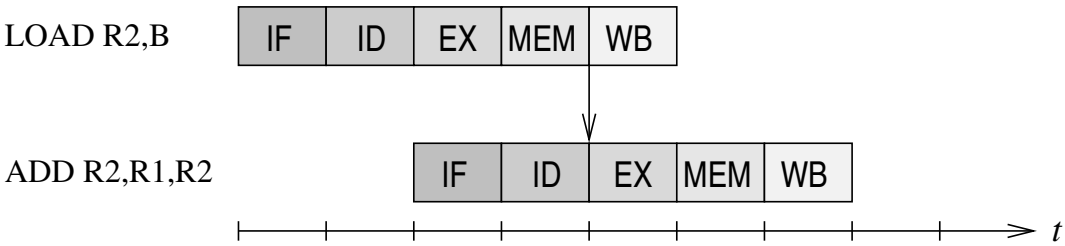


Abbildung 2.18: Datenkonflikt: Hardware-Lösung durch Forwarding mit Interlocking

Man spricht von **statischer** Befehlsanordnung, wenn der Compiler die Befehle, die Datenkonflikte erzeugen, durch Umordnung oder Leerbefehle trennt, und von **dynamischer** Anordnung der Befehle, wenn dies zur Laufzeit in Hardware geschieht. Unser einfacher Pipeline-Prozessor kann Befehle natürlich nicht zur Laufzeit umordnen, sondern nur bei geeigneter Hardware-Erweiterung, wenn nötig, in ihrer Ausführung verzögern. Dynamisches Umordnen der Befehle ist jedoch in heutigen Superskalarprozessoren möglich. Trotzdem kann eine geeignete statische Befehlsanordnung die Programmausführung beschleunigen, wenn die Pipeline-Struktur, die eigentlich zur Mikroarchitektur gehört, von einem optimierenden Compiler bedacht wird.

2.5.3 Steuerflusskonflikte

Zu den Programmsteuerbefehlen gehören die bedingten und die unbedingten Sprungbefehle, die Unterprogrammaufruf- und -rückkehrbefehle sowie die Unterbrechungsbefehle, die per Software Unterbrechungsroutinen aufrufen bzw. aus einer solchen Routine zurückkehren. Abgesehen von einem nicht genommenen bedingten Sprung erzeugen alle diese Befehle eine **Steuerflussänderung**, da nicht der nächste im Speicher stehende Befehl, sondern ein Befehl an einer Zieladresse geholt und ausgeführt werden muss. Damit ergibt sich eine Steuerflussabhängigkeit zu dem in der Speicheranordnung nächsten Befehl des Programms.

Steuerflussabhängigkeiten verursachen **Steuerflusskonflikte** in der DLX-Pipeline, da der Programmsteuerbefehl erst in der ID-Stufe als solcher erkannt und damit bereits ein Befehl des falschen Programmpfades in die Pipeline geladen wurde. Darüber hinaus muss erst die Sprungzieladresse in der ALU berechnet werden, so dass weitere Befehle des falschen Programmpfades in die Pipeline geraten, bevor der richtige Befehl durch den Befehlszähler adressiert und in die Pipeline geladen werden kann. Eine Besonderheit stellen die bedingten Sprungbefehle dar, da bei diesen Befehlen die Änderung des Programmflusses außerdem von der Auswertung der Sprungbedingung abhängt.

Steuerflusskonflikte werden in unserer DLX-Pipeline beispielsweise durch Sprünge verursacht. Sei $I_1, I_2, I_3, ...$ eine Befehlsfolge, die in dieser Reihenfolge im Speicher steht und nacheinander in die Pipeline geladen wird. Angenommen, I_1 sei ein Sprung. Die Sprungzieladresse wird in der EX-Stufe berechnet und ersetzt den Befehlszähler in der MEM-Stufe, während I_2 in der EX-Stufe, I_3 in der ID-Stufe und I_4 in der IF-Stufe ist. Unter der Annahme, dass die Sprungadresse nicht auf I_2, I_3 oder I_4 zeigt, müssen die vorher geladenen Befehle I_2, I_3 und I_4 aus der Pipeline entfernt werden und der Befehl an der Sprungadresse geladen werden.

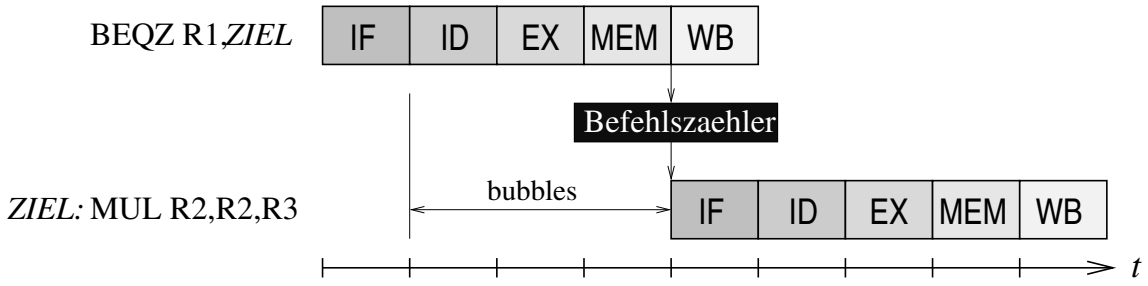


Abbildung 2.19: Leertakte nach einem genommenen bedingten Sprung

Steuerflusskonflikte treten außerdem auf, wenn I_1 ein bedingter Sprung ist, da die Sprungrichtung und die Zieladresse des Sprungs, die erforderlich ist, wenn der Sprung vollzogen wird, beide in der EX-Stufe berechnet werden (die Zieladresse des Sprungs ersetzt den PC in der MEM-Stufe). Wenn gesprungen wird, kann die korrekte Befehlsfolge mit einer Verzögerung von drei Takten gestartet werden, da drei Befehle des falschen Befehlspfades bereits in die verschiedenen Pipeline-Stufen geladen wurden (Abb. 2.19).

2.5.4 Lösung von Steuerflusskonflikten

Um die Anzahl der Wartezyklen zu mindern, sollten die Sprungrichtung und die Sprungadresse in der Pipeline so früh wie möglich berechnet werden. Das könnte in der ID-Stufe geschehen, nachdem der Befehl als Sprungbefehl erkannt worden ist. Jedoch kann die ALU dann nicht länger für die Berechnung der Zieladresse benutzt werden, da sie noch von dem vorhergehenden Befehl benötigt wird. Dies wäre sonst ein Strukturkonflikt, den man allerdings durch eine zusätzliche ALU zur Berechnung des Sprungziels in der ID-Stufe vermeiden kann. Angenommen, man habe eine *zusätzliche Adressberechnungs-ALU* und das Zurückschreiben der Zieladresse in den Befehlszähler findet schon in der ID-Stufe statt (falls der Sprung genommen wird), so ergibt sich nur *ein* Wartezyklus (Abb. 2.20).

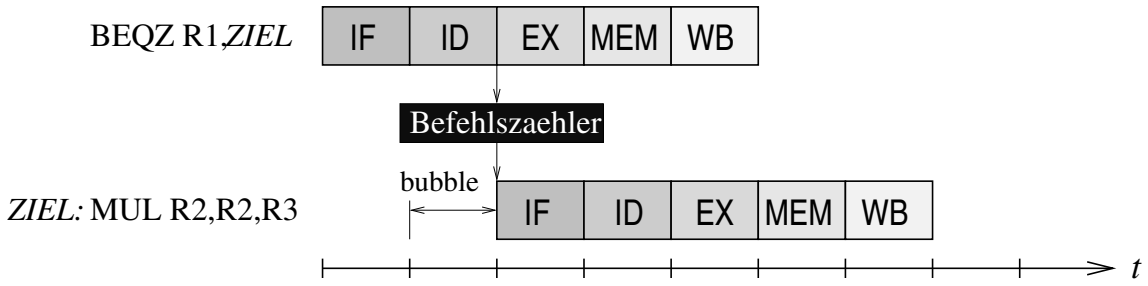


Abbildung 2.20: Leertakte mit Adressberechnungs-ALU

Obwohl die Maßnahme einer zusätzlichen Adressberechnungs-ALU die Verzögerung auf einen einzelnen Takt verringert, kann noch ein anderer, nicht behebbarer Pipeline-Konflikt entstehen. Ein ALU-Befehl gefolgt von einem bedingten Sprung, wobei der Sprungausgang vom Ergebnis dieses ALU-Befehls abhängt, wird einen Konflikt mit Verzögerung verursachen, auch wenn das Ergebnis von der EX- in die ID-Stufe weitergeleitet wird. Dieser Fall ist vergleichbar mit dem vorher beschrie-

benen Datenkonflikt eines Ladebefehls mit einer nachfolgenden ALU-Operation, die den geladenen Wert benötigt.

Das Hauptproblem bei einer solchen Reorganisation der Pipeline ist jedoch, dass die Decodierung, die Sprungberechnung und das Rückschreiben des Befehlszählers sequenziell in einer einzigen Pipeline-Stufe (hier bei DLX in der ID-Phase) ausgeführt werden müssen. Das kann zu einem kritischen Pfad in der Decodierstufe führen, der die Taktfrequenz der gesamten Pipeline reduziert. Der kritische Pfad einer Pipeline-Stufe ist der Pfad mit der längsten Signallaufzeit.

Bei der einfachen DLX-Pipeline werden Steuerflusskonflikte per Hardware weder erkannt noch behandelt. Die drei Befehle, die auf einen Sprungbefehl folgen, werden immer ausgeführt. Sie bilden die sog. **Verzögerungszeitschlitze** (*delay slots*). Auch hier wird, wie beim Nichterkennen von Datenkonflikten, die Pipeline-Implementierung von Programmsteuerbefehlen zur Architektur hin offen gelegt. Compiler oder Assemblerprogrammierer müssen die Steuerflusskonflikte beheben, damit eine korrekte Programmausführung gewährleistet bleibt.

Diese Steuerflusskonflikte können mit Hilfe von verschiedenen software-basierten Techniken behandelt werden.

1. Verzögerte Sprungtechnik (*delayed branch technique*)

Eine einfache Methode, eine korrekte Programmausführung ohne Hardware-Änderungen herzustellen, ist das Ausfüllen der Verzögerungszeitschlitze mit Leerbefehlen. Im Falle der DLX-Pipeline müssen nach jedem Programmsteuerbefehl drei Leerbefehle eingefügt werden.

2. Statische Befehlsanordnung

Der Compiler füllt den/die Verzögerungszeitschlitz(e) mit Befehlen, die in der logischen Programmreihenfolge vor dem Sprung liegen. Natürlich ist das nur möglich, wenn diese Befehle keinen Einfluss auf die Sprungrichtung haben (dabei wird angenommen, dass die Sprungadresse nicht auf einen der Befehle in den Verzögerungszeitschlitzen zeigt). In diesem Fall werden die Befehle, die in die Schlitze verschoben wurden, ohne Rücksicht auf das Sprungergebnis ausgeführt.

Praktisch alle heutigen Prozessoren behandeln Steuerflusskonflikte durch die Hardware. Dies kann schon durch die folgenden einfachen Hardware-Techniken geschehen:

1. Pipeline-Leerlauf

Dies ist wieder die einfachste, aber ineffizienteste Methode, um mit Steuerflusskonflikten umzugehen. Die Hardware erkennt in der ID-Stufe, dass der decodierte Befehl ein bedingter Sprung ist und lädt keine weiteren Befehle in die Pipeline, bis die Sprungzieladresse berechnet und im Falle bedingter Sprungbefehle die Sprungentscheidung getroffen ist. Außerdem muss der eine Befehl, der durch die IF-Stufe bereits in den Befehlspuffer geladen wurde, wieder gelöscht werden.

2. Spekulation auf nicht genommene bedingte Sprünge

Eine kleine Erweiterung des Pipeline-Leerlaufverfahrens besteht darin, darauf zu spekulieren, dass bedingte Sprünge *nicht* genommen werden. Damit können einfach die direkt nach dem Sprungbefehl stehenden drei (im Falle der DLX-Pipeline) Befehle in die Pipeline geladen werden. Falls die Sprungbedingung doch als „genommen“ ausgewertet wird, müssen die drei Befehle wieder gelöscht werden und wir erhalten die üblichen drei Pipeline-Blasen. Falls der Sprung nicht genommen wird, so können die drei Befehle als Befehle auf dem gültigen Pfad weiterverarbeitet werden, ohne dass ein Leertakt entsteht. Diese Technik stellt die einfachste der sog. *statischen Sprungvorhersagen* dar.

Statische Sprungvorhersagetechniken

Die statische Sprungvorhersage ist eine sehr einfache Vorhersagetechnik, bei der die Hardware jeden bedingten Sprung nach einem festen Muster vorhersagt. In manchen Architekturen kann der Compiler durch ein Bit die Spekulationsrichtung für alle bedingten Sprungbefehle gemeinsam festlegen, indem er die hardwarebasierte Vorhersage umkehrt. In beiden Fällen kann sich die Vorhersage für einen

speziellen Sprungbefehl nie ändern.

Eine solche „dynamische“ Änderung der Vorhersagerichtung ist jedoch bei den sog. dynamischen Sprungvorhersagetechniken der Fall, welche die Vorgeschichte des Sprungbefehls zur Sprungvorhersage nutzen. Die dynamische Sprungvorhersage ist ein Merkmal des Superskalarprozessor, über den wir im Abschnitt 2.6 dieser KE noch einiges kennen lernen wollen. Dabei werden wir auf die dynamische Sprungvorhersage zurückkommen.

Einfache Muster für die statische Vorhersage in Hardware sind:

- *Predict always not taken*: Dies ist das einfachste Schema, bei dem angenommen wird, dass kein Sprung genommen wird. Dies entspricht dem schon weiter vorne beschriebenen geradlinigen Durchlaufen eines Programms. Da die meisten Programme auf Schleifen basieren und bei jedem Schleifendurchlauf der Rücksprung als nicht genommen vorhergesagt wird, wird vor jedem erneuten Schleifendurchlauf eine Fehlspekulation stattfinden.
- *Predict always taken*: Hier wird angenommen, dass jeder Sprung genommen wird. Damit werden alle Rücksprünge bei einem wiederholten Schleifendurchlauf richtig vorhergesagt.
- *Predict backward taken, forward not taken*: Alle Sprünge, die in der Programmreihenfolge rückwärts springen, werden als „genommen“ vorhergesagt und alle Vorwärtssprünge als „nicht genommen“. Dahinter steckt die Idee, dass die Rückwärtssprünge am Ende einer Schleife fast immer genommen und alle anderen vorzugsweise nicht genommen werden.

Um eine gute Vorhersage zu erzielen, kann der Compiler folgende Techniken anwenden:

- Analyse der Programmstrukturen hinsichtlich der Vorhersage (Sprünge zurück zum Anfang einer Schleife sollten als „genommen“ vorhergesagt werden, Sprünge aus if-then-else-Konstrukten dagegen nicht).
- Der Programmierer kann über Compiler-Direktiven sein Wissen über bestimmte Abläufe in die Sprungvorhersage einbringen.
- Durch Profiling von früheren Programmabläufen kann das voraussichtliche Verhalten jedes Sprungs ermittelt werden.

2.5.5 Strukturkonflikte und deren Lösungsmöglichkeiten

Struktur- oder Ressourcenkonflikte treten in unserer einfachen DLX-Pipeline nicht auf. Schließlich ist es ein Ziel beim Pipeline-Entwurf, Strukturkonflikte möglichst zu vermeiden und da, wo sie nicht vermeidbar sind, zu erkennen und zu behandeln.

Einen Strukturkonflikt kann man demonstrieren, wenn man die DLX-Pipeline leicht verändert: Wir nehmen an, die Pipeline sei so konstruiert, dass die MEM-Stufe in der Lage ist, ebenfalls auf den Registersatz zurückzuschreiben. Betrachten wir zwei Befehle I_1 und I_2, wobei I_1 vor I_2 geholt wird und nehmen an, dass I_1 ein Ladebefehl ist, während I_2 ein datenunabhängiger Register-Register-Befehl ist. Aufgrund der Speicheradressierung kommt die Datenanforderung von I_1 in den Registern zur gleichen Zeit an wie das Ergebnis von I_2, so dass ein Ressourcenkonflikt entsteht, sofern nur ein einzelner Schreibkanal auf die Register vorhanden ist (Abb. 2.21).

Zur Vermeidung von Strukturkonflikten gibt es folgende Hardware-Lösungen:

- *Arbitrierung mit Interlocking*: Strukturkonflikte können durch eine sog. Arbitrierungslogik erkannt und aufgelöst werden. Die Arbitrierungslogik hält den im Programmfluss späteren der beiden um die Ressource konkurrierenden Befehle an. Bei dieser Technik kommt man natürlich nicht ohne Verzögerung aus. Im Beispiel in Abb. 2.21 darf der LOAD-Befehl in das Register schreiben, während das Ergebnisrückschreiben des MUL-Befehls verzögert wird.
- *Übertaktung*: Manchmal ist es möglich, die Ressource, die den Strukturkonflikt hervorruft, schneller zu takten als die übrigen Pipeline-Stufen. In diesem Fall könnte die Arbitrierungslogik zweimal auf die Ressource zugreifen und die Ressourcenanforderungen in der Ausführungsreihenfolge erfüllen.

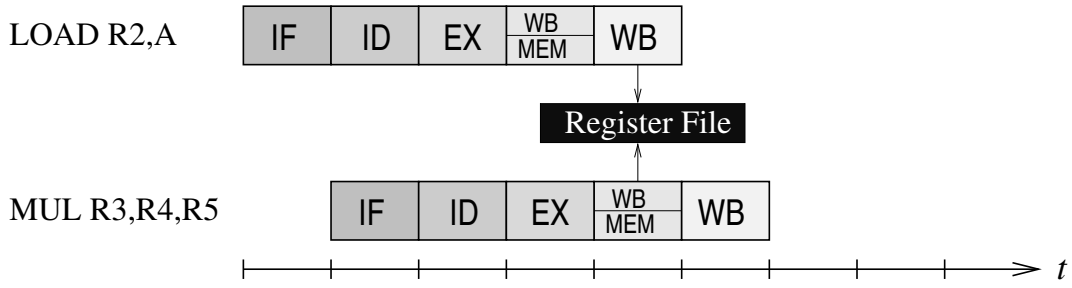


Abbildung 2.21: Beispiel eines Strukturkonflikts bei einer veränderten DLX-Pipeline

- *Ressourcenreplizierung*: Die Auswirkungen von Strukturkonflikten können durch Vervielfachen von Hardware-Ressourcen gemindert werden. Auf diese Weise treten keine Verzögerungen mehr auf. Im obigen Beispiel würde ein Registersatz mit mehreren Schreibkanälen in der Lage sein, gleichzeitig in *verschiedene* Zielregister zu schreiben. Im Falle des Schreibzugriffs beider konkurrierender Befehle auf das *gleiche* Zielregister ist jedoch wieder eine Arbitrierung und Verzögerung des zweiten Zugriffs nötig.

Aufgabe 2.4 *Einfügen von Leerbefehlen*
Wie in Aufgabe 2.3 sei folgende Programmsequenz gegeben:

	Befehl		**Bedeutung**
I_1	ADDI	R1,R2,#2	R1=R2+2
I_2	SUB	R4,R1,R3	R4=R1-R3
I_3	SGE	R7,R4,R0	R4 $\geq$ 0?, Status in R7
I_4	BNEZ	R7,I7	wenn ja, gehe zu I_7
I_5	MUL	R3,R5,R6	R3=R5*R6
I_6	JMP	I8	Springe nach I8
I_7	ADDI	R3,R3,#2	R3=R3+2
I_8	ADDI	R4,R4,#1	R4=R4+1

Es sei angenommen, dass bei der Ausführung der oben abgebildeten Programmsequenz die einzelnen Befehle in einer fünfstufigen beschleunigten DLX-Befehls-Pipeline verarbeitet werden.

Fügen Sie die zur Beseitigung aller Pipelinekonflikte minimal erforderlichen NOP-Befehle in die Programmsequenz ein. ◇

Aufgabe 2.5 *Fragen zum Pipelining*

a) Geben Sie alle Arten von Pipeline-Konflikten an, die in einer Befehls-Pipeline auftreten können. Welche Arten von Datenkonflikten gibt es?

b) Welche Datenabhängigkeiten führen bei der im Kurs gegebenen DLX-Pipeline nicht zu Pipeline-Konflikten (mit Begründung)?

c) In welche fünf Phasen kann man die Befehlsausführung im allgemeinen zerlegen?

d) Was sind die charakteristischen Eigenschaften einer RISC-Architektur?

e) Können die von CISC-Rechnern bekannten Adressierungsarten auch mit den wenigen Adressierungsarten der RISC-Rechner realisiert werden?

◇

2.6 Merkmale eines Superskalarprozessors

In diesem letzten Abschnitt dieser KE wollen wir die Superskalartechnik behandeln, die in fast allen heutigen PC- und Workstation-Prozessoren eingesetzt wird. Die wesentlichen Unterschiede eines Superskalarprozessors zu einem Prozessor mit einer skalaren Pipeline, der genau einen Befehl pro Takt in die Pipeline aufnimmt, können wie folgt zusammengefasst werden:

- Den Ausführungseinheiten kann mehr als ein Befehl pro Takt zugewiesen werden (dies motiviert den Begriff *superskalar* im Vergleich zu *skalar*).
- Die Befehle werden aus einem sequenziellen Strom von *normalen* Befehlen zugewiesen.
- Die Zuweisung der Befehle erfolgt in Hardware durch einen dynamischen Scheduler.
- Die Anzahl der zugewiesenen Befehle pro Takt wird dynamisch von der Hardware bestimmt und liegt zwischen Null und der maximal möglichen Zuweisungsbandbreite ($\leq$ Anzahl der verfügbaren Ausführungseinheiten).
- Die dynamische Zuweisung von Befehlen führt zu einem komplexen Hardware-Scheduler. Die Komplexität des Schedulers steigt mit der Größe des Befehlsfensters und mit der Anzahl der Befehle, die außerhalb der Programmreihenfolge zugewiesen werden können.
- Es ist unumgänglich, dass mehrere Ausführungseinheiten verfügbar sind. Die Anzahl der Ausführungseinheiten entspricht mindestens der Zuweisungsbandbreite, wobei es häufig noch mehr sind, um potenzielle Strukturkonflikte zu umgehen.

Ein wichtiger Punkt ist, dass die Superskalartechnik eine Mikroarchitekturtechnik ist und keinen Einfluss auf die Befehlssatz-Architektur hat. Damit kann Code, der für einen skalaren Mikroprozessor generiert wurde, ohne Änderung auch auf einem superskalaren Prozessor mit der gleichen Architektur ablaufen und umgekehrt. Das Befehls-Pipelining und die Superskalartechnik nutzen beide die sog. feinkörnige Parallelität (*fine-grain* oder *instruction-level parallelism*), d.h. Parallelität zwischen einzelnen Befehlen. Das Pipelining nutzt dabei zeitliche Parallelität und die Superskalartechnik zusätzlich die räumliche Parallelität. Eine Leistungssteigerung durch zeitliche Parallelität kann mit einer längeren Pipeline und „schnelleren“ elektrischen Bausteinen (höherer Taktfrequenz) erreicht werden. Falls genügend feinkörnige Parallelität vorhanden ist, kann die Leistungssteigerung im superskalaren Fall durch Erhöhung der räumlichen Parallelität in Form von mehr Ausführungseinheiten und einer damit möglichen höheren Zuweisungsbandbreite erreicht werden.

2.6.1 Komponenten eines erweiterten superskalaren Prozessors

Neben den oben beschriebenen Merkmalen weisen heutige Superskalarprozessoren zusätzliche Komponenten, insbesondere zur Out-of-Order-Ausführung auf, die im Folgenden beschrieben werden. Die Abb. 2.22 gibt einen Überblick. Die gezeigten Komponenten haben die folgenden Funktionen

- Die internen Speichereinheiten (*Daten- und Befehls-Cache*) sind über Busse (in der Abbildung nicht gezeichnet) mit der Busschnittstelle verbunden. Diese ermöglicht über ein gemeinsames externes Bussystem den Zugriff auf einen L2-Cache (*Level-2-Cache*, Sekundär-Cache) bzw. den Arbeits- oder Hauptspeicher, aus denen insbesondere die auszuführenden Befehle geholt werden müssen. Diese Befehle werden dann im internen Befehls-Cache abgelegt. Die Berechnung der physikalischen Befehlsadresse aus der logischen Speicheradresse wird durch eine Speicherverwaltungseinheit (*Memory Management Unit* – MMU) unterstützt.
- Eine Verzweigungseinheit (*Branch Unit*) überwacht die Ausführung von Sprungbefehlen. Nach dem Holen eines Sprungbefehls aus dem Code-Cache-Speicher (*Befehls-Cache*) ist das Sprungziel meist für mehrere Takte noch nicht bekannt. In solch einer Situation, wenn noch unbekannt ist, ob der Sprung genommen wird oder nicht, werden nachfolgende Befehle spekulativ geholt, decodiert und ausgeführt. Die Spekulation über den weiteren Programmverlauf wird dabei von einer dynamischen Sprungvorhersagetechnik entschieden, die auf der Historie der Sprünge beim Ausführen des Programms basiert.

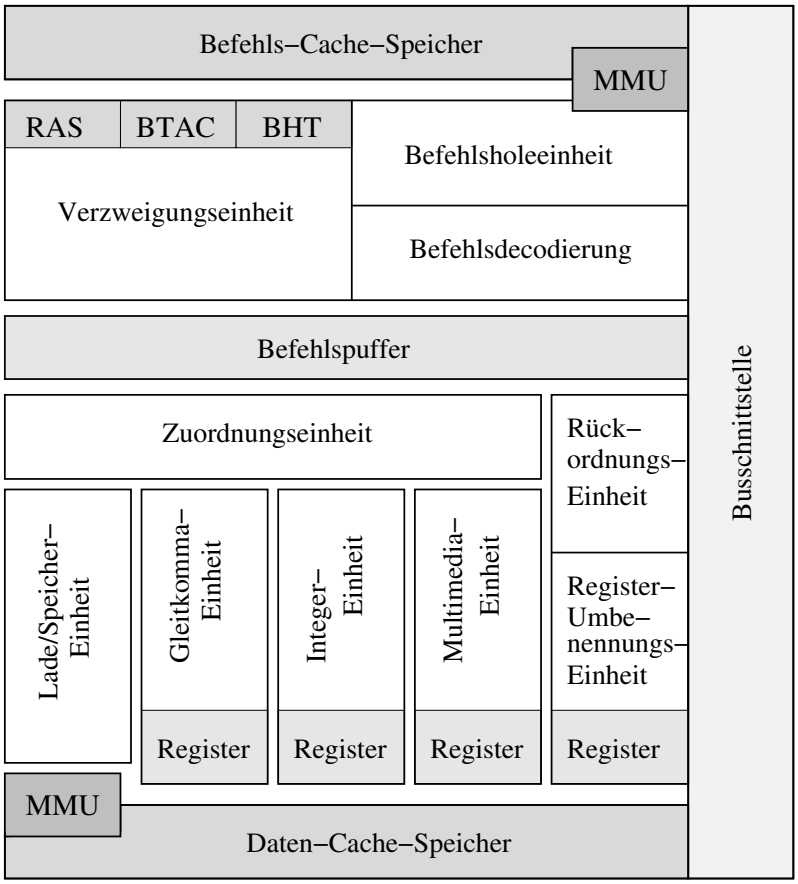


Abbildung 2.22: Komponenten eines superskalaren Prozessors

- Ein Sprungzieladress-Cache (*Branch Target Address Cache* – BTAC) enthält die Adresse des Sprungbefehls sowie dessen Sprungziel und eventuell Voraussagebits einer einfachen Sprungvorhersage. In einer Sprungverlaufstabelle (*Branch History Table* – BHT) werden weitere Informationen über die Sprungausgänge bei der vorherigen Ausführung der Befehle aufgezeichnet. Ferner gibt es zusätzlich einen kleinen sechs bis 18 Einträge fassenden Rücksprungadressstapel (*return address stack* – RAS), der die Rücksprungadressen von Unterprogrammaufrufen speichert. Auf diese drei Tabellen wird während der Befehlsholephase zugegriffen, um die Befehlsadresse des als nächstes zu holenden Befehlsblocks zu bestimmen. Die Verzweigungseinheit überwacht das Aufzeichnen der Sprungvergangenheit und gewährleistet im Falle einer Fehlspekulation die Abänderung der Tabellen sowie das Rückrollen der fälschlicherweise ausgeführten Befehle.
- Eine Lade-/Speichereinheit (*Load/Store Unit*) lädt Werte aus dem Daten-Cache in eines der allgemeinen Register, Multimediaregister oder Gleitkommaregister oder schreibt umgekehrt einen berechneten Wert in den Daten-Cache zurück. Die Berechnung der physikalischen Datenadresse aus der logischen Speicheradresse wird wiederum durch eine eigene Speicherverwaltungseinheit (*Memory Management Unit* – MMU) unterstützt. Im Falle eines Cache-Fehlzugriffs wird der neue Cache-Block automatisch über die Busschnittstelle geladen, während die zugehörige Lade- oder Speicheroperation angehalten wird.
- Mehrere von einander unabhängige Ausführungseinheiten, deren Anzahl und Art stark variieren und jeweils vom spezifischen Prozessor abhängen, führen die in den Befehlen spezifizierten Operationen aus.
  So führen eine oder mehrere Integer-Einheiten (*Integer Units*) die arithmetischen und logischen Befehle auf den allgemeinen Registern aus. In Abhängigkeit der Komplexität der Befehle können Integer-Einheiten einstufig mit einer Latenz von eins und einem Durchsatz von eins sein, oder z.B. aus einer dreistufigen Pipeline mit einer Latenz von drei und einem Durchsatz von

eins bestehen.
Eine oder mehrere Multimediaeinheiten (*Multimedia Units*) führen arithmetische, maskierende, auswählende, umordnende und konvertierende Befehle auf 8-Bit-, 16-Bit- oder 32-Bit-Werten parallel aus. Dabei werden eigenständige 64-, 128- und bald sogar 256 Bit breite Multimediaregister als Quelle oder Ziel der Operanden genutzt.
Eine oder mehrere Gleitkommaeinheiten (*Floating-Point Units*) führen die Gleitkommabefehle aus, die ihre Operanden von den Gleitkommaregistern beziehen. Üblicherweise wird eine 64-Bit-Gleitkommaarithmetik nach IEEE-754-Standard angewandt. Die Gleitkommaeinheit ist meist als Pipeline mit einer Latenz von drei und einem Durchsatz von eins implementiert.

Neben diesen Einheiten hängen am internen Bussystem auch der Befehlspuffer (*Instruction Buffer*) und der sog. Rückordnungspuffer (*Reorder Buffer*). Auf diese Komponenten wird im Folgenden noch eingegangen.

2.6.2 Superskalare Prozessor-Pipeline

Eine superskalare Pipeline erweitert eine einfache RISC-Pipeline derart, dass mehrere Befehle gleichzeitig geholt, decodiert, ausgeführt und deren Ergebnisse in die Register zurückgeschrieben werden. Zu diesem Zweck werden zusätzlich eine Zuordnungs- und eine Rückordnungs- und Rückschreibestufe sowie weitere Puffer zur Entkoppelung der Pipelinestufen benötigt (Abb. 2.23).

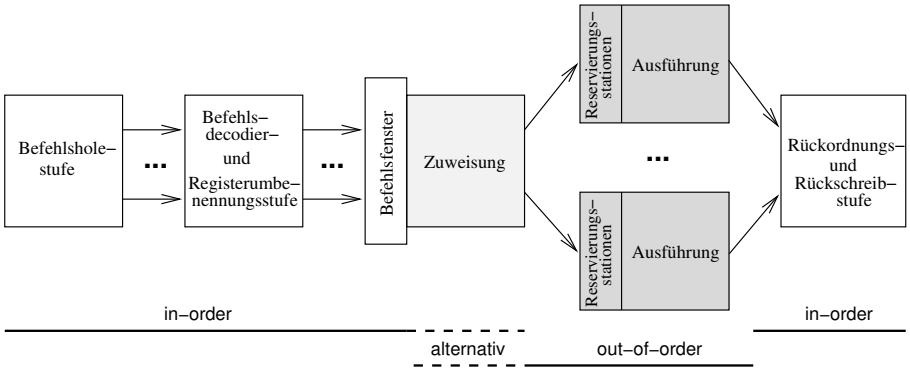


Abbildung 2.23: Superskalare Pipeline

Durch die Möglichkeit, Befehle auch außerhalb ihrer Programmreihenfolge auszuführen, gliedert sich eine superskalare Pipeline in drei Abschnitte:

- Eine *In-order*-Sektion besteht aus der Befehlsholestufe (*Instruction Fetch*) und der Befehlsdecodier- und Registerumbenennungsstufe (*Instruction Decode and Rename*). Die Zuordnung der Befehle an die Ausführungseinheiten geschieht in einem Superskalarprozessor dynamisch, d.h. der sog. *Scheduler* bestimmt die Anzahl der Befehle, die im nächsten Taktzyklus zugewiesen werden. Wird die Zuweisung der Befehle an die Ausführungseinheiten immer in Programmreihenfolge gemacht (*in-order issue*), gehört die Zuordnungsstufe ebenfalls zu dieser *In-order*-Sektion.
- Die *Out-of-order*-Sektion startet – bei out-of-order-Zuweisung – mit der Zuordnungsstufe . Sonst beinhaltet sie nur die gleichzeitig genutzten Ausführungsstufen (*execution*) bis zur Resultatwerterzeugung.
- Die zweite *In-order*-Sektion sorgt für das Rückordnen (*retire*) der Ergebnisse in der ursprünglichen Programmreihenfolge und das Rückschreiben (*write back*) in die Architekturregister.

Die am Beginn der Pipeline stehende Befehlsholestufe (*Instruction Fetch* – IF) lädt gleichzeitig mehrere Befehle aus dem Code-Cache. Typischerweise werden in einem Takt mindestens so viele Befehle geholt wie maximal den Ausführungseinheiten zugewiesen werden können. Welche Befehle geholt

werden, hängt von der Sprungvorhersage ab. Um die Pipeline bei Sprüngen nicht anhalten zu müssen, werden im sog. Sprungszieladress-Cache (BTAC) die Adressen der Sprungbefehle und die dazugehörigen Sprungziele gespeichert. Ein Befehlsholepuffer entkoppelt die Befehlsholestufe von der Decodierstufe.

In der Decodierstufe (*Instruction Decode* – ID) wird in der Regel eine der maximale Zuordnungsbandbreite entsprechende Anzahl von Befehlen decodiert. Die Operanden- und Resultatregister werden umbenannt (*rename*). Dazu werden die in den Befehlen spezifizierten Architekturregister auf die physikalisch vorhandenen Register (bzw. die Umbenennungspufferregister) abgebildet. Danach werden die Befehle in einen Befehlspuffer geschrieben, der oft als Befehlsfenster (*instruction window* oder *instruction pool*) bezeichnet wird. Die Befehle im Befehlsfenster sind durch die Sprungvorhersage frei von Steuerflussabhängigkeiten und durch das Registerumbenennen frei von Namensabhängigkeiten. So müssen nur noch die echten Datenabhängigkeiten beachtet und Strukturkonflikte gelöst werden.

Die Logik für die Zuweisung (*issue*) von Befehlen an die Ausführungseinheiten, der *Scheduler*, überprüft die wartenden Befehle im Befehlsfenster und weist in einem Takt bis zur maximalen Zuordnungsbandbreite Befehle zu. Die Programmreihenfolge der zugewiesenen Befehle wird im Rückordnungspuffer abgelegt. Die Befehle können in der sequenziellen Programmreihenfolge (*in order*) oder außerhalb der Reihenfolge (*out of order*) zugewiesen werden.

Wenn der Befehl die Ausführungseinheit verlassen hat und das Ergebnis für das Forwarding zur Verfügung steht, sagt man, die Befehlsausführung sei vollständig (*complete*). Die Befehlsvervollständigung geschieht außerhalb der Programmreihenfolge. Nach der Vervollständigung können die Befehlsresultate in der Programmreihenfolge gültig gemacht werden (*commited*) wenn:

- die Befehlsausführung vollständig ist,
- die Resultate aller Befehle, die in Programmreihenfolge vor dem Befehl stehen, bereits gültig sind oder im gleichen Taktzyklus gültig gemacht werden,
- keine Unterbrechung vor oder während der Ausführung auftrat und
- der Befehl von keiner Spekulation mehr abhängt.

Während oder nach dem Gültigmachen (*commitment*) werden die Ergebnisse der Befehle durch das Rückschreiben aus den Umbenennungsregistern in die Architekturregistern *dauerhaft* gemacht.

Aufgabe 2.6 *Superpipeline versus Superskalar*
Worin besteht der Unterschied zwischen Superpipeline-Prozessoren und superskalaren Prozessoren?
◇

2.6.3 Befehlsbereitstellung beim Superskalarprozessor

Ein Superskalarprozessor besteht aus einem Ausführungsteil und einem Befehlsbereitstellungsteil, die durch das Befehlsfenster voneinander entkoppelt werden. Der Ausführungsteil wird von der Anzahl der Ausführungseinheiten und der Zuordnungsbandbreite bestimmt. Die Aufgabe des Befehlsbereitstellungsteils ist es, den Ausführungsteil mit genügend Befehlen zu versorgen. Beide Teile des Prozessors müssen aufeinander abgestimmt entworfen werden.

Die erste Stufe einer Prozessor-Pipeline ist immer die Befehlsholestufe oder IF-Stufe (*Instruction Fetch*). In dieser Stufe wird der vom Befehlszählerregister adressierte Befehlsblock aus dem nächst gelegenen Befehlsspeicher geholt. Dies ist bei heutigen Superskalarprozessoren der Code-Cache-Speicher. Der Befehlszähler adressiert den in Ausführungsreihenfolge voraussichtlich als nächstes auszuführenden Befehl. Geholt wird jedoch ein Befehlsblock, der mindestens der Zuordnungsbandbreite des Superskalarprozessors entspricht, da ansonsten die nachfolgenden Pipeline-Stufen des Ausführungsteils des Prozessors nicht mit genügend Befehlen versorgt werden können.

Code-Cache-Speicher

Um in jedem Takt einen solchen Befehlsblock bereitzustellen, ist eine sog. **Harvard-Cache-Architektur** erforderlich. Diese zeichnet sich durch separate Code- und Daten-Cache-Speicher aus, die jeweils eigene Speicherverwaltungseinheiten und separate Zugriffspfade für Befehle und Daten aufweisen. Die Harvard-Cache-Architektur wird auf der Ebene der Primär-Cache-Speicher, die auf dem Prozessor-Chip untergebracht sind, angewendet. Strukturkonflikte beim gleichzeitigen Speicherzugriff der Lade-/Speicher- und der Befehlsholeeinheit können damit für die Primär-Cache-Speicher vermieden werden. Dabei ist die Organisation eines Code-Cache-Speichers einfacher als die eines Daten-Cache-Speichers, da die Befehle aus dem Code-Cache-Speicher nur geladen werden, während auf dem Daten-Cache-Speicher gelesen und geschrieben wird und die sog. Cache-Kohärenz beachtet werden muss. Die nächste Ebene der Speicherhierarchie, der Sekundär-Cache-Speicher, vereint üblicherweise wieder Code und Daten in einem Cache-Speicher.

Behandlung von Steuerflussbefehlen in der Befehlsholestufe

Die größten Probleme für die Befehlsholestufe entstehen durch die Steuerflussbefehle, die das lineare Weiterschalten des Befehlszählers unterbrechen. Diese Unterbrechung des Programmflusses kann durch einen Steuerflussbefehl ausgelöst werden, der zu Anfang oder in der Mitte eines Befehlsblocks steht. In diesen Fällen können alle Befehle, die in diesem Befehlsblock nach dem Steuerflussbefehl stehen, nicht verwendet werden. Nachfolgende Pipeline-Stufen können dann nicht mit der notwendigen Anzahl von Befehlen versorgt werden. Ein ähnliches Problem entsteht, wenn die Befehlszähleradresse durch eine vorangegangene Steuerflussänderung nicht auf den Beginn eines Cache-Blocks im Code-Cache-Speicher zeigt. Man spricht dann von einem nicht ausgerichteten Cache-Zugriff (*non aligned*). Dann können aus einem Befehlsblock ebenfalls weniger Befehle als notwendig den nachfolgenden Pipeline-Stufen zur Verfügung gestellt werden.

Falls die Zieladressen von Steuerflussbefehlen nicht auf die Anfänge der Cache-Blöcke ausgerichtet sind, kann dieses Problem in Hardware durch sog. selbstausrichtende Code-Cache-Speicher gelöst werden. Diese lesen und konkatenieren zwei aufeinanderfolgende Cache-Blöcke innerhalb eines Taktes und geben damit die volle Zuordnungsbreite zur Befehlsholestufe. Eine andere Möglichkeit, die Effizienz der Befehlsbereitstellung zu erhöhen, ist, die Länge des Cache-Blocks über die Länge eines Befehlsblocks hinaus zu erhöhen und ein nicht auf den Beginn des Cache-Blocks ausgerichtetes Befehlsholen zu ermöglichen. Diese Techniken können mit einem Vorabladen der Befehle aus dem Cache-Speicher in einen Befehlspuffer kombiniert werden. Dies entkoppelt die Befehlsholestufe von der Decodierstufe. Schwankungen in der Anzahl der bereitgestellten Befehle können so ausgeglichen und der Pipeline-Durchsatz erhöht werden, doch müssen die nach einem Steuerflussbefehl geladenen Befehle wieder gelöscht werden. Die Lösung dafür ist eine Befehlsladespekulation, die mit einer Sprungspekulation kombiniert wird.

Für zukünftige Superskalarprozessoren mit hohen Zuordnungsbandbreiten wird es notwendig sein, pro Takt Befehle aus mehreren, nicht aufeinander folgenden Code-Cache-Blöcken bereitzustellen. Dies trifft insbesondere dann zu, wenn mehrere Sprungspekulationen pro Takt erfolgen, da in großen Befehlsblöcken mehrere Sprungbefehle enthalten sein können. Auch um Techniken wie die spekulative beidseitige Ausführung nach einem bedingten Sprungbefehl zu zu unterstützen, erscheint eine solche Befehlsbereitstellung notwendig.

Trace Cache

Das Hauptproblem der effiziente Befehlsbereitstellung entsteht durch die vielen Sprungbefehle in einem konventionellen sequenziellen Befehlsstrom. Bei einer Zuordnungsbandbreite von acht oder höher sind somit in der Regel mehrere Sprungvorhersagen pro Takt nötig, und dazu müssen pro Takt dann noch Befehle von mehreren Stellen des Code-Cache-Speichers geladen werden. Als Lösung dafür erscheint der sog. Trace Cache geeignet, der bei Voranschreiten einer Programmausführung den aus dem Code-Cache geladenen Befehlsstrom in Befehlsfolgen (*traces*) fester Länge abspeichert. Ein Trace ist dabei eine Folge von Befehlen, die an einer beliebigen Stelle des dynamischen Befehlsab-

laufs starten und sich über mehrere Sprungbefehle hinweg erstrecken kann. Wird die entsprechende Befehlsfolge erneut ausgeführt, so werden die Befehle nicht mehr dem Code-Cache, sondern dem Trace Cache entnommen.

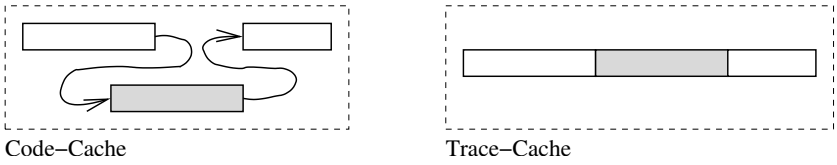


Abbildung 2.24: Code-Cache- und Trace-Cache-Speicher

Ein Trace-Cache enthält also die dynamischen erzeugten Befehlsablauffolgen, während der Code-Cache-Speicher die statischen, vom Compiler erzeugten, Befehlsfolgen speichert. Abbildung 2.24 zeigt, wie die Befehle einer Ablauffolge stückweise auf den Code-Cache-Speicher verteilt sein können, während dieselbe Ablauffolge im Trace Cache in einem Trace-Cache-Block hintereinander abgespeichert werden kann.

2.6.4 Sprungvorhersage und spekulative Ausführung

Eine effiziente Behandlung von Sprungbefehlen ist für alle Mikroprozessoren enorm wichtig. In den einzelnen Stufen eines Superskalarprozessors befinden sich viele Befehle in verschiedenen Ausführungszuständen. Der Ausführungsteil arbeitet am besten, wenn der Zuordnungseinheit sehr viele Befehle zur Auswahl stehen, d.h. das Befehlsfenster groß ist. Im Befehlsstrom eines typischen Anwenderprogramms ist im Mittel etwa jeder fünfte bis siebte Befehl ein bedingter Sprungbefehl, der den kontinuierlichen Befehlsfluss durch die Pipeline unterbrechen kann. Unter Berücksichtigung der spekulativen Ausführung von Befehlen befinden sich bei heutigen Superskalarprozessoren in der Regel sogar *mehrere* Sprungbefehle gleichzeitig in der Pipeline. Für eine hohe Leistung bei der Sprungbehandlung sind folgende Voraussetzung unbedingt zu erfüllen:

- Die Sprungrichtung muss möglichst rasch festgestellt werden.
- Die Sprungzieladresse muss in einem Sprungzieladress-Cache (BTAC) nach ihrem erstmaligen Berechnen gespeichert und bei Bedarf (falls die Sprungvorhersage einen genommenen Sprung vorhersagt) sofort in den Befehlszähler geladen werden, so dass die Befehle ohne Verzögerung ab der Sprungzieladresse spekulativ in die Pipeline geladen werden können.
- Die Sprungvorhersage (*branch prediction*) sollte sich durch eine sehr hohe Genauigkeit und die Möglichkeit, Befehle spekulativ auszuführen, auszeichnen.
- Häufig muss ein zweiter Sprung vorhergesagt werden, obwohl der Test der Bedingung des vorhergehenden Sprungs noch nicht ausgeführt ist. Der Prozessor muss daher mehrere Ebenen der Spekulation verwalten können.
- Wenn ein Sprung falsch vorhergesagt wurde, ist ein schneller Rückrollmechanismus mit geringem Fehlspekulationsaufwand (*misprediction penalty*) wichtig.

Die Leistungsfähigkeit einer Sprungvorhersage hängt zum einen von der Genauigkeit der Vorhersage und zum anderen von den Kosten für das Rückrollen bei einer Fehlspekulation ab. Die Genauigkeit kann durch eine qualitativ bessere Sprungvorhersagetechnik erhöht werden. Gegenüber den in Abschnitt 2.5.4 beschriebenen statischen Sprungvorhersagetechniken haben sich bei heutigen Superskalarprozessoren die wesentlich genaueren dynamischen Sprungvorhersagetechniken durchgesetzt. Ebenso führt der Einsatz größerer Informationstabellen über die bisherigen Sprungverläufe bei diesen Techniken auch zu weniger Fehlspekulationen. Mit den „bisherigen Sprungverläufen“ oder der „Historie“ der Sprünge sind die Sprungrichtungen der bedingten Sprünge gemeint, die vom Programmstart bis zum augenblicklichen Ausführungszustand in den Sprungverlaufstabellen gesammelt worden sind.

Die Kosten für das Rückrollen einer Fehlspekulation liegen selbst bei einfachen RISC-Pipelines meist bei zwei oder mehr Takten. Diese Kosten hängen jedoch von vielen organisatorischen Faktoren einer Pipeline ab. Eine vielstufige Pipeline verursacht mehr Kosten als eine kurze Pipeline.

Eine andere Technik, mit Sprüngen umzugehen, ist die sog. Prädikation (*predication*), die prädikative (*predicated* oder *conditional*) Befehle benutzt, um einen bedingten Sprungbefehl durch Ausführung beider Programmpfade nach der Bedingung zu ersetzen und spätestens bei der Rückordnung die fälschlicherweise ausgeführten Befehle zu verwerfen. Die Prädikationstechnik ist besonders effektiv, wenn der Sprungausgang völlig irregulär wechselt und damit nicht vorhersagbar ist. In diesem Fall spart man sich die Kosten für das Rückrollen falsch vorhergesagter Sprungpfade, die meist höher sind als die unnötige Ausführung einer der beiden Programmpfade bei der Prädikation. Bei heutigen Superskalarprozessoren ist diese Technik jedoch wegen des dazu benötigten Prädikationsbits im Befehl kaum einsetzbar.

2.6.5 Dynamische Sprungvorhersagetechniken

Bei den statischen Sprungvorhersagetechniken kann sich die Vorhersage im Verlauf einer Programmausführung nicht ändern. Bei der dynamischen Sprungvorhersagetechnik hingegen wird die Entscheidung über die Spekulationsrichtung eines Sprungs in Abhängigkeit vom bisherigen Programmablauf getroffen. Die bisherigen Sprungverläufe werden in Sprungverlaufstabellen gesammelt und auf der Grundlage der Tabelleneinträge werden aktuelle Sprungvorhersagen getroffen. Bei Fehlspekulationen werden die Tabellen per Hardware aktualisiert.

Wie bei allen Prädiktoren wird natürlich auch hier zusätzlich die Sprungzieladresse benötigt, was einen Sprungzieladress-Cache (*Branch-Target Address Cache* – BTAC) Cache-Speicher, der für die aufgetretenen Sprünge die folgenden Parameter speichert:

- **Feld 1**: die Adresse (*branch address*) eines Sprungbefehls, der in der Vergangenheit ausgeführt wurde,
- **Feld 2**: die Zieladresse (*target address*) dieses Sprungs,
- **Feld 3**: Vorhersagebits (*prediction bits*), die steuern, ob im Falle eines bedingten Sprungs dieser als „genommen“ oder „nicht genommen“ vorhergesagt wird, oder ob es sich um einen unbedingten Sprung handelt.

Feld 1 **Branch address**	Feld 2 **Target address**	Feld 3 Prediction Bits
$34A1	$1974	T
$7C4D	$B12A	NT
...	...	...

Der Sprungzieladress-Cache funktioniert wie folgt: Die Befehlsholestufe vergleicht den Inhalt des Befehlszählerregisters (PC) mit den Adressen der Sprungbefehle im Sprungzieladress-Cache (Feld 1). Falls ein passender Eintrag gefunden wird und der geholte Befehl ein bedingter Sprung ist, wird eine Vorhersage, ob der Sprung genommen wird oder nicht, auf Basis der Vorhersagebits in Feld 3 getroffen. Wird der Sprung als „ nicht genommen“ (NT) vorhergesagt, so wird im Programmverlauf der nächste Befehl geholt. Wird er als „genommen“ (T) vorhergesagt, so wird die zugehörige Sprungzieladresse (Feld 2) in den PC übertragen und mit ihrer Hilfe der Zielbefehl geholt. Falls für den betrachteten Sprung kein Eintrag im BTAC vorhanden ist, so wird ein Eintrag erzeugt, sobald die Sprungadresse berechnet ist, und dafür eventuell ein anderer Eintrag überschrieben.

Der Sprungzieladress-Cache speichert die Adressen von bedingten wie auch von unbedingten Sprüngen. Bei bedingten Sprüngen handelt es sich daher immer um eine spekulierte Sprungzieladresse, wohingegen bei unbedingten Sprüngen die Adresse fest ist. Natürlich kann die in der Tabelle eingetragene Vorhersage sich als falsch herausstellen. Dann muss der BTAC die Vorhersagebits korrigieren, sobald die tatsächliche Sprungrichtung bekannt ist. Zu einer Fehlspekulation kann es aus drei Gründen kommen:

- Zunächst erfolgt nach dem Programmstart eine Warmlaufphase, während derer die ersten Informationen über die Sprungverläufe gesammelt werden. In dieser Phase wird die Qualität der dynamischen Sprungvorhersage zunehmend genauer.
- Der Sprung wird falsch vorhergesagt, da der Sprung eine unvorhergesehene Richtung nimmt. Beispielsweise führt der Austritt aus einer Schleife nach vielen durchlaufenen Iterationen immer zu einer falschen Vorhersage.
- Genau genommen besteht die Sprungadresse in Feld 1 wegen der im Speicherplatz beschränkten Sprungverlaufstabelle nicht aus der vollständigen Sprungadresse, sondern nur aus einem Teil, der als Index bezeichnet wird. Durch diese Indizierung kann, wenn zwei Sprungbefehle in dem betreffenden Adressteil dasselbe Bitmuster aufweisen, die Verlaufsgeschichte eines *anderen* Sprungbefehls auf den gleichen Eintrag mit erfasst werden. Eine solche **Wechselwirkung** oder **Interferenz** (*branch interference*, *aliasing*) zwischen zwei Sprungbefehlen führt besonders bei kleinen Tabellen häufig zu einer Fehlvorhersage. Die Anzahl der Interferenzen lässt sich verringern und damit die Spekulationsgenauigkeit signifikant verbessern, wenn der BTAC und damit die Tabelle vergrößert wird. Die Interferenzen würden ganz vermieden, wenn jeder Sprungbefehl einen eigenen Eintrag in der Tabelle erhält, dies ist jedoch für beliebig große Programme wegen des beschränkten Platzes auf dem Prozessor-Chip nicht möglich.

Ein-Bit-Prädiktoren

Die einfachste dynamische Sprungvorhersagetechnik ist der **Ein-Bit-Prädiktor**, der für jeden Sprungbefehl die zwei Zustände „genommen“ (T, *Taken*) oder „nicht genommen“ (NT, *Not Taken*) in einem Bit speichert. Diese Zustände geben die Vorhersage für die nächste Ausführung des Sprungbefehls an. Die Abb. 2.25 zeigt das Vorhersageverhalten des Ein-Bit-Prädiktors in einem sog. Zustandsdiagramm. Befindet sich der Prädiktor im Vorhersagezustand T und der nächste Sprung wird nicht genommen (nt), dann wechselt der Prädiktor in den Zustand NT über und umgekehrt.

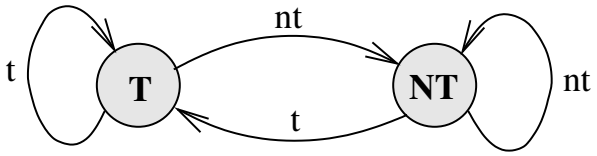


Abbildung 2.25: Verhalten des Ein-Bit-Prädiktors

Der Ein-Bit-Prädiktor wird in der Befehlsholestufe implementiert und benötigt eine Sprungverlaufstabelle (*Branch History Table*) BHT, die mit einem Teil der Sprungbefehlsadresse adressiert wird und pro Eintrag ein Bit enthält, das die Sprungvorhersage bestimmt. Ist dieses Bit gesetzt, wird der Sprung als „genommen“ vorhergesagt, und wenn es gelöscht ist, als „nicht genommen“. Im Falle einer Fehlspekulation wird das Bit invertiert und damit die Richtung der Vorhersage umgekehrt. Diese BHT kann in Form des oben beschriebenen BTAC implementiert werden.

Der Ein-Bit-Prädiktor sagt jeden Sprung am Ende einer Schleifeniteration richtig voraus, solange die Schleife iteriert wird. Der Prädiktor des Sprungbefehls steht auf „genommen“ (T). Wird die Schleife verlassen, so ergibt sich eine falsche Vorhersage und damit eine Invertierung des Vorhersagebits des Sprungbefehls, auf „nicht genommen“ (NT). Damit kommt es in geschachtelten Schleifen jedoch in der inneren Schleife zu einer weiteren falschen Vorhersage. Beim Wiedereintritt in die innere Schleife steht am Ende der ersten Iteration der inneren Schleife die Vorhersage noch auf NT. Die zweite Iteration wird damit falsch vorhergesagt, denn erst ab dieser zweiten Iteration steht der Prädiktor des

Sprungbefehls wieder auf „genommen“ (T). Mit einem Zwei-Bit-Prädiktor wird bei geschachtelten Schleifen eine dieser zwei Fehlvorhersagen vermieden.

Zwei-Bit-Prädiktoren

Beim **Zwei-Bit-Prädiktor** werden für die Zustandscodierungen der bedingten Sprungbefehle zwei Bits pro Eintrag in der Sprungverlaufstabelle verwendet. Damit ergeben sich die vier Zustände „sicher genommen“ (ST, *strongly taken*), „vielleicht genommen“ (WT, *weakly taken*), „vielleicht nicht genommen“ (WNT, *weakly not taken*) und „sicher nicht genommen“ (SNT, *strongly not taken*). Befindet sich ein Sprungbefehl in einem „sicheren“ Vorhersagezustand, so sind zwei aufeinander folgende Fehlspekulationen nötig, um die Vorhersagerichtung umzudrehen. Damit kommt es bei inneren Schleifen einer Schleifenschachtelung nur beim Austritt aus der Schleife zu einer falschen Vorhersage. Es gibt zwei Ausprägungen des Zwei-Bit-Prädiktors, die sich in der Definition der Zustandsübergänge unterscheiden. Das Schema mit einem **Sättigungszähler** (*saturation up-down counter*) ist in Abbildung 2.26 dargestellt. Die zweite, **Hysteresezähler** genannte Variante verdeutlicht Abbildung 2.27.

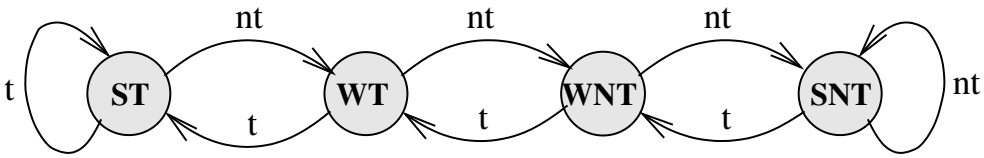


Abbildung 2.26: Verhalten des Zwei-Bit-Prädiktors mit Sättigungszähler

Der Zwei-Bit-Prädiktor mit Sättigungszähler erhöht jedes Mal, wenn der Sprung genommen wurde, den Zähler und erniedrigt ihn, falls er nicht genommenen wurde. Durch die Sättigungsarithmetik fällt der Zähler nie unter Null (00) oder wird größer als drei (11). Das höchstwertige Bit gibt die Richtung der Vorhersage an.

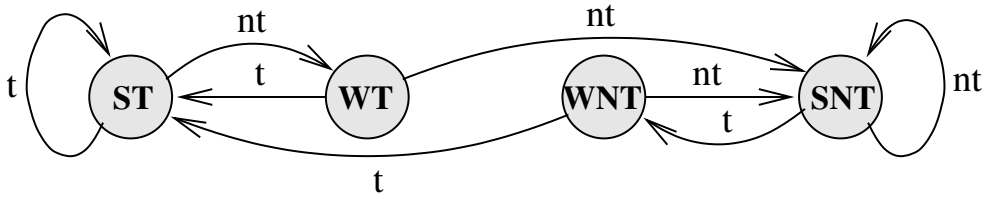


Abbildung 2.27: Verhalten des Zwei-Bit-Prädiktors mit Hysterese

Die zweite Variante, die Hysteresemethode, unterscheidet sich von der des Sättigungszählers dadurch, dass direkt von einem „unsicheren“ (*weakly*) Zustand in den „sicheren“ (*strongly*) Zustand der entgegengesetzten Richtung gewechselt wird. Damit kommt man von einem sicheren Zustand in den anderen sicheren Zustand durch zwei Fehlspekulationen.

Die Technik der Zwei-Bit-Prädiktoren lässt sich leicht auf *n* Bits erweitern. Es zeigte sich jedoch, dass dabei so gut wie keine Verbesserungen mehr erzielbar sind. Ein Zwei-Bit-Prädiktor kann ebenfalls in einem BTAC implementiert werden, wobei jeder Eintrag um zwei Vorhersagebits erweitert wird.

Im Allgemeinen arbeitet ein Zwei-Bit-Prädiktor sehr gut bei numerischen Programmen, die Schleifen mit vielen Iterationen enthalten. Die Anzahl der Fehlspekulationen steigt jedoch bei allgemeinen, ganzzahlintensiven Programmen stark an, da die Schleifen häufig nur wenige Iterationen aufweisen, dafür aber viele IF-THEN- und IF-THEN-ELSE-Konstrukte vorkommen. Aufeinander folgende Sprünge sind oft in der Art des folgenden Programmkonstruktion voneinander abhängig (*correlated*):

```
      if (d == 0)   ⇒ Sprung nach M1
      d = 1
M1:   if (d == 0)   ⇒ Sprung nach M2
```

Wie man sieht, wird der zweite Sprung immer dann genommen, wenn der erste genommen wurde, und der zweite Sprung wird immer dann nicht genommen, wenn der erste nicht genommen wurde. Dies

ist eine Korrelationsinformation, die von Ein- und Zwei-Bit-Prädiktoren nicht genutzt werden kann. Bei diesem Programmkonstrukt kann es passieren, dass von einem Ein-Bit- oder Zwei-Bit-Prädiktor jeder Sprung falsch vorhergesagt wird. Wir betrachten eine mögliche Übersetzung der IF-THEN-Konstrukte in Maschinensprache (die Variable d sei in Register R1 abgelegt):

```
        ⋮
        BNEZ   R1,L1      Sprung nach L1 falls d≠0
        ADDI   R1,R0,#1   R1=R0+1 (d=1)
L1:     SUBI   R3,R1,#1   R3=R1-1
        BNEZ   R3,L2      Sprung nach L2 falls d≠1
        ⋮
L2:     ⋮
        ⋮
```

Angenommen, der Wert von d alterniert zwischen 0 und 2. Damit ergibt sich eine Folge von nt-t-nt-t-nt-t bezüglich beider Sprungbefehle S1 und S2. Die folgende Tabelle verdeutlicht diesen Sachverhalt.

Anfangswert für d bei Iterationsbeginn	d=0 ?	Sprung-richtung für S1	d vor S2	d=1 ?	Sprung-richtung für S2
0	Ja	nt	1	ja	nt
2	Nein	t	2	nein	t

Wenn die Sprünge S1 und S2 mit einem Ein-Bit-Prädiktor mit Anfangszustand „genommen“ vorhergesagt werden, so werden beide Sprünge ständig falsch vorhergesagt. Das gleiche Verhalten zeigt der Zwei-Bit-Prädiktor mit Sättigungszähler aus Abbildung 2.26 mit Anfangszustand „vielleicht genommen“. Der Zwei-Bit-Prädiktor mit Hysteresezähler aus Abbildung 2.27 verspekuliert sich bei einer Initialisierung von „vielleicht genommen“ nur bei jedem zweiten Durchlauf der Sprünge S1 und S2. Der Grund liegt darin, dass die Zwei-Bit-Prädiktoren für eine Vorhersage immer nur den Verlauf des Sprungs selbst in Betracht ziehen. Die Beziehungen zwischen verschiedenen Sprüngen werden nicht berücksichtigt.

Komplexere Vorhersageverfahren können den Zusammenhang zwischen beiden Sprüngen erkennen und nutzen. Sie treffen nur in der ersten Iteration eine falsche Vorhersage. Beispiele für diese Verfahren sind die *Korrelationsprädikatoren*, die neben der eigenen Vergangenheit eines Sprungbefehls auch die Historie benachbarter, im Programmlauf vorhergegangener Sprünge berücksichtigen. Mit der Betrachtung dieser Korrelationsprädikatoren wollen wir diesen Abschnitt abschließen. Auf die noch komplexeren *zweistufig adaptiven Prädiktoren*, die ihre Entscheidungen aus den Einträgen in zwei Tabellenebenen ziehen, wobei der Eintrag in der ersten Tabelle dazu dient, die Vorhersagebits auf der zweiten Tabellenebene zu selektieren, wollen wir nicht mehr eingehen.

Aufgabe 2.7 *Vorhersage bei einem Vor-/Rückwärtszähler*
Man betrachte das obige Programmbeispiel mit dem in der Tabelle gegebenen Programmablauf. Vervollständige die Zustände der Prädiktortabelleneinträge für die beiden Sprünge bei Verwendung der folgenden Prädiktoren:

a) Ein-Bit-Prädiktor (Initialzustand „predict taken“)

Sprung	Initialzustand	d=0	d=2	d=0
S1	T			
S2	T			

b) Zwei-Bit-Prädiktor mit Sättigungszähler (Initialzustand: „predict weakly taken“)

Sprung	Initialzustand	d=0	d=2	d=0
S1	WT			
S2	WT			

c) Zwei-Bit-Prädiktor mit Hysteresezähler (Initialzustand: „predict weakly taken“)

Sprung	Initialzustand	d=0	d=2	d=0
S1	WT			
S2	WT			

◇

Aufgabe 2.8 *Vorhersage bei einem Vor-/Rückwärtszähler*
Das folgende Programm besteht aus zwei verschachtelten Schleifen. Das Register R4 sei mit $m > 0$ und das Register R3 mit $n > 0$ vorbelegt. Weiterhin sei das Register R1 mit 1 belegt.

```
LOOP1:  SLE   R10,R3,R0     R3 ≤ R0? Status in R10
        BNEZ  R10,ENDE1     Sprung nach ENDE1 falls R10≠0
LOOP2:  SLE   R11,R4,R0     R4 ≤ R0? Status in R11
        BNEZ  R11,ENDE2     Sprung nach ENDE2 falls R11≠ 0
        ⋮
        SUB   R4,R4,R1      R4=R4-R1
        JMP   LOOP2         Springe nach LOOP2
ENDE2:  SUB   R3,R3,R1      R3=R3-R1
        JMP   LOOP1         Springe nach LOOP1
ENDE1:  ⋮
```

Es sollen verschiedene Sprungvorhersagetechniken verglichen werden, aber dabei nur bedingte Sprünge betrachtet werden. Wie viele richtige und falsche Vorhersagen gibt es bei:

a) statischer Sprungvorhersage mit „Always taken“-Technik,

b) dynamischer Sprungvorhersage: Ein-Bit-Prädiktor (Initialzustand „predict taken“),

c) dynamischer Sprungvorhersage: Zwei-Bit-Prädiktor mit Sättigungszähler (Initialzustand: „predict weakly taken“)?

Begründen Sie Ihre Ergebnisse durch Aufstellen einer Tabelle. ◇

Korrelationsprädiktoren

Die Zwei-Bit-Prädiktoren ziehen für eine Vorhersage immer nur den Verlauf des Sprungs selbst in Betracht. Die Beziehungen zwischen verschiedenen Sprüngen werden nicht berücksichtigt. Untersuchungen haben jedoch gezeigt, dass bei Auswertung dieser Beziehungen eine bessere Sprungvorhersage durchgeführt werden kann. Die **Korrelationsprädiktoren** (*correlation-based predictors*) berücksichtigen neben der eigenen Vergangenheit eines Sprungbefehls auch die Historie benachbarter, im Programmlauf vorhergegangener Sprünge. Korrelationsprädiktoren erzielen gewöhnlich bei ganzzahlintensiven Programmen eine höhere Trefferrate als die Zwei-Bit-Prädiktoren. Sie benötigen dabei nur wenig mehr Hardware.

Ein Korrelationsprädiktor wird als (m,n)**-Prädiktor** bezeichnet, wenn er das Verhalten der letzten m Sprünge für die Auswahl aus 2^m Prädiktoren nutzt, wobei jeder Prädiktor einen n-Bit-Prädiktor für einen einzelnen Sprung darstellt. Die globale Vergangenheit der letzten m Sprünge kann in einem m-Bit-Schieberegister gespeichert werden, das als **Sprungverlaufsregister** oder **BHR** (*Branch History Register*) bezeichnet wird. Der Zustand der Bits zeigt dann an, ob die letzten Sprünge genommen wurden oder nicht. Nach jedem ausgeführten Sprungbefehl wird der BHR-Inhalt um ein Bit nach links verschoben und der Sprungausgang an der freigewordenen Bitposition eingefügt (1 für genommene

und 0 für nicht genommene Sprünge). Der Inhalt des BHR wird als Adresse (*index*) benutzt, um eine **Sprungverlaufstabelle** oder **PHT** (*Pattern History Table*) zu selektieren. Für ein m Bit breites BHR werden somit 2^m PHTs benötigt, wobei jede PHT eine n-Bit-Prädiktortabelle darstellt. Der PHT-Eintrag selbst wird wie beim Zwei-Bit-Prädiktor über einen Teil der Sprungbefehlsadresse selektiert.

Ein (1,1)-Prädiktor wählt in Abhängigkeit vom Verhalten des letzten Sprungs aus einem Paar von Ein-Bit-Prädiktoren aus. Die Bezeichnung (2,2)-Prädiktor weist auf ein zwei Bit breites BHR hin, womit aus vier Sprungverlaufstabellen (PHTs) von Zwei-Bit-Prädiktoren ausgewählt wird. Üblicherweise werden in der PHT Zwei-Bit-Prädiktoren eingesetzt. Ein Zwei-Bit-Prädiktor wird in dieser Notation als (0,2)-Prädiktor bezeichnet.

Beispiel 2.2 *Sprungvorhersage mit einem (2,2)-Korrelationsprädiktor*
An folgendem in Maschinensprache dargestellten IF-THEN-Konstrukt soll das Verhalten eines (2,2)-Korrelationsprädiktors mit Sättigungszähler zur Sprungvorhersage untersucht werden.

```
      ⋮
      BNEZ   R1,L1      Sprung nach L1 falls R1 ≠ 0
      SUBI   R2,R2,#2   R2=R2 - 2
L1:   ADDI   R1,R1,#2   R1=R1+2
      XORI   R3,R1,#2   R3=R1⊕2
      BNEZ   R3,L2      Sprung nach L2 falls R3 ≠ 0
      ⋮
L2:   ⋮
      ⋮
```

Bei dem Befehl XORI handelt es sich um einen Bitbefehl, der die einzelnen Bitstellen der beiden Operanden antivalent verknüpft (Exclusiv-ODER).
Es sei angenommen, dass der Wert der Variablen, die im Register R1 abgelegt ist, beim Eintritt in diese Programmsequenz zwischen 0 und 1 alterniert.
Die folgende Tabelle zeigt einen für die zwei Sprünge S1 und S2 relevanten Ausschnitt aus der initialen Sprungverlaufstabelle PHT des (2,2)-Korrelationsprädiktors, in der alle Elemente auf den Initialzustand "predict weakly taken" (WT) gesetzt sind. Das zwei Bit breite Sprungverlaufsregister (BHR) ist auf den Initialzustand $\boxed{1\ 1}$ gesetzt.

BHR $\boxed{1\ 1}$

	00	01	10	11
S1	WT	WT	WT	WT
S2	WT	WT	WT	WT

a) Zunächst ist das Sprungverhalten der beiden Sprünge S1 und S2 in den ersten drei Durchläufen zu ermitteln, wobei initial R1=0 gelte.

b) Danach sollen die Verläufe von BHR und PHT für die ersten 6 Sprünge angegeben werden, wobei wie unter a) initial R1=0 gelte.

c) Wird sich bei einem Korrelationsprädiktor allgemein in jedem Fall nach einer endlichen Zahl von Durchläufen ein stationärer Zustand einstellen?

Lösung

a) Darstellung der Sprungverläufe von S1 und S2

Der Wert der Variablen, die im Register R1 abgelegt ist, alterniert beim Eintritt in die Programmsequenz zwischen 0 und 1. Der Befehl SUBI ist für die Aufgabenstellung irrelevant, da das Zielregister R2 für das Sprungverhalten keine Bedeutung hat.
Durch den Befehl ADDI wird das R1 nach dem zweiten Sprung um 2 erhöht, der Wert alterniert also

vor dem Befehl XORI zwischen 2 und 3. Der Befehl XORI führt als Bitbefehl ein Exklusiv-Oder zwischen dem Inhalt von R1 und dem Wert 2 aus, das Zielregister R3 alterniert also vor der zweiten Schleife zwischen 0 und 1.

	S1	S2	Begründung
1. Durchlauf	nt	nt	R1=0 $\Rightarrow$ S1=nt, mit R1=R1+2=2 und R3=R1$\oplus$2=2$\oplus$2=0 ist R3=0 $\Rightarrow$ S2=nt
2. Durchlauf	t	t	R1=1$\neq$0 $\Rightarrow$ S1=t, mit R1=R1+2=3 und R3=R1$\oplus$2=3$\oplus$2=1 ist R3=1$\neq$0 $\Rightarrow$ S2=t
3. Durchlauf	nt	nt	R1=0 $\Rightarrow$ S1=nt, mit R1=R1+2=2 und R3=R1$\oplus$2=2$\oplus$2=0 ist R3=0 $\Rightarrow$ S2=nt

Beim ersten Durchlauf ist R1=0, der Sprung S1 wird nicht genommen (S1=nt).
Mit R1=R1+2=2 und R3=R1$\oplus$2=2$\oplus$2=0 ist R3=0, der Sprung S2 wird nicht genommen.

Beim zweiten Durchlauf ist R1=1$\neq$0, der Sprung S1 wird genommen (S1=t).
Mit R1=R1+2=3 und R3=R1$\oplus$2=3$\oplus$2=1 ist R3=3$\neq$0, der Sprung S2 wird genommen (S2=t).

Beim dritten Durchlauf ist R1=0, der Sprung S1 wird nicht genommen (S1=nt).
Mit R1=R1+2=2 und R3=R1$\oplus$2=2$\oplus$2=0 ist R3=0, der Sprung S2 wird nicht genommen.

Damit ergibt sich für den Sprungbefehl S1 als auch für den Sprungbefehl S2 die gleiche Sprungfolge: nt-t-nt-t-nt-t-nt-t... Der zweite Sprung wird genommen wenn der erste genommen wurde und wird nicht genommen wenn der erste nicht genommen wurde.

b) Verläufe von BHR und PHT für 5 Sprünge

Wie oben beschrieben nutzt ein (2,2)-Korrelationsprädiktor das Verhalten der letzten zwei Sprünge für die Vorhersage des aktuellen Sprungverhaltens. Für diese Vorhersage trifft er anhand des BHR, das die Vergangenheit der letzten zwei Sprünge speichert, eine Auswahl aus den vier 2-Bit-Prädiktoren 00, 01, 10 und 11, welcher dann den aktuellen Vorhersagewert liefert.

1. Durchlauf:
R1=0, S1=nt, BHR $\boxed{1\ 1}$ $\Rightarrow$ VHS-S1=WT $\Rightarrow$ falsch, $\Rightarrow$ S1(11)=WNT, BHR $\boxed{1\ 0}$

BHR und PHT nach dem 1. Sprung

BHR $\boxed{1\ 0}$

	00	01	10	11
S1	WT	WT	WT	WNT
S2	WT	WT	WT	WT

R3=0, S2=nt, BHR $\boxed{1\ 0}$ $\Rightarrow$ VHS-S2=WT $\Rightarrow$ falsch, $\Rightarrow$ S2(10)=WNT, BHR $\boxed{0\ 0}$

BHR und PHT nach dem 2. Sprung

BHR $\boxed{0\ 0}$

	00	01	10	11
S1	WT	WT	WT	WNT
S2	WT	WT	WNT	WT

2. Durchlauf:
R1=1, S1=t, BHR $\boxed{0\ 0}$ $\Rightarrow$ VHS-S1=WT $\Rightarrow$ richtig, $\Rightarrow$ S1(00)=ST, BHR $\boxed{0\ 1}$

BHR und PHT nach dem 3. Sprung

BHR $\boxed{0\ 1}$

	00	01	10	11
S1	ST	WT	WT	WNT
S2	WT	WT	WNT	WT

R3=1, S2=t, BHR $\boxed{0\ 1}$ $\Rightarrow$ VHS-S2=WT $\Rightarrow$ richtig, $\Rightarrow$ S2(01)=ST, BHR $\boxed{1\ 1}$

BHR und PHT nach dem 4. Sprung

BHR [1 1]

	00	01	10	11
S1	ST	WT	WT	WNT
S2	WT	ST	WNT	WT

3. Durchlauf:
R1=0, S1=nt, BHR [1 1] $\Rightarrow$ VHS-S1=WNT $\Rightarrow$ richtig, $\Rightarrow$ S1(11)=SNT, BHR [1 0]

BHR und PHT nach dem 5. Sprung

BHR [1 0]

	00	01	10	11
S1	ST	WT	WT	SNT
S2	WT	ST	WNT	WT

R3=0, S2=nt, BHR [1 0] $\Rightarrow$ VHS-S2=WNT $\Rightarrow$ richtig, $\Rightarrow$ S2(10)=SNT, BHR [0 0]

BHR und PHT nach dem 6. Sprung

BHR [0 0]

	00	01	10	11
S1	ST	WT	WT	SNT
S2	WT	ST	SNT	WT

c) stationärer Zustand

Ein stationärer Zustand wird sich einstellen wenn aufgrund vergangener Sprünge und der Korrelation zwischen einzelnen Sprüngen eine Vorhersage möglich ist. Dabei kann die Zahl der Sprünge sehr groß sein, bis sich ein bestimmter Zyklus wiederholt, ein Zyklus ist aber die Voraussetzung für Stationarität.
Bei einem völlig irregulären Sprungverlauf hingegen kann sich kein stationärer Zustand einstellen. □

Aufgabe 2.9 *Sprungvorhersage mit einem (1,1)-Korrelationsprädiktor*
Wir betrachten erneut das Programmbeispiel aus Aufgabe 2.7 mit dem in der Tabelle gegebenen Programmablauf.

```
        ⋮
        BNEZ   R1,L1        Sprung nach L1 falls d≠0
        ADDI   R1,R0,#1     R1=R0+1 (d = 1)
L1:     SUBI   R3,R1,#1     R3=R1-1
        BNEZ   R3,L2        Sprung nach L2 falls d≠1
        ⋮
L2:     ⋮
        ⋮
```

Anfangswert für d bei Iterationsbeginn	d=0 ?	Sprung-richtung für S1	d vor S2	d=1 ?	Sprung-richtung für S2
0	Ja	nt	1	ja	nt
2	Nein	t	2	nein	t

Der Initialzustand des $(1,1)$-Prädiktors sei in allen Elementen „predict taken“ (T), das BHR sei auf den Initialzustand [1] gesetzt.

BHR [1]

	0	1
S1	T	T
S2	T	T

Vervollständigen Sie die Zustände der Prädiktortabelleneinträge für die beiden Sprünge bei Verwendung eines (1,1)-Korrelationsprädiktors für die Durchläufe mit $d = 0$ und $d = 2$.

Sprung	Initialzustand	d=0	d=2
S1	T		
S2	T		

BHR [1]

	0	1
S1		
S2		

◇

2.7 Moderne PC-Prozessoren (H. Bähring)

Ein wichtiges Ziel bei der Entwicklung der x86-kompatiblen Prozessoren – kurz x86-Prozessoren –, die heute fast ausschließlich in PCs verwendet werden, war die Fähigkeit, in Maschinensprache gegebene Software für die x86-Familie ohne erneute Übersetzung (Recompilation) auch auf neueren Familienmitgliedern ausführen zu können. Dazu müssen diese Prozessoren – anders als die RISC-Prozessoren – mit einem Befehlssatz zurechtkommen, der durch sehr komplexe Befehle und Adressierungsarten gekennzeichnet ist. Andererseits wollten die Entwickler auf die Vorteile der RISC-Architekturen nicht verzichten. Dies führt zu dem in Abbildung 2.28 gezeigten inneren Aufbau der PC-Prozessoren.

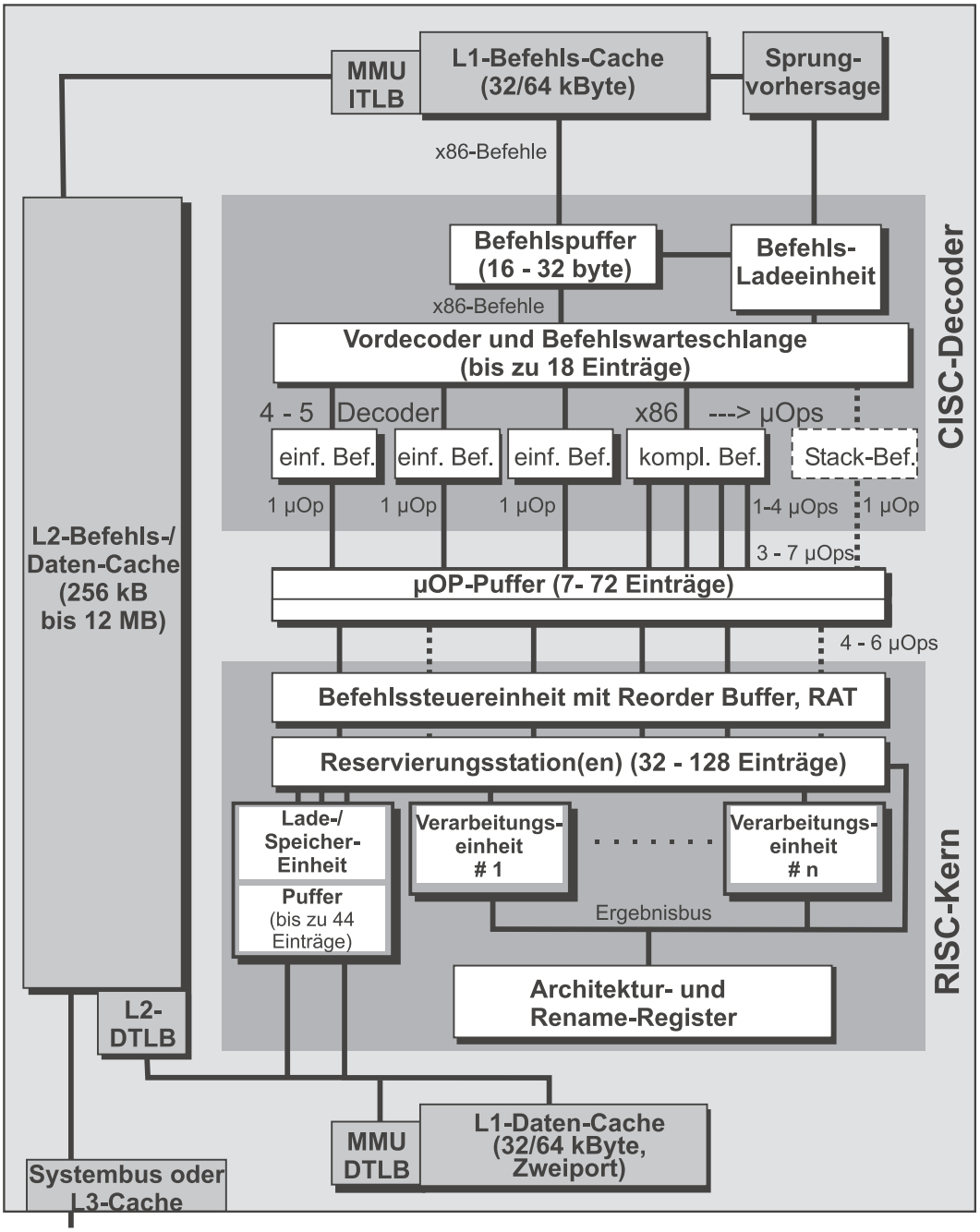


Abbildung 2.28: Architektur der x86-Prozessoren.

Als Hauptkomponenten erkennt man im Bild:

- einen **L2-Cache**, der als „unified Cache" sowohl Befehle als auch Daten (Operanden) enthält und dem Prozessorkern entweder als lokaler Cache (privater Cache) zur ausschließlichen Nutzung zugewiesen ist oder als gemeinsamer Cache mehreren Prozessorkernen zur Verfügung steht. Die Größe des L2-Caches liegt zwischen 256 kB als privater Cache und 12 MB als gemeinsamer Cache. Datenzugriffe auf den L2-Cache werden durch eine eigene Speicherverwaltungseinheit beschleunigt, die im Bild als L2-DTLB (L2-Data Translation Lookaside Buffer) bezeichnet ist und zu jedem Zeitpunkt 256 oder 512 Adressberechnungen unterstützt. Der Austausch von Befehlen und Daten zwischen dem L2-Cache und der „Außenwelt" des Prozessors geschieht über die Systembusschnittstelle oder eine besondere Schnittstelle zu einem weiteren Cache-Speicher, dem Level-3 Cache (L3-Cache).

- zwei **L1-Caches**, die einerseits für die Abspeicherung der momentan benötigten Befehle (L1-Befehls-Cache, L1-Instruction Cache) bzw. Daten (L1-Daten-Cache, L1-Data Cache) dienen. Auch sie verfügen über eigene Einheiten zur Adressberechnung, die mit MMU (Memory Management Unit) bzw. ITLB, DTLB (Instruction/Data Translation Lookaside Buffer) bezeichnet sind. Die Größe der L1-Caches variiert von 32 kB bei den Intel-Prozessoren bis zu 64 kB bei den AMD-Prozessoren. Eine Besonderheit des L1-Daten-Caches ist, dass er als Zweiport-Speicher realisiert ist und so zwei unabhängige Zugriffe auf zwei (verschiedene Daten zulässt, wobei diese Zugriffe wahlweise Schreib- oder Lesezugriffe sein können.

- Der **CISC-Decoder** hat die Aufgabe, aus den komplexen x86-Befehlen einfache Befehle („RISC-ähnliche Operationen" – μOps) zu erzeugen.

  – Die **Befehls-Ladeeinheit** (*Prefetcher*) entnimmt dem Befehls-Cache die Bytes der als nächstes auszuführenden x86-Befehle mit unterschiedlicher Länge und überträgt sie in den Befehlspuffer (Prefetch Buffer, Instruction Buffer), der 16 oder 32 Bytes groß ist.

    * Dies geschieht bei den Intel-Prozessoren über einen 128-bit-Datenpfad ohne jede Vorverarbeitung, sodass insbesondere im Befehlspuffer nicht zu erkennen ist, welche Bytes zu welchen x86-Befehlen gehören. Diese Erkennung von Beginn und Ende der einzelnen Befehle wird erst im folgenden Vordecoder vorgenommen, der die einzelnen Befehle durch die Einfügung von bestimmten Informationen voneinander trennt – was keinesfalls eine triviale Aufgabe ist.
    * Bei den AMD-Prozessoren liegen im L1-Befehls-Cache bereits vordecodierte Befehle, die über einen 256 bit breiten Datenpfad zwischen L1-Befehls-Cache und Befehlspuffer übertragen werden. Dazu kommen noch einmal 152 Leitungen, über die verschiedene Zusatzinformationen transferiert werden – unter anderem die eben erwähnten Vordecoderbits zur Kennzeichnung von Befehlsanfang und Ende, Bits zur Kennzeichnung von Verzweigungsbefehlen und Paritätsbits zur Fehlererkennung.

    Die vordecodierte Befehle werden in einer Befehlswarteschlange eingetragen, die bis zu 18 x86-Befehle aufnehmen kann. Die Intel-Prozessoren können dabei mehrere geeignete x86-Befehle zusammenfassen (MacroOps Fusion) und so die Decodierung dieser Befehle beschleunigen. Die Befehls-Ladeeinheit erkennt in den vordecodierten Befehlen Sprung- und Verzweigungsbefehle und stößt deren spekulative Ausführung mit Hilfe der beschriebenen Maßnahmen und Komponenten zur spekulativen Sprungvorhersage an. Diese veranlassen, dass der Vorgang des Befehlsholens an den vorhergesagten Stellen im L1-Cache fortgesetzt wird.

  – Die **Decodier-Einheit** (*Decoder*) übersetzt die x86-Befehle (unterschiedlicher Länge) in einen oder mehrere RISC-ähnliche Befehle konstanter Länge, die von Intel als μOps, von AMD als MacroOps bezeichnet werden[2] und aus über 100 Bits bestehen. Sie besteht aus mehreren parallel arbeitenden Decodern. Die Decoder haben unterschiedliche Befehlsgruppen zu verarbeiten, die sich insbesondere in ihrer Komplexität und der daraus

[2] Wir werden im Folgenden vereinfachend von „RISC-Operationen" sprechen.

resultierenden Anzahl der pro x86-Befehl erzeugten RISC-Operationen unterscheiden: drei Decoder bearbeiten einfache Befehle (wie z.B. die Addition) und übersetzen Sie in jeweils eine μOP. Der vierte Decoder ist für die Übersetzung von komplexen Befehlen zuständig, aus denen er bei AMD-Prozessoren wenigstens 3, bei den Intel-Prozessoren bis zu 4 μOps erzeugt. Sehr komplexe Befehle hingegen werden auch bei den modernsten PC-Prozessoren noch durch ein integriertes Mikroprogramm-Steuerwerk übersetzt, das für einen einzigen x86-Befehl einige Dutzend μOps sequentiell ausgeben kann. Als Besonderheit besitzen AMD-Prozessoren noch über einen weiteren Decoder zur Verarbeitung von Stack-Befehlen. Die Decoder-Einheit kann aus ihren Decodern zwischen 3 und 7 μOps in einem einzigen Taktintervall parallel in den folgenden μOp-Puffer übertragen, der von 7 bis 72 Einträge aufnehmen kann. Dieser Puffer trennt den CISC-Decoder von dem RISC-Kern, der für die Ausführung der Befehle verantwortlich ist.

- **Der RISC-Kern**

  – Die zentrale Einheit des RISC-Kerns ist die **Befehlssteuereinheit**, die für die korrekte Ausführung der Befehle sorgen muss und dazu die in den vorherigen Abschnitten beschriebenen Komponenten besitzt und verwaltet. Dazu gehören insbesondere der Umordnungspuffer (Reorder Buffer – ROB) und die Tabelle zur Umbenennung der Register (Register Allocation Table – RAT). Die Befehlssteuereinheit entnimmt pro Taktschwingung 4 bis 6 μOps dem μOp-Puffer, führt die Registerumbenennung (Register Renaming) durch, trägt die μOps in den ROB ein und übergibt sie daraufhin der folgenden Stufe zu. Bei den neuere Intel-Prozessoren kann die Befehlssteuereinheit auch mehrere geeignete μOps zusammenfassen (μOp Fusion) und so gemeinsam zur Ausführung weiterreichen. Dadurch wird die Ausführungsgeschwindigkeit der μOps erhöht.

  – Von der Befehlssteuereinheit gelangen die μOps in die **Reservierungsstation** (*Reservation Station*). Sie dient als Puffer für die μOps und speichert diese solange, bis alle Operanden verfügbar und die gewünschten Verarbeitungseinheiten frei sind. Durch die Ablage in der Reservierungsstation werden Verzögerungen in die Verarbeitung der Befehle eingefügt, deren Länge nicht vorhersehbar ist[3]. Die Prozessoren unterscheiden sich darin, ob es eine gemeinsame Reservierungsstation für alle Verarbeitungseinheiten gibt oder aber separate Stationen für Gruppen von Verarbeitungseinheiten realisiert sind.

    * Der erst genannte Fall liegt bei den Intel-Prozessoren vor, die eine Reservierungsstation mit bis zu 128 Einträgen besitzen. Hier wird die Reservierungsstation auch als *Instruction Pool* bezeichnet.
    * Die AMD-Prozessoren besitzen vier getrennte Reservierungsstationen, die als *Scheduler* bezeichnet werden und von denen drei jeweils acht Einträge aufweisen und einem Paar aus Integer-Rechenwerk bzw. Adressrechenwerk zugeordnet sind, die vierte mit 36 Einträgen für drei Gleitkommarechenwerke zuständig ist.

  – Die für den Anwender wichtigsten Komponenten des RISC-Kerns stellen die **Verarbeitungseinheiten** dar, zu denen die folgenden gehören:

    * bis zu fünf Rechenwerke zur Verarbeitung von arithmetischen Operationen mit ganzzahligen Operanden und logischen Operationen, die so genannten ALUs (*Arithmetic and Logical Unit*),
    * bis zu drei Rechenwerke zur Berechnung von Operanden- und Befehlsadressen (*Address Generation Unit* – AGU),
    * Rechenwerke zur Verarbeitung von Gleitpunktzahlen (*Floating-Point Unit* – FPU), die jedoch üblicherweise auf bestimmte Operationen spezialisiert sind, z.B. die Gleitkomma-Addition oder –Multiplikation/Division bzw. alle übrigen Gleitkomma–Operationen.

[3] und nur durch einen statistisch gewonnenen Mittelwert angegeben werden können

  - zwei oder drei Rechenwerke zur Verarbeitung von Multimedia-Befehlen auf ganzen Zahlen (*Multi-Media Extension* – MMX) bzw. Gleitkommzahlen (*Streaming SIMD Extension* – SSE). Diese Einheiten führen insbesondere auch die Befehle aus, die die Komponenten eines Vektors in eine beliebige Reihenfolge setzen und als *Shuffle*-Befehle bezeichnet werden.

– Als **Architekturregister**, d.h. Register, die vom Programmierer direkt angesprochen werden können, besitzen alle betrachteten Prozessoren den Satz der ursprünglichen universellen Register (*General Purpose Register* – GPR) ihres „Stammvaters" Intel 8086, die jedoch im Laufe der Entwicklungsgeschichte von ursprünglich 8 bzw. 16 bit Länge auf bis zu 64 bit verlängert wurden (s. Abbildung 2.29). Dazu kommen die acht 80–bit–Gleitkommaregister (MMXi/FPRi) der ersten FPU Intel 8087, die jedoch als 64–bit–Register gleichzeitig für die Verarbeitung der MMX-Daten dienen. Für die Verarbeitung der SSE-Operanden wurden 16 128-bit-Register (XMMi) hinzugefügt, die als zweiter Registersatz auch für die MMX-Befehle zur Verfügung stehen.

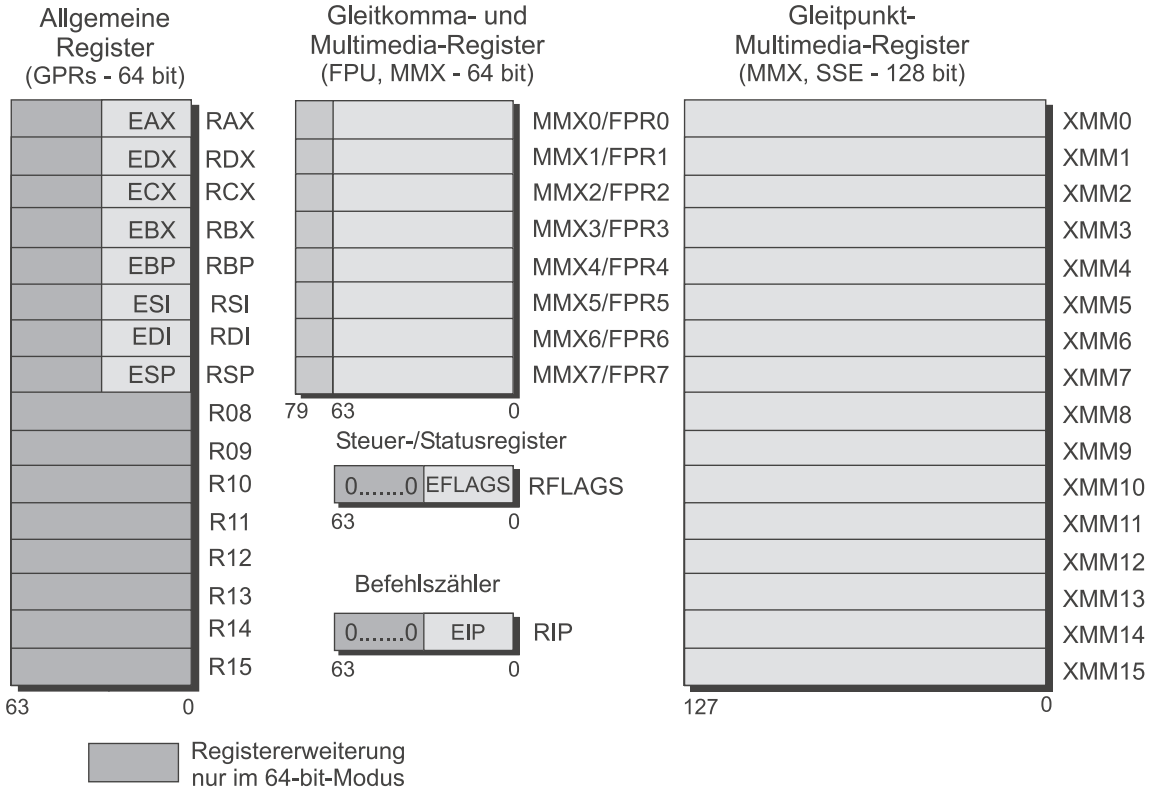


Abbildung 2.29: Die Architekturregister der PC-Prozessoren.

Zu den Architekturregister kommt noch eine unterschiedlich große Anzahl von Umbenennungsregister, die zur Aufnahme der Ergebnisse von spekulativ ausgeführten Operationen benötigt werden.

– Die Übertragung der Ergebnisse von den Verarbeitungseinheiter in die Register geschieht über einen speziellen Bus, den **Ergebnisbus** (*Result Bus*). Dieser führt aber die Ergebnisse auch direkt zur Reservierungsstation bzw. zu den Reservierungsstationen zurück, so dass sie dort so schnell wie möglich für die Ausführung von wartenden Operationen zur Verfügung stehen und nicht erst zeitaufwändigen den Umweg durch die Registersätze nehmen müssen.

– Die letzte Komponente, die **Lade-/Speichereinheit** (*Load/Store Unit*), die für die Ausführung der Zugriffe auf den L1-Daten-Cache zuständig ist. In drei Untereinheiten können gleichzeitig Adressberechnungen für jeweils einen Lese- und einen Schreibzugriff sowie

außerdem eine Datenschreib-Operation ausgeführt werden. In einem Puffer können bis zu 44 Schreib-/Leseoperationen gespeichert werden. Die Lade-/Speichereinheit untersucht dabei diese Befehle auf Datenabhängigkeiten und löst diese selbstständig auf. So erkennt sie z.B. Lesezugriffe auf Daten, die von einer gepufferten Schreiboperation erst noch im L1-Daten-Cache abgelegt werden soll. In diesem Fall entnimmt sie das verlangte Datum direkt dem Pufferspeicher.

Da ein x86-Prozessor der beschriebenen Art x86–Befehle zunächst in RISC–ähnliche Operationen umwandelt und diese dann von seinem RISC-Kern verarbeiten lässt, gibt es nun zwei unterschiedliche Möglichkeiten, den **Superskalaritätsgrad** zu definieren: Einerseits kann er die maximale Anzahl der x86-Befehle bezeichnen, die in einem Takt an den Decoder übergeben werden, andererseits die maximale Anzahl der RISC-Operationen, die in einem Taktzyklus aus der Reservierungsstation/den Reservierungsstationen an die Verarbeitungseinheiten geschickt werden können. In der Regel ist der zweite genannte Wert größer als der erste. Da die Prozessoren über vier bis fünf Decoder, ist der „x86-Superskalaritätgrad" 4 – 5. Der „μOp–Superskalaritätsgrad" ist 6, da in jedem Takt maximal sechs μOPs in die Reservierungsstation(en) übertragen werden können.

2.8 Anhang: In dieser Kurseinheit 2 verwendete Befehle

Befehlssyntax		Semantik	verbale Beschreibung
arithmetische Befehle			
ADD	R1,R2,R3	R1=R2+R3	Inhalte der Register R2 und R3 werden addiert, Ergebnis nach Register R1 (alle Werte vom Typ Real)
ADD	R1,R2,#C	R1=R2+C	Inhalt des Registers R2 und Wert der Konstanten C werden addiert, Ergebnis nach Register R1 (alle Werte vom Typ Real)
SUB	R1,R2,R3	R1=R2-R3	Inhalt des Registers R3 wird vom Inhalt des Register R2 subtrahiert, Ergebnis nach Register R1 (alle Werte vom Typ Real)
SUB	R1,R2,#C	R1=R2-C	Wert der Konstanten C wird vom Inhalt des Register R2 subtrahiert, Ergebnis nach Register R1 (alle Werte vom Typ Real)
MUL	R1,R2,R3	R1=R2*R3	Inhalte der Register R2 und R3 werden multipliziert, Ergebnis nach Register R1 (alle Werte vom Typ Real)
MUL	R1,R2,#C	R1=R2*C	Wert der Konstanten C wird mit Inhalt des Register R2 multipliziert, Ergebnis nach Register R1 (alle Werte vom Typ Real)
ADDI	R1,R2,R3	R1=R2+R3	Inhalte der Register R2 und R3 werden addiert, Ergebnis nach Register R1 (alle Werte vom Typ Integer)
ADDI	R1,R2,#C	R1=R2+C	Inhalt des Registers R2 und Wert der Konstanten C werden addiert, Ergebnis nach Register R1 (alle Werte vom Typ Integer)
SUBI	R1,R2,R3	R1=R2-R3	Inhalt des Registers R3 wird vom Inhalt des Register R2 subtrahiert, Ergebnis nach Register R1 (alle Werte vom Typ Integer)
SUBI	R1,R2,#C	R1=R2-C	Wert der Konstanten C wird vom Inhalt des Register R2 subtrahiert, Ergebnis nach Register R1 (alle Werte vom Typ Integer)
Vergleichs-Befehle			
SGE	R1,R2,R0	R2-R0, R1=¬N	Abfrage R2$\geq$0, wenn ja $\rightarrow$ R1=1, wenn nein $\rightarrow$ R1=0
SLE	R1,R2,R0	R2-R0, R1=N	Abfrage R2$\leq$0, wenn ja $\rightarrow$ R1=1, wenn nein $\rightarrow$ R1=0
Sprung-Befehle			
JMP	L1	GOTO L1	unbedingter Sprung nach Marke L1
BEQZ	R1,L1	GOTO L1 IF R1=0	bedingter Sprung nach Marke L1 wenn R1=0
BNEZ	R1,L1	GOTO L1 IF R1$\neq$0	bedingter Sprung nach Marke L1 wenn R1$\neq$0
BEQZ	nn	GO ON *nn* IF Z=1	bedingter Sprung um *nn* Worte falls Zero-Bit gleich 1
BNEZ	nn	GO ON *nn* IF Z=0	bedingter Sprung um *nn* Worte falls Zero-Bit gleich 0

Bei den Befehlen SGE, SLE bezeichnet N das Negativ-Bit und bei BEQZ und BNEZ bezeichnet Z das Zero-Bit innerhalb des Statusregisters (siehe Seite 15). Jeder Prozessor besitzt ein solches Statusregister, in dem bestimmte durch Befehle erzeugte Informationsbits gespeichert werden, die der Programmsteuerung dienen. Auf dieses Statusregister wird in dieser Kurseinheit nicht weiter eingegangen. Informationen darüber, welche Assemblerbefehle die Statusbits in welcher Form beeinflussen und welche Befehle die Statusbits nutzen, findet man in der Befehlstabelle des jeweiligen Prozessors. Hier wird angenommen, dass die Befehle SGE und SLE das Negativ-Bit verändern und auslesen und die Befehle BEQZ und BNEZ das Zero-Bit auslesen und für die Sprungentscheidung nutzen.

2.9 Lösungen zu den Selbsttestaufgaben

Aufgabe 2.1

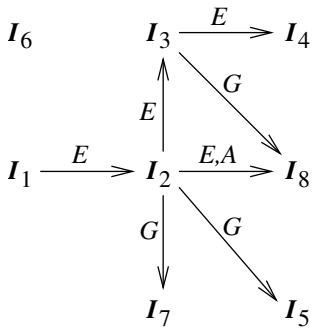
Der Speicher besitzt eine Kapazität von 4 Gigabyte.
1 KB sind $1024 = 2^{10}$ Byte, 1 MB sind 2^{10} KB, also $2^{10} \cdot 2^{10} = 2^{20}$ Byte,
1 GB sind 2^{10} MB, also $2^{10} \cdot 2^{20} = 2^{30}$ Byte.
Damit ist die Anzahl der möglichen Adressen $4 \cdot 2^{30}/8 = 2^{29}$.
Eine Adresse besteht somit aus mindestens 29 Bit.

Aufgabe 2.2

Die registerindirekte Adressierung mit Verschiebung kombiniert die registerindirekte Adressierung mit der absoluten Adressierung, d.h. wird der Verschiebewert auf Null gesetzt, so erhält man die registerindirekte Adressierung, wird dagegen der Registerwert auf Null gesetzt, so erhält man die absolute Adressierung.

Aufgabe 2.3

Datenabhängigkeiten	Echte	Gegen	Ausgabe
	$I_1 \longrightarrow I_2$	$I_2 \longrightarrow I_7$	$I_2 \longrightarrow I_8$
	$I_2 \longrightarrow I_3$	$I_2 \longrightarrow I_5$	
	$I_2 \longrightarrow I_8$	$I_3 \longrightarrow I_8$	
	$I_3 \longrightarrow I_4$		



Steuerflussabhängigkeiten: Nur die bedingte Sprunganweisung I_4 erzeugt eine Kontrollabhängigkeit. Allerdings gibt es noch eine weitere Steuerflussänderung bei der unbedingten Sprunganweisung I_6.

Aufgabe 2.4

Einfügen von NOP-Befehlen

I_1	ADDI	R1,R2,2	R1=R2+2
	NOP		
	NOP		
I_2	SUB	R4,R1,R3	R4=R1-R3
	NOP		
	NOP		
I_3	SGE	R7,R4,R0	R4 $\geq$ 0?, Status in R7
	NOP		
	NOP		
I_4	BNEZ	R7,I7	wenn ja, gehe zu I_7
	NOP		
	NOP		
	NOP		
I_5	MUL	R3,R5,R6	R3=R5*R6
I_6	J	I8	Springe nach I8
	NOP		
	NOP		
	NOP		
I_7	ADDI	R3,R3,2	R3=R3+2
I_8	ADDI	R4,R4,1	R4=R4+1

Aufgabe 2.5

a) Pipeline-Konflikte sind: Datenkonflikte, Struktur-/Ressourcenkonflikte und Steuerflusskonflikte.

Datenkonflikte sind: Lese-nach-Schreib-Konflikt, Schreib-nach-Lese-Konflikt und Schreib-nach-Schreib-Konflikt.

b) Keine Pipelinekonflikte ergeben sich bei Gegenabhängigkeit und Ausgabeabhängigkeit.
Bei der Gegenabhängigkeit wird zuerst ein Register gelesen, dann geschrieben. Da der Lesevorgang bei der gegebenen Pipeline schon beendet ist, bis ein Schreibvorgang beginnt, tritt kein Konflikt auf. Ebenso wird ein nachfolgender Schreibvorgang erst begonnen, wenn der vorherige Schreibvorgang schon beendet ist, so dass auch die Ausgabeabhängigkeit zu keinem Konflikt führt.

c) IF (Befehl holen), ID (Decodieren und Operanden laden), EX (Befehl ausführen bzw. effektive Adresse berechnen), MEM (Speicherzugriff), WB (Zurückschreiben in das Register in der ersten Hälfte des Taktzyklus).

d) Wenige und einfache Befehle, Befehlsformate, Adressierungsarten; einheitliche Befehlslänge von 32 Bit; Lade-/Speicherarchitektur; große Zahl von allgemeinen Registern.

e) Alle komplexen Adressierungsarten können durch einfache Adressierungsarten ersetzt werden. Allerdings sind dafür mehrere Befehle notwendig.

Aufgabe 2.6

Superpipeline-Prozessoren nutzen zeitliche Parallelität durch eine Verdopplung der Taktfrequenz in Verbindung mit einer längeren Pipeline; superskalare Prozessoren nutzen räumliche Parallelität durch mehr Ausführungseinheiten.

Aufgabe 2.7

a) Prädiktorzustand für den Ein-Bit-Prädiktor

Sprung	Initialzustand	d=0	d=2	d=0
S1	T	NT	T	NT
S2	T	NT	T	NT

b) Prädiktorzustand für den Zwei-Bit-Prädiktor mit Sättigungszähler

Sprung	Initialzustand	d=0	d=2	d=0
S1	WT	WNT	WT	WNT
S2	WT	WNT	WT	WNT

c) Prädiktorzustand für den Zwei-Bit-Prädiktor mit Hysteresezähler

Sprung	Initialzustand	d=0	d=2	d=0
S1	WT	SNT	WNT	SNT
S2	WT	SNT	WNT	SNT

Aufgabe 2.8

Zahl der durchlaufenen Verzweigungsbefehle der inneren und äußeren Schleife
Die Tabelle auf der nächsten Seite zeigt die Sprungverläufe der einzelnen inneren und äußeren Schleifen an. Dabei steht *t* für einen genommenen und *nt* für einen nicht genommenen Sprung.

Im ersten Durchlauf der äußeren Schleife wird der Sprungbefehl S2 der inneren Schleife $(m+1)$-mal durchlaufen. In den nächsten $(n-1)$ Durchläufen der äußeren Schleife wird der Sprungbefehl S2 der inneren Schleife nur je einmal durchlaufen und im letzten Durchlauf nicht mehr.
Es gibt also $(m+1)$ zuzüglich $(n-1)$ Durchläufe der inneren Schleife bzw. des Sprungbefehls S2.
Der Sprungbefehl S1 wird also $(n+1)$-mal und der Sprungbefehl S2 $(n+m)$-mal durchlaufen.

Anzahl Durchläufe		
Sprung S1	Sprung S2	Insgesamt
$n+1$	$n+m$	$n+m+1$

Äußere Schleife Durchlauf	Sprung S1	Innere Schleife Durchlauf	Sprung S2	Wert in Register R3	Wert in Register R4
1		1	*nt*	n	m
		2	*nt*	n	$m-1$
		$\cdots$	$\cdots$	n	$\cdots$
		$m-2$	*nt*	n	2
		$m-1$	*nt*	n	1
		m	*nt*	n	0
		$m+1$	*t*	n	<0
	nt			n	<0
2		1	*t*	$n-1$	<0
	nt			$n-1$	<0
3		1	*t*	$n-2$	<0
	nt			$n-2$	<0
$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$
$n-1$		1	*t*	1	<0
	nt			1	<0
n		1	*t*	0	<0
	nt			0	<0
$n+1$	*t*			<0	<0

Für die $(n+1)$ äußeren Schleifendurchläufe wird der Sprung S1 bei den ersten n Durchläufen nicht genommen (*nt*) und beim $(n+1)$-ten Durchlauf genommen (*t*).

Im ersten Durchlauf der äußeren Schleife wird der Sprung S2 bei den ersten m Durchläufen der inneren Schleife nicht genommen (*nt*) und beim $(m+1)$-ten Durchlauf genommen (*t*). In allen weiteren $(n-1)$ Durchläufen der äußeren Schleife wird der Sprungbefehl S2 der inneren Schleife einmal genommen (*t*). Der Sprung S2 wird also insgesamt m-mal nicht genommen und $(n+1)$-mal genommen.

	Sprung S1	Sprung S2	Insgesamt
nicht genommen	n	m	$n+m$
genommen	1	n	$n+1$
	$n+1$	$n+m$	$2n+m+1$

a) Zahl der richtigen Vorhersagen bei statischen Verfahren

Äußere Schleife Durchlauf	Sprung S1	Innere Schleife Durchlauf	Sprung S2	Vorhersage $a1)$	$a2)$	$a3)$	Wert in Register R3	Wert in Register R4
1		1	*nt*	*NT*	*T*	*NT*	n	m
		2	*nt*	*NT*	*T*	*NT*	n	$m-1$
		$\cdots$	$\cdots$	*NT*	*T*	*NT*	n	$\cdots$
		$m-2$	*nt*	*NT*	*T*	*NT*	n	2
		$m-1$	*nt*	*NT*	*T*	*NT*	n	1
		m	*nt*	*NT*	*T*	*NT*	n	0
		$m+1$	*t*	*NT*	*T*	*NT*	n	<0
	nt			*NT*	*T*	*NT*	n	<0
2		1	*t*	*NT*	*T*	*NT*	$n-1$	<0
	nt			*NT*	*T*	*NT*	$n-1$	<0
3		1	*t*	*NT*	*T*	*NT*	$n-2$	<0
	nt			*NT*	*T*	*NT*	$n-2$	<0
$\cdots$	$\cdots$	$\cdots$	$\cdots$	*NT*	*T*	*NT*	$\cdots$	$\cdots$
$n-1$		1	*t*	*NT*	*T*	*NT*	1	<0
	nt			*NT*	*T*	*NT*	1	<0
n		1	*t*	*NT*	*T*	*NT*	0	<0
	nt			*NT*	*T*	*NT*	0	<0
$n+1$	*t*			*NT*	*T*	*NT*	<0	<0

Die Tabelle gibt die Sprungverläufe der einzelnen inneren und äußeren Schleifen mit den entsprechenden Vorhersagen an. Dabei steht *T* für einen als genommen und *NT* für einen als nicht genommen vorhergesagten Sprung.

a1) *predict always not taken*
Die Sprünge werden als immer nicht genommen vorausgesagt. Damit werden die $(n+m)$ nicht genommenen Sprüngen *richtig* und die $(n+1)$ genommenen Sprüngen *falsch* vorhergesagt.

a2) *predict always taken*
Die Sprünge werden als immer genommen vorausgesagt. Damit werden die $(n+1)$ nicht genommenen Sprüngen *richtig* und die $(n+m)$ genommenen Sprüngen *falsch* vorhergesagt.

a3) *predict backward taken/forward not taken*
Da alle Sprünge vorwärtige Sprünge sind, ist das Ergebnis gleich dem vom Verfahren a1).

	Vorhersagen für *predict always taken*		
Vorhersage	Sprung S1	Sprung S2	Insgesamt
richtig	1	n	$n+1$
falsch	n	m	$n+m$
	$n+1$	$n+m$	$2n+m+1$

b) Zahl der richtigen Vorhersagen mit einem *Ein-Bit-Prädiktor*
(Initialzustand: *T* "predict taken")

Wir können hier davon ausgehen, dass es für jeden Sprungbefehl einen eigenen Prädiktor gibt.
Die zwei Zustände des 1-Bit-Prädiktors sind: *T*=predict taken und *NT*=predict not taken.
Die folgende Tabelle gibt die Sprungverläufe der einzelnen äußeren (S1) und inneren (S2) Schleifen mit den entsprechenden Vorhersagen für einen Ein-Bit-Prädiktor an. Dabei steht *W* für eine richtige Vorhersage und *F* für eine falsche Vorhersage.

Äußere Schleife		Innere Schleife		Ein-Bit-Prädiktor				
Durchlauf	S1	Durchlauf	S2	Vorhersage	Folgezustand	Korrekt?	R3	R4
Anfangszustand der Prädiktoren					*T*			
1		1	*nt*	*T*	*NT*	*F*	n	m
		2	*nt*	*NT*	*NT*	*R*	n	$m-1$
		$\cdots$	$\cdots$	*NT*	*NT*	*R*	n	$\cdots$
		$m-2$	*nt*	*NT*	*NT*	*R*	n	2
		$m-1$	*nt*	*NT*	*NT*	*R*	n	1
		m	*nt*	*NT*	*NT*	*R*	n	0
		$m+1$	*t*	*NT*	*T*	*F*	n	<0
	nt			*T*	*NT*	*F*	n	<0
2		1	*t*	*T*	*T*	*R*	$n-1$	<0
	nt			*NT*	*NT*	*R*	$n-1$	<0
3		1	*t*	*T*	*T*	*R*	$n-2$	<0
	nt			*NT*	*NT*	*R*	$n-2$	<0
$\cdots$	$\cdots$	$\cdots$	$\cdots$	*NT*	*NT*	*R*	$\cdots$	$\cdots$
$n-1$		1	*t*	*T*	*T*	*R*	1	<0
	nt			*NT*	*NT*	*R*	1	<0
n		1	*t*	*T*	*T*	*R*	0	<0
	nt			*NT*	*NT*	*R*	0	<0
$n+1$	*t*			*NT*	*NT*	*F*	<0	<0

Im ersten äußeren Schleifendurchlauf wird der Sprung S1 einmal *falsch* vorhergesagt. Der Sprung S2 wird zweimal *falsch* und $(m-1)$-mal *richtig* vorhergesagt.
In den nächsten $(n-1)$ äußeren Schleifendurchlauf wird der Sprung S1 je einmal *richtig* vorhergesagt. Der Sprung S2 wird auch je einmal *richtig* vorhergesagt.
Im letzten äußeren Schleifendurchlauf wird der Sprung S1 einmal *falsch* vorhergesagt.

	Vorhersagen für *Ein-Bit-Prädiktor*		
Vorhersage	Sprung S1	Sprung S2	Insgesamt
richtig	$n-1$	$n+m-2$	$2n+m-3$
falsch	2	2	4
	$n+1$	$n+m$	$2n+m+1$

c) Zahl der richtigen Vorhersagen mit einem *Zwei-Bit-Prädiktor mit Sättigungszähler*
(Initialzustand: *WT* "predict weakly taken")

Wir können auch hier davon ausgehen, dass es für jeden Sprungbefehl einen eigenen Prädiktor gibt. Für die Zustände der 2-Bit-Prädiktoren gilt:

ST	=	predict strongly taken	SNT	=	predict strongly not taken
WT	=	predict weakly taken	WNT	=	predict weakly not taken

Die folgende Tabelle gibt die Sprungverläufe der einzelnen äußeren (S1) und inneren (S2) Schleifen mit den entsprechenden Vorhersagen für einen Ein-Bit-Prädiktor an. Dabei steht *W* für eine richtige Vorhersage und *F* für eine falsche Vorhersage.

Äußere Schleife		Innere Schleife		Zwei-Bit-Prädiktor				
Durchlauf	S1	Durchlauf	S2	Vorhersage	Folgezustand	Korrekt?	R3	R4
Anfangszustand der Prädiktoren					*WT*			
1		1	*nt*	*T*	*WNT*	*F*	n	m
		2	*nt*	*NT*	*SNT*	*R*	n	$m-1$
		$\cdots$	$\cdots$	*NT*	*SNT*	*R*	n	$\cdots$
		$m-2$	*nt*	*NT*	*SNT*	*R*	n	2
		$m-1$	*nt*	*NT*	*SNT*	*R*	n	1
		m	*nt*	*NT*	*SNT*	*R*	n	0
		$m+1$	*t*	*NT*	*WNT*	*F*	n	<0
	nt			*T*	*WNT*	*F*	n	<0
2		1	*t*	*NT*	*WT*	*F*	$n-1$	<0
	nt			*NT*	*SNT*	*F*	$n-1$	<0
3		1	*t*	*T*	*ST*	*R*	$n-2$	<0
	nt			*NT*	*SNT*	*R*	$n-2$	<0
$\cdots$	$\cdots$	$\cdots$	$\cdots$	*NT*	*SNT*	*R*	$\cdots$	$\cdots$
$n-1$		1	*t*	*T*	*ST*	*R*	1	<0
	nt			*NT*	*SNT*	*R*	1	<0
n		1	*t*	*T*	*ST*	*R*	0	<0
	nt			*NT*	*SNT*	*R*	0	<0
$n+1$	*t*			*NT*	*SNT*	*F*	<0	<0

Im ersten äußeren Schleifendurchlauf wird der Sprung S1 einmal *falsch* vorhergesagt. Der Sprung S2 wird zweimal *falsch* und $(m-1)$-mal *richtig* vorhergesagt.
Im zweiten äußeren Schleifendurchlauf wird der Sprung S1 einmal *richtig* vorhergesagt. Der Sprung S2 wird einmal *falsch* vorhergesagt.
In den nächsten $(n-2)$ äußeren Schleifendurchlauf wird der Sprung S1 je einmal *richtig* vorhergesagt. Der Sprung S2 wird auch je einmal *richtig* vorhergesagt.
Im letzten äußeren Schleifendurchlauf wird der Sprung S1 einmal *falsch* vorhergesagt.

	Vorhersagen für *Ein-Bit-Prädiktor*		
Vorhersage	Sprung S1	Sprung S2	Insgesamt
richtig	$n-1$	$n+m-3$	$2n+m-4$
falsch	2	3	5
	$n+1$	$n+m$	$2n+m+1$

Aufgabe 2.9

Der Initialzustand des $(1,1)$-Prädiktors ist

BHR [1]

	0	1
S1	T	T
S2	T	T

Anfangswert für d bei Iterationsbeginn	d=0 ?	Sprung-richtung für S1	d vor S2	d=1 ?	Sprung-richtung für S2
0	Ja	nt	1	ja	nt
2	Nein	t	2	nein	t

a) $d=0$

Laut Tabelle werden für den Fall $d=0$ sowohl der erste Sprung S1 als auch der zweite Sprung S2 nicht genommen.

Die Vorhersage für S1 mit dem BHR [1] ist T und damit *falsch*. S1 bewirkt die Änderung des BHR auf 0. In der PHT wird für S1 das Element unter 1 aktualisiert, und zwar von T auf NT.

BHR [0]

	0	1
S1	T	NT
S2	T	T

Die Vorhersage für S2 mit dem BHR [0] ist T und damit *falsch*. Das BHR bleibt auf 0. In der PHT wird für S2 das Element unter 0 aktualisiert, und zwar von T auf NT.

BHR [0]

	0	1
S1	T	NT
S2	NT	T

b) $d=2$

Laut Tabelle werden für den Fall $d=2$ sowohl der erste Sprung S1 als auch der zweite Sprung S2 genommen.

Die Vorhersage für S1 mit dem BHR [0] ist T und damit *richtig*. S1 bewirkt die Änderung des BHR auf 1. Die PHT bleibt unverändert.

BHR [1]

	0	1
S1	T	NT
S2	NT	T

Die Vorhersage für S2 mit dem BHR [1] ist T und damit *richtig*. Das BHR als auch die PHT bleiben unverändert.

BHR [1]

	0	1
S1	T	NT
S2	NT	T