Kurs 01609 Computersysteme II

Autor: Prof. Dr. T. Ungerer

Überarbeitung:
Dr. H. Bähring
Prof. Dr. J. Keller
Prof. Dr. W. Schiffmann

Kurseinheiten 1 – 4

Inhaltsverzeichnis

Vorwort

Liebe Studierende,

wir begrüßen Sie herzlich zum Kurs Computersysteme II (1609). Dieser Kurs führt Sie in die Grundlagen der *Rechnerarchitektur* ein. Neben grundlegenden Prozessortechniken werden die Architekturkonzepte hochperformanter Mikroprozessoren behandelt und an Beispielen verdeutlicht. Der Kurs befasst sich aber auch mit der Speicherhierarchie und Zukunftstechniken für Mikroprozessoren. So werden z. B. mehrstufige Caches verwendet, um die hohen Taktraten moderner Mikroprozessoren voll auszunutzen. Die Leistungsgrenzen heutiger Prozessortechniken können nur durch parallel arbeitende Architekturen überwunden werden. Auf der Prozessorebene kann der Durchsatz durch eine mehrfädige Befehlsverarbeitung weiter gesteigert werden. Parallele Rechnersysteme verwenden entweder mehrere Prozessoren oder parallel arbeitende Rechenwerke. Bei Multiprozessorsystemen muss die Aufgabenstellung in Teilprobleme zerlegt werden, die durch miteinander kommunizierende Programme gleichzeitig bearbeitet werden. Im Kurs werden sowohl nachrichten- als auch speichergekoppelte Parallelrechnerysteme behandelt.

Der Kurs wurde zum Sommersemester 2009 überarbeitet. Dabei wurde ein in sich geschlossener Kurstext erstellt, der auch über ein gemeinsames Inhalts-, Literatur- und Stichwortverzeichnis (Index) verfügt. Die einzelnen Kapitel stimmen mit den jeweiligen Kurseinheiten 1-4 überein, d.h. die Bearbeitungszeiträume und zugehörigen Einsendetermine beziehen sich nun auf die Kapitelnummern. Der bisherige Kursinhalt wurde stark gekürzt und auf die wesentlichen Grundlagen reduziert. Wir hoffen, dass dadurch der überarbeitete Kurs leichter zu verstehen und der Bearbeitungsaufwand im Vergleich zur bisherigen Fassung geringer ist.

Wichtiger Hinweis

Diesem Kurs liegt in weiten Teilen das Buch „Mikrocontroller und Mikroprozessoren“ von U. Brinkschulte und T. Ungerer [3] zugrunde. Dort können Sie sich intensiv über Themen informieren, die aus Platzgründen in diesem Kurs nicht oder nicht sehr tiefgehend behandelt werden können. Insbesondere finden Sie dort zu allen behandelten Themen ein ausführliches Literaturverzeichnis.

Wir wünschen Ihnen viel Spaß beim Bearbeiten des Kurses!

Ihre Kursbetreuer

Kapitel 1

Grundlegende Prozessortechniken

Kapitelinhalt

Zusammenfassung

Wir werden in diesem Kapitel die wesentlichen Prozessortechniken, d.h. die für den System- und Assemblerprogrammierer zu beachtenden Details der Befehlssatzarchitektur und die Pipelining-Mechanismen, ausführlich behandeln. Die Grundlagen dieser Techniken wurden bereits in KE4 von Kurs 1608 kurz dargestellt.

Wir beginnen in Abschnitt 1 mit einer kurzen Einführung in die Rechnerarchitektur. Im 2. Abschnitt führen wir in die wichtigsten Eigenschaften der Befehlssatzarchitekturen ein. Am Ende des Abschnitts wird näher auf die Unterschiede zwischen CISC- und RISC-Befehlssätzen (*Complex* bzw. *Reduced Instruction Set Computer*) eingegangen.

Der 3. Abschnitt behandelt die besonderen Eigenschaften der RISC-Architekturen und zeigt als Anwendung der im 1. Abschnitt gelernten Techniken die Befehlssatzarchitektur des DLX-Prozessors – eines von den bekannten amerikanischen Wissenschaftlern Hennessy und Patterson entworfenen „Lehrprozessors“, der mit den in Steuerungssystemen, wie z.B. Navigationssystemen, Empfängern fürs Satellitenfernsehen und Spieleconsolen, häufig verwendeten MIPS-Prozessoren fast identisch ist. Die zu diesem Kapitel gehörenden Programmieraufgaben in Maschinensprache (bzw. Assembler) werden mit dem vorgestellten DLX-Befehlssatz programmiert.

Abschnitt 4 behandelt die Umsetzung des von-Neumann-Prinzips durch Befehls-Pipelining. Abschnitt 5 führt in den Entwurf von Pipelining-Prozessoren ein. Die Implementierung der Befehlssatzarchitektur des DLX-Prozessors durch eine fünfstufige Pipeline wird im Detail behandelt, ebenso die Lösungen zu auftretenden Pipeline-Konflikten.

Dieses Kapitel beschränkt sich auf Prozessoren, die pro Takt in jeder Stufe der Pipeline nur einen Befehl ausführen können. Die hier gelernten Programmier- und Prozessortechniken finden ihre direkte Anwendung bei den in heutigen Steuerungssystemen eingesetzten Mikrocontrollern. (Diese sog. „eingebetteten Systeme“ werden im Kurs 1706 ausführlich behandelt.) Abschnitt 6 verweist auf weitere Aspekte des Befehls-Pipelining. Dieser Abschnitt bildet den Übergang zum Kapitel 2, das der Superskalartechnik gewidmet ist, die insbesondere in den hochperformanten Mikroprozessoren moderner Personal Computer (PC) eingesetzt wird.

Lernziele

Die Lernziele dieses Kapitels sind:

- Zusammenhang zwischen Prozessor- und Mikroarchitektur,
- Komponenten einer Befehlssatzarchitektur,
- CISC- und RISC-Prinzipien,
- Mikroarchitektur einfacher Prozessoren und Prozessorkerne,
- Befehlspipelining.

1.1 Rechnerarchitektur

In der letzten Kurseinheit des Kurses 1608 haben Sie gelernt, wie ein Computer nach dem von Neumann Prinzip [4] aufgebaut ist. Er besteht aus vier Funktionseinheiten: dem Rechen- und Leitwerk, die zusammen den Prozessor bilden, dem Speicher und einer Ein-/Ausgabe. Das Zusammenspiel dieser vier Komponenten bestimmt die Leistungsfähigkeit eines Computers. Speicher und Ein-/Ausgabe werden über die Systembusschnittstelle angesprochen und belegen jeweils verschiedene Bereiche in dem vom Prozessor ansprechbaren Adressraum. Der Aufbau des Speichersystems und dessen Einfluss auf die Leistungsfähigkeit eines Computers wird im Kapitel 3 ausführlich behandelt. Ein-/Ausgabeeinheiten bestehen im allgemeinen aus spezialisierten Controllerbausteinen, die Peripheriegeräte zur Interaktion mit dem Menschen (Tastatur, Monitor, Maus, Drucker), anderen Computern (lokale Netzwerke, Internet) oder nichtflüchtige Speichermedien (Festplatten, CD–ROM/DVD) unterstützen. Diese Controllerbausteine werden ähnlich wie Speicher über einen speziellen Adressbereich angesprochen.

Entscheidenden Einfluss auf die Leistung und die Einsatzmöglichkeiten eines Computers hat das Programmiermodell des Prozessors. Es beschreibt die Sicht eines Systemprogrammierers auf den Prozessor und beinhaltet alle notwendigen Details, um ablauffähige Maschinenprogramme für den betreffenden Prozessor zu erstellen. Natürlich muss auch der Compiler dieses Programmiermodell kennen, um Hochsprachenprogramme in Maschinensprache zu übersetzen. Da dieses 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. Sie beschreibt zwar das Verhalten des Prozessors nicht aber seine Implementierung, die entweder durch die *logische Organisation* oder durch die tatsächliche *technologische Realisierung* beschrieben werden kann (s. Abbildung 1.1).

Befehlssatzarchitektur und Architektur werden synonym verwendet.

Abbildung 1.1: 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 *Rechnerarchitekt* 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 Rechnerarchitekt dann eine logische Struktur zur Implementierung ab (diese wird auch *Mikroarchitektur* genannt). Dabei legt er fest, welche Funktionseinheiten (z.B. Register(sätze), ALUs, Mul-

Mikroarchitektur

tiplexer 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).

Im Laufe der Entwicklung von Computersystemen kamen verschiedene Technologien zur Realisierung von Prozessoren zum Einsatz. Es begann mit elektromechanischen Bauelementen, wenig später verwendete man Elektronenröhren bzw. einzelne Transistoren und schließlich ab Ende der fünfziger Jahre integrierte Schaltkreise (*Integrated Circuits* – ICs), die ganze Schaltungen mit mehreren Transistoren auf einem einzigen Halbleiterchip realisieren. Die Integrationsdichte hat sich seit der Einführung integrierter Schaltkreise stetig gesteigert. Gordon Moore, Mitbegründer des weltweit bekannten Prozessorherstellers Intel, stellte 1965 in einem Beitrag zur Zeitschrift „*Electronics*“ fest, dass sich bis zu diesem Zeitpunkt die Anzahl der Transistoren pro IC jedes Jahr in etwa verdoppelt hatte. Dieser als *Moore'sches Gesetz* bekannt gewordene Zusammenhang hat sich prinzipiell bis heute fortgesetzt. Lediglich die „Zeitkonstante“ muss etwas größer angesetzt werden. Sie beträgt bei Speicher-ICs mit regulärer Struktur etwa 18 Monate und bei Prozessoren wegen der erhöhten Komplexität etwa 24 Monate.

Integrierte Schaltkreise

Moore'sches Gesetz

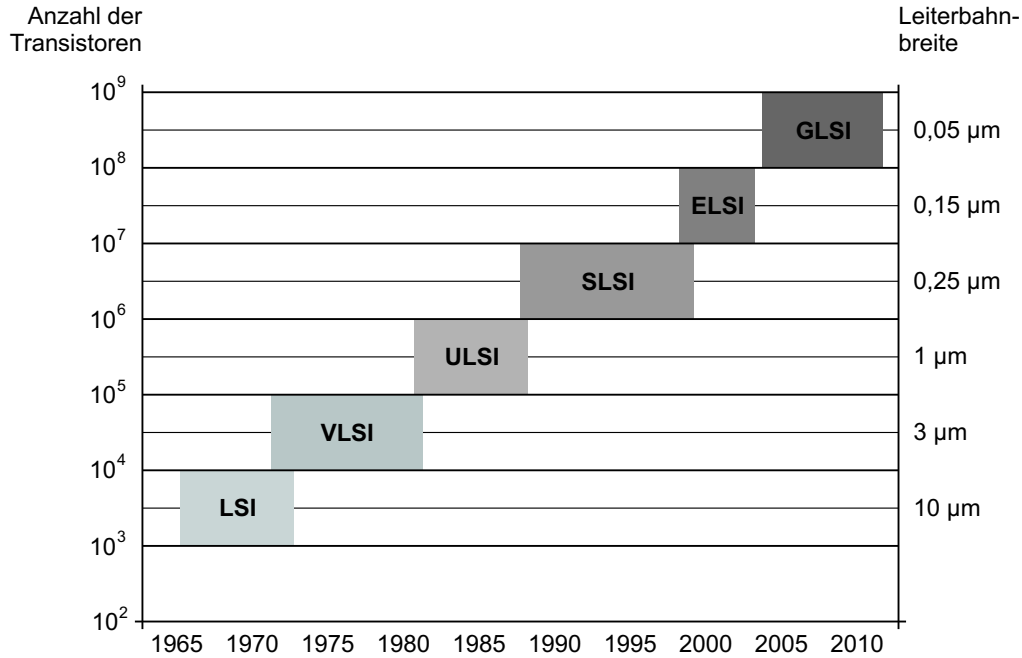


Abbildung 1.2: Entwicklung der Integrationsdichten und Strukturgrößen integrierter Schaltungen.

Um eine logische Organisation in Hardware umzusetzen, müssen mehrere Schichten geometrischer Strukturen (*Chip Layouts*) auf eine Halbleiterscheibe (*Wafer*) geätzt werden. Diese implementieren dann elektronisch die einzelnen Funktionseinheiten. Im Laufe der Jahre wurde die Verfahrenstechnik zur Her-

stellung solcher Mikrochips stetig verbessert (s. Abbildung 1.2). Heute ist man in der Lage, kleinste Strukturen von weniger als 50 Nanometer (nm) herzustellen. (1 nm bedeutet 10^{-9} m, das ist ein Millionstel eines Millimeters.) Durch die feineren Strukturen ist es möglich, immer mehr Transistoren auf einen Mikrochip zu integrieren und gleichzeitig auch die Taktfrequenzen zu erhöhen. Wegen der kleineren Strukturen können die Transistoren schneller schalten und durch kürzere Verbindungsleitungen miteinander gekoppelt werden. Dadurch verkürzen sich auch die Laufzeiten zwischen den Transistoren.

Strukturgrößen kleiner als 50 nm

Tabelle 1.1: Technologien integrierter Schaltungen.

Integrationsdichte	Kurzbezeichnung	Anzahl Transistoren
Small Scale Integration	SSI	100
Medium Scale Integration	MSI	1.000
Large Scale Integration	LSI	10.000
Very Large Scale Integration	VLSI	100.000
Ultra Large Scale Integration	ULSI	1.000.000
Super Large Scale Integration	SLSI	10.000.000
Extra Large Scale Integration	ELSI	100.000.000
Giga Scale Integration	GSI	$>$ 1.000.000.000

Seit 1960 entstanden verschiedene Generationen von Mikrochips (vgl. Tabelle 1.1). Während man bei der SSI-Technologie mit Strukturgrößen von um die 100 Mikrometer gerade mal 100 Transistoren auf einem Mikrochip integrieren konnte, sind heute (2007) bei der GSI-Technologie mit Strukturgrößen von ca. 50 Nanometer Integrationsdichten von fast einer Milliarde Transistoren möglich. Hauptproblem zukünftiger Entwicklungen ist vor allem die Kühlung der Mikrochips, deren Wärmedichten die von Kochplatten bei weitem übersteigen.

Problem Kühlung

Es gibt verschiedene Arten von Mikrochips, die sich bezüglich der Regelmäßigkeit der geätzten Strukturen unterscheiden. Speicherchips sind am regelmäßigsten aufgebaut und können daher auch die höchsten Integrationsdichten erreichen. Die Prozessoren erreichen weniger als die Hälfte der Integrationsdichte von Speicherbausteinen. Neben Prozessoren gibt es auch noch anwendungsspezifische Bausteine, die entweder maskenprogrammiert werden (*Application Specific Integrated Circuit* – ASIC) oder elektrisch programmierbar sind (*Field Programmable Gate Array* – FPGA). Die Hersteller dieser Bausteine verwenden häufig statt der Transistorenanzahl als Komplexitätsmaß die Zahl der Gatteräquivalente. Als Faustregel gilt, dass zur Realisierung eines Gatters etwa vier Transistoren benötigt werden. (Wenn Sie das Thema „technologische Realisierung“ näher interessiert, dann sollten Sie den Kurs 1721 belegen. Dort erfahren Sie mehr über Aufbau und Entwurfsmethoden hochintegrierter Mikrochips.)

anwendungsspezifische Mikrochips

Betrachten wir als nächstes den Zusammenhang zwischen der Befehlssatzarchitektur und den beiden darunter liegenden Schichten. Die bisher im Kurs 1608 eingeführten Prozessoren basieren auf mikroprogrammierten Steuerwerken. Diese Art der logischen Organisation hatte sich in den sechziger und siebziger Jahren entwickelt. Damals hatten die Hauptspeicher nur geringe Speicherkapazitäten und man versuchte daher, möglichst mächtige Maschinenbefehle

bereitzustellen, um die zur Lösung eines Problems benötigten Verarbeitungsschritte in einem möglichst kurzen Programm zu codieren. Gleichzeitig wollte man die Maschinenprogrammierung ähnlich komfortabel gestalten wie die Programmierung in einer höheren Programmiersprache.

Die Befehlssätze derartiger *CISC-Prozessoren* (*Complex Instruction Set Computer*) boten eine große Vielfalt an Adressierungsarten, die unterschiedlich viele Taktzyklen erforderten und nur mittels Mikroprogrammierung effizient implementiert werden konnten. Die Auslastung der einzelnen Prozessor-Funktionseinheiten war schlecht, da die Maschinenbefehle nur nacheinander abgearbeitet werden konnten. So wurde beispielsweise bei einem arithmetischen Befehl mit Speicherzugriff die ALU im Rechenwerk nur einen Taktzyklus lang genutzt, während der Befehl meist mehr als zehn Taktzyklen dauerte.

Durch Einschränkungen der Befehlssatzarchitektur – vor allem bei den Adressierungsmöglichkeiten – erreichte man mit der Einführung so genannter *RISC-Prozessoren* (*Reduced Instruction Set Computer*) eine deutliche Vereinfachung der Implementierung. Das Ziel der Architekturveränderungen bestand darin, die Implementierung *aller* Maschinenbefehle auf eine feste Anzahl von Mikroschritten (Taktzyklen) zu beschränken. Damit war man dann in der Lage, das *Pipelining*-Prinzip mit einer überlappenden Verarbeitung dieser Mikroschritte anzuwenden. Im Idealfall kann mit diesem Ansatz die Auslastung der Prozessor-Funktionseinheiten auf 100% gesteigert und in jedem Taktzyklus ein Befehl beendet werden. RISC-Prozessoren wurden zunächst *skalar* ausgelegt, d.h. es gab nur eine ALU für die Ausführungsstufe. Wir werden uns in diesem Kapitel ausführlich mit einen Vertreter dieser skalaren RISC-Prozessoren befassen. Indem man mehrere ALUs (bzw. auch Gleitkommeinheiten) in der Ausführungsstufe hinzufügte, entstanden so genannte *Superskalar*-(RISC-)Prozessoren, die gleichzeitig mehrere Befehle holen, verplanen (*Scheduling*) und parallel ausführen. Auf diese Weise können mit jedem Taktzyklus mehrere Befehle beendet werden.

skalare RISC

Während bei superskalaren RISC-Prozessoren die Verplanung der Befehle in einer entsprechenden Pipelinestufe mittels Hardware erfolgt, entstanden auch Prozessoren mit parallel arbeitenden Ausführungseinheiten, die auf einem *statischen* Scheduling basieren. Hier wird das Scheduling nicht zur Laufzeit sondern vom Compiler ausgeführt wird. Man spricht von VLIW-Prozessoren (*Very Long Instruction Word*), weil der Compiler sehr breite Maschinenbefehlcodes erzeugt, die dann im Prozessor zur Ansteuerung der einzelnen Ausführungseinheiten genutzt werden. Eine Kombination von statischem und dynamischem Scheduling bildet das EPIC (*Explicitly Parallel Instruction Computing*), das für die IA64-Architektur des Intel Itanium Prozessors entwickelt wurde. Die zentrale Idee besteht darin, durch den Compiler das Hardware-Scheduling zu unterstützen und so den Hardwareaufwand im Prozessor zu verringern. Die Konzepte superskalarer RISC-, VLIW- und EPIC-Prozessoren werden im Kapitel 2 dieses Kurses ausführlicher behandelt. Im Folgenden beschränken wir uns auf die Konzepte skalarer RISC-Prozessoren.

superskalare RISC

statisches Scheduling bei VLIW

1.2 Befehlssatzarchitekturen

1.2.1 Prozessor– und Mikroarchitektur, Programmiermodell

Prozessorarchitektur

Eine **Prozessorarchitektur** definiert die Grenze zwischen Hardware und Software. Sie umfasst den für den Systemprogrammierer und für den Compiler sichtbaren Teil des Prozessors. Synonym wird deshalb oft auch von der **Befehlssatz-Architektur** (*Instruction Set Architecture* – ISA) oder dem **Programmiermodell** eines Prozessors gesprochen. Dazu 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 Erscheinungsbild. Die internen Vorgänge werden ausgeklammert.

Mikroarchitektur

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. Die Art und Stufenzahl des Befehls-Pipelining, der Grad der Verwendung der Superskalartechnik, Art und Anzahl der internen Ausführungseinheiten eines Mikroprozessors sowie Einsatz und Organisation von Cache-Speichern[1] zählen zu den Mikroarchitekturtechniken. 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.

Prozessortechniken

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 binärkompatibel zueinander. Die Implementierungstechniken zeigen sich bei den unterschiedlichen Verarbeitungsgeschwindigkeiten.

Prozessorfamilie

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, doch nicht umgekehrt.

Programmiermodell

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

[1] Unter einem Cache oder Cache-Speicher versteht man einen kleinen, schnellen Pufferspeicher, in dem Kopien derjenigen Teile des Hauptspeichers gepuffert werden, auf die aller Wahrscheinlichkeit nach vom Prozessor als nächstes zugegriffen wird. Näheres dazu erfahren Sie im Unterabschnitt 3.1.3.

- Wie werden Daten repräsentiert?
- Wo werden die Daten gespeichert?
- Welche Operationen 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. Im Folgenden werden diese Eigenschaften im Einzelnen kurz vorgestellt.

1.2.2 Datenformate

Wie in den höheren Programmiersprachen können auch in maschinennahen Sprachen die Daten verschiedenen Datenformaten zugeordnet werden. Diese bestimmen, wie die Daten repräsentiert werden.

Datenlängen

Bei der Datenübertragung zwischen Speicher und Register 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. MOVB, um ein

Mikrocontroller

Byte zu übertragen. 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) angewandt.

n-Bit-Prozessor

Ein 32-Bit-Datenwort reflektiert dabei die Sicht eines 32-Bit-Prozessors bzw. ein 16-Bit-Datenwort diejenige eines 16-Bit-Prozessors oder -Mikrocontrollers. (Wir sprechen in diesem Kurs 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 32 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 alle Datenlängen von 8 Bit bis hin zu 256 Bit aufweisen. Das Besondere ist dabei, dass einzelne Bits eines solchen Worts manipuliert werden können.

Ganzzahldatenformate

Ganzzahldatenformate (s. Abbildung 1.3) 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

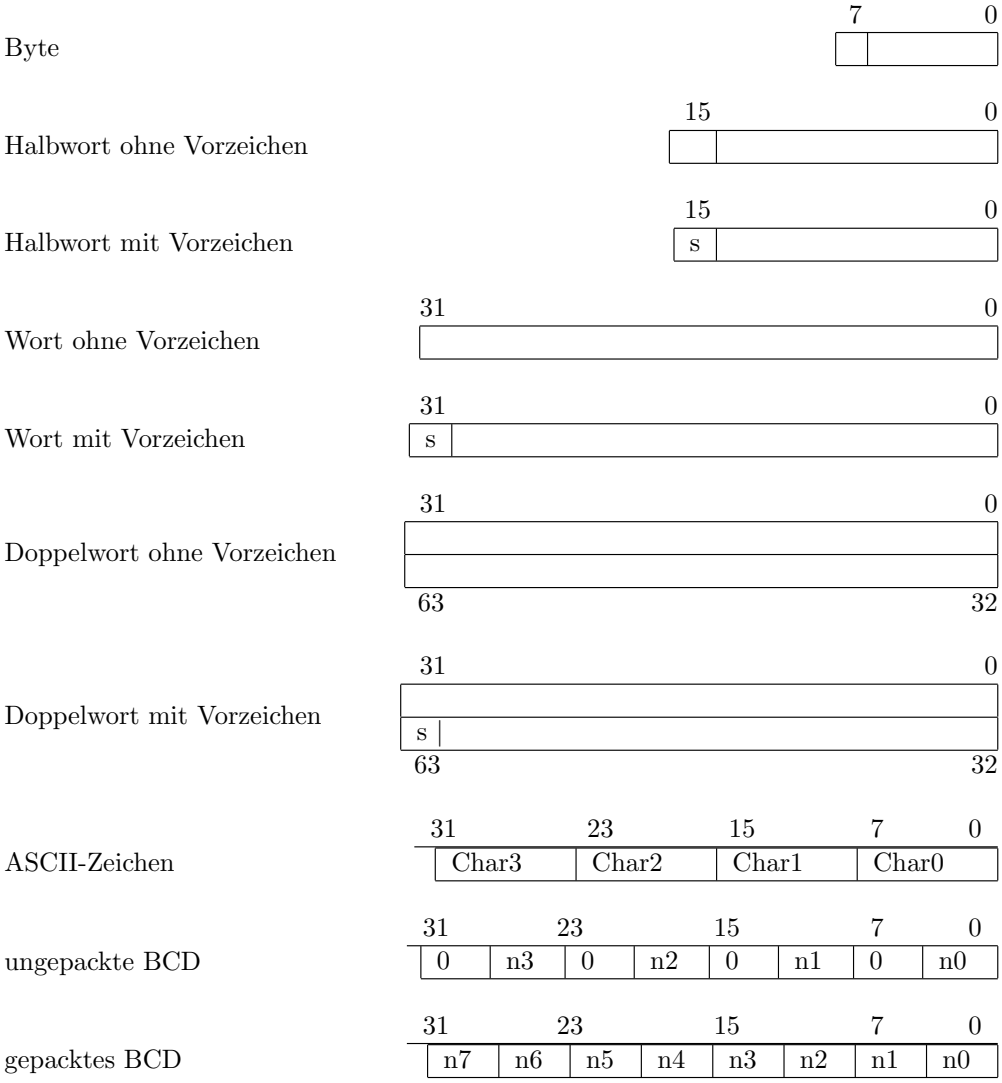


Abbildung 1.3: Ganzzahlendatenformate.

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, sodass vier ASCII-codierte Zeichen in einem 32-Bit-Wort untergebracht werden.

Gleitkomma–datenformate

Die Gleitkommadatenformate (s. Abbildung 1.4) wurden mit dem 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.

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

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

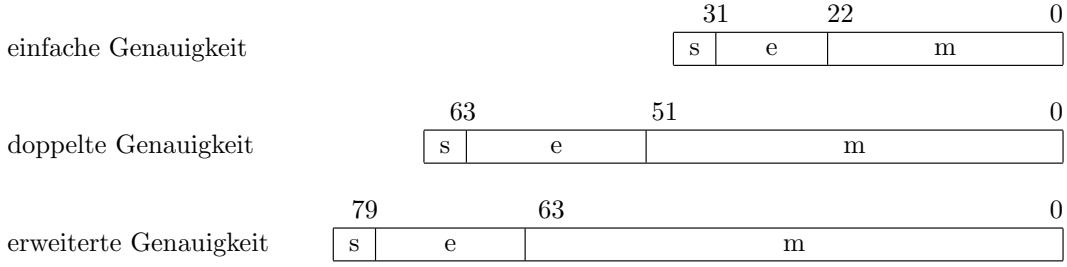


Abbildung 1.4: Gleitkomma-Datenformate.

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 (s. Abbildung 1.4). Für die Mantisse m gilt:

$m = .m_1 \cdots m_p$ 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.

Den richtigen Exponenten E erhält man aus der Gleichung:

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

Multimediadatenformate

Multimediadatenformate definieren 64 oder 128 Bit breite Wörter. Man unterscheidet zwei Arten von Multimediadatenformaten (s. Abbildung 1.5): Die bitfeldnorientierten 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.

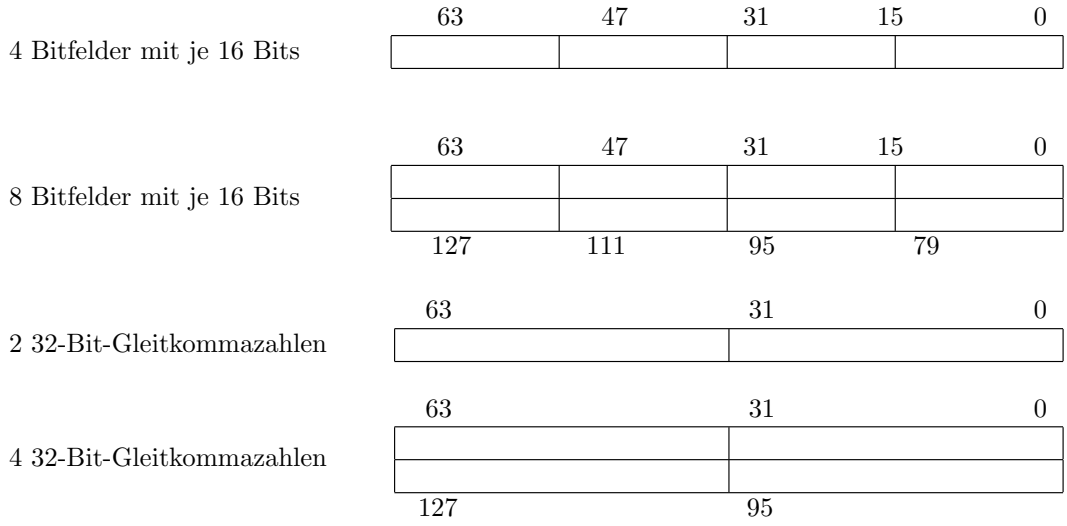


Abbildung 1.5: Beispiele bitfeld- und graphikorientierter Multimediadaten.

1.2.3 Adressraumorganisation

Registerarten

Die Adressraumorganisation bestimmt, wo die Daten gespeichert werden. Jeder Prozessor enthält eine kleine Anzahl von **Registern**, d.h. schnellen 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 Register zurückgespeichert. 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,..., 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. Die Multimediaregister stehen für sich oder sind mit den Gleitkommaregistern identisch. All diese für den Programmierer ansprechbaren Register werden als **Architekturregister** bezeichnet, da sie in der „Architektur“ sichtbar sind. Im Gegensatz dazu sind die auf heutigen Mikroprozessoren oft vorhandenen physikalischen Register oder Umbenennungspufferregister (vgl. Abschnitt 2.2.3 in Kapitel 2) für den Programmierer in der Architektur nicht erkennbar.

Um Daten zu speichern, werden mehrere Adressbereiche unterschieden. Diese sind neben den Registern und Spezialregistern insbesondere Speicheradressbereiche 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.

Wortadresse:	x0				x4			
Bytestelle im Wort:	7	6	5	4	3	2	1	0

Wortadresse:	x0				x4			
Bytestelle im Wort:	0	1	2	3	4	5	6	7

Abbildung 1.6: Big-endian- (oben) und Little-endian-Formate (unten).

Adressierbarkeit

Abgesehen von den Registern werden alle anderen Adressbereiche meist auf einen einzigen, durchgehend adressierten Adressraum abgebildet. Dieser kann **byteadressierbar** sein, d.h. jedes Byte in einem Speicherwort kann einzeln adressiert werden. Oft sind heutige Prozessoren jedoch **wortadressierbar**, sodass 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 zu einem Speicherwort mit einer Länge von n Bytes ab der Speicheradresse A heißt ausgerichtet, wenn A modulo $n = 0$ gilt, d.h. der Rest der ganzzahligen Division von A durch n den Wert 0 ergibt.

Ein 64-Bit-Wort umfasst 8 Bytes und benötigt auf einem byteadressierbaren Prozessor 8 Speicheradressen. Für die Speicherung im Hauptspeicher unterscheidet man zwei Arten der Byteanordnung innerhalb eines Worts (s. Abbildung 1.6):

Little- und Big-endian-Format

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

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

1.2.4 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: Befehlssatz

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 Worts um eine Anzahl von Stellen entweder nach links oder nach rechts bzw. rotieren die Bits nach links oder rechts (s. Abbildung 1.7).

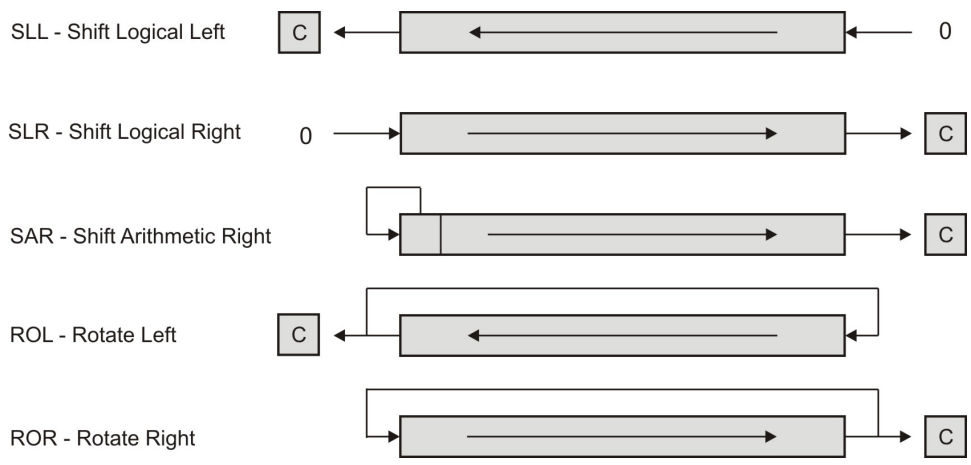


Abbildung 1.7: Einige Schiebe- und Rotationsbefehle.

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 im Zweierkomplement 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 so genann-

tes 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 (s. Abbildung 1.8). Man unterscheidet zwei Arten von Multimediabefehlen: bitfeldorientierte und graphikorientierte Multimediabefehle.

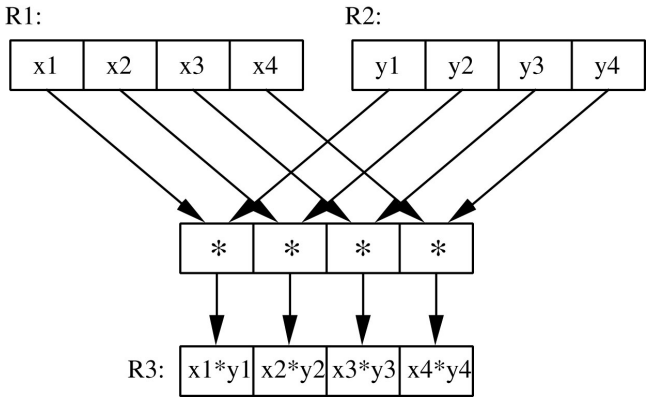


Abbildung 1.8: Grundprinzip einer Multimediaoperation.

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.

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[3]. Weiterhin gehören der HALT-Befehl zum Anhalten des Prozessors und

[3]Näheres zu Caches und Speicherverwaltungseinheiten finden Sie im Abschnitt 3.1.

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.

1.2.5 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: Adressformate

- 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 so genannte 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 Stackarchitektur (s.u.).

Befehlssatz-architekturen

Die Adressformate hängen eng mit folgender 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 so genanntes **Akkumulatorregister**, das bei arithmetisch-logischen Befehlen immer implizit eine Quelle und das Ziel darstellte, sodass **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 so genannten **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 1.2 zeigt, wie ein Zweizeilenprogramm in Pseudo-Assemblerbefehle für die vier Befehlssatz-Architekturen übersetzt 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.

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 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.

Tabelle 1.2: Programm C=A+B; D=C-B; in den vier Befehlsformatsarten codiert.

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

Die Befehlscodierung kann eine feste oder eine variable Befehlslänge festlegen. Um die Decodierung 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.

1.2.6 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.

Zusammenhang: effektive, logische und physikalische Adresse

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 Adressrechnung. In modernen Prozessoren, die eine virtuelle Speicherverwaltung[4] anwenden, wird die effektive Adresse als so genannte 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.

[4] Näheres zur virtuellen Speicherverwaltung finden Sie in Unterabschnitt 3.1.4

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:

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

Bei der Registeradressierung steht der Operand in einem Register und bei der unmittelbaren Adressierung steht der Operand direkt im Befehlswort. In beiden Fällen sind weder Adressrechnung noch ein zusätzlicher Speicherzugriff nötig.

Bei der einstufigen Speicheradressierung steht der Operand im Speicher und für die effektive Adresse ist nur *eine* Adressrechnung notwendig. Diese Adressrechnung 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 Adressrechnung wiederum auf den Speicher zugegriffen wird, um einen weiteren Datenwert für die Adressrechnung 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.

Bei der **Registeradressierung** (*Register*) steht der Operand direkt in einem Register (s. Abbildung 1.9).

Bei der **unmittelbaren Adressierung** (*immediate* oder *literal*) steht der Operand als Konstante direkt im Befehlswort (s. Abbildung 1.10).

Bei der **direkten** oder **absoluten Adressierung** (*direct, absolute*) steht die Adresse eines Speicheroperanden im Befehlswort (s. Abbildung 1.11).

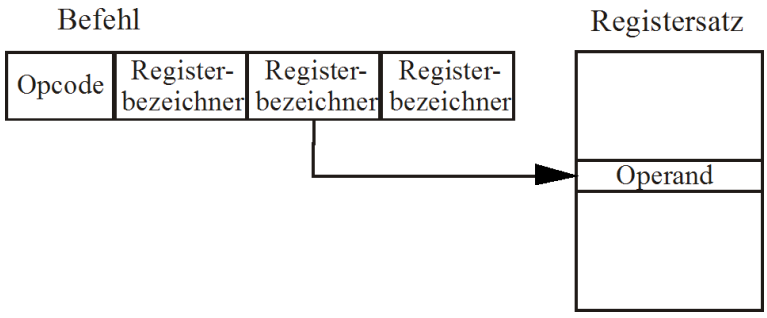


Abbildung 1.9: Registeradressierung.

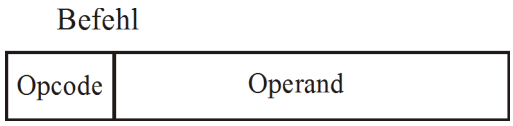


Abbildung 1.10: Unmittelbare Adressierung

Bei der **registerindirekten Adressierung** (*Register indirect* oder *Register deferred*) steht die Operandenadresse in einem Register. Der Inhalt des Registers dient als Zeiger auf eine Speicheradresse (s. Abbildung 1.12).

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. In Abbildung 1.12 ist dies die „Speicheradresse“, die vor oder nach dem Benutzen dieser Adresse um die Länge des adressierten Operanden inkrementiert oder dekrementiert wird. 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.

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 (*Dis-*

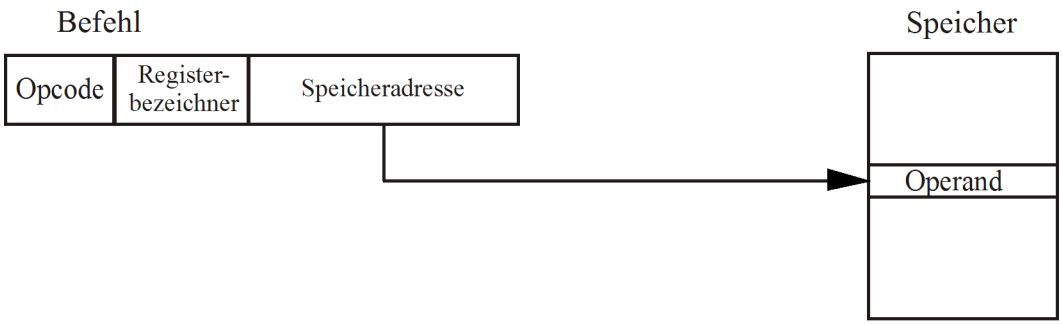


Abbildung 1.11: Direkte Adressierung

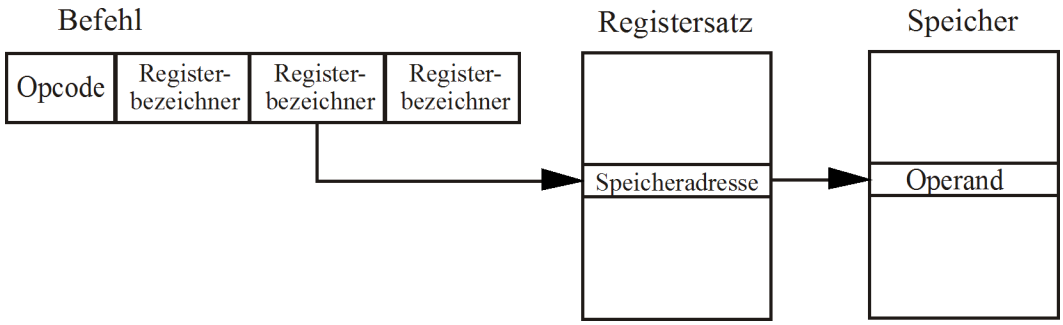


Abbildung 1.12: Registerindirekte Adressierung

placement), d.h. eines konstanten, vorzeichenbehafteten Werts, der im Befehl steht (s. Abbildung 1.13).

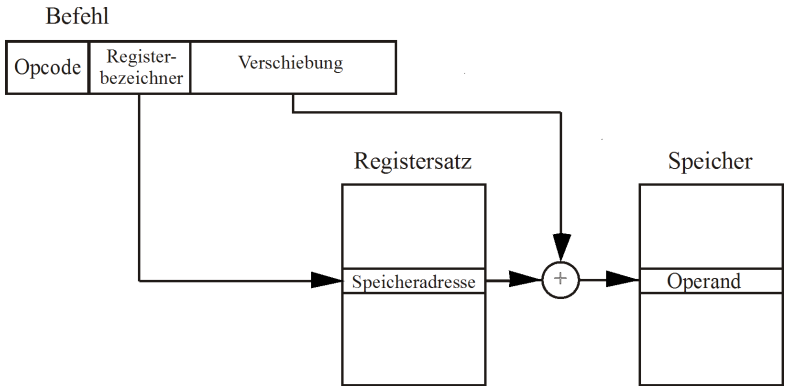


Abbildung 1.13: Registerindirekte Adressierung mit Verschiebung.

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 (s. Abbildung 1.14).

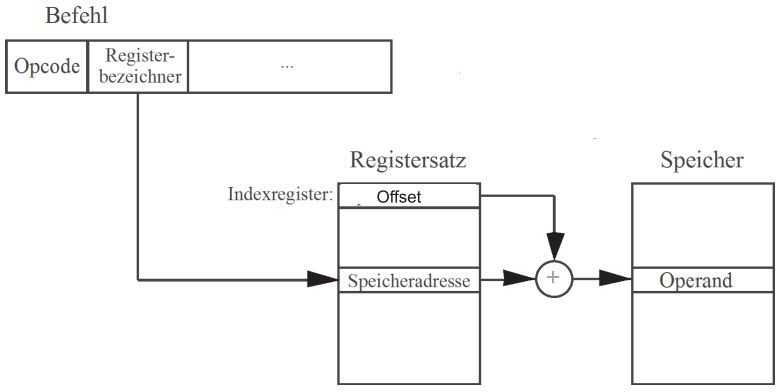


Abbildung 1.14: Indizierte Adressierung.

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 (s. Abbildung 1.15).

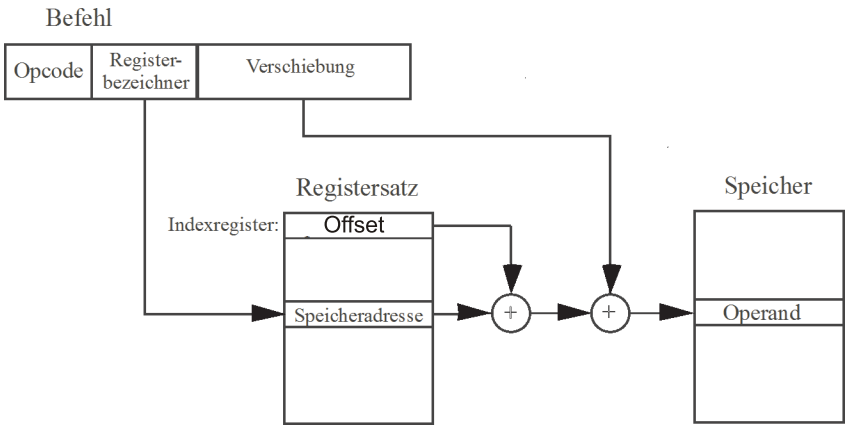


Abbildung 1.15: Indizierte Adressierung mit Verschiebung.

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

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, sodass nur wenige Bits für den Verschiebewert im Befehl benötigt werden (s. Abbildung 1.16).

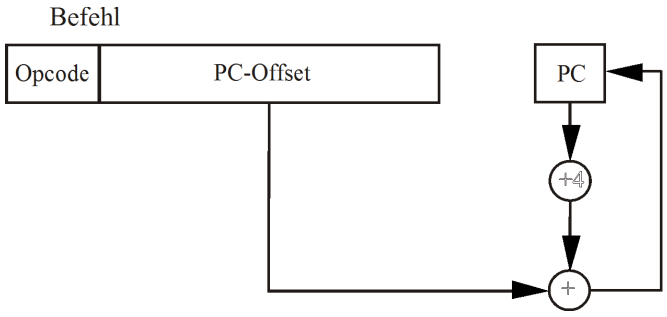


Abbildung 1.16: Befehlszählerrelative Adressierung.

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 (s. Abbildung 1.17).

Tabelle 1.3 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.

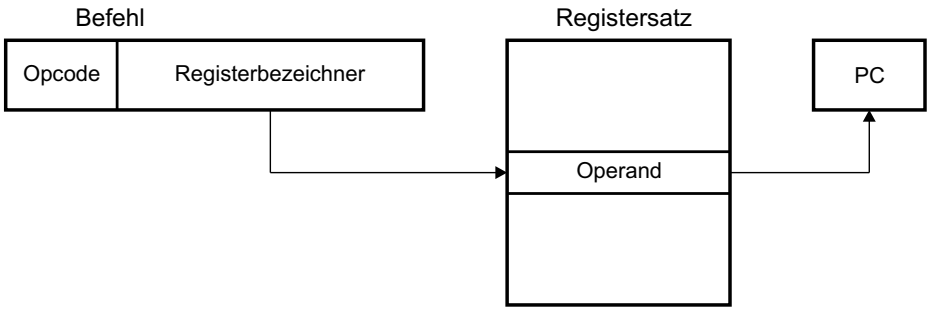


Abbildung 1.17: Befehlszählerindirekte Adressierung.

Tabelle 1.3: Adressierungsarten.

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

Es gibt darüber hinaus eine ganze Anzahl weiterer Adressierungsarten, die beispielsweise in [1], [2] und [6] beschrieben sind.

Selbsttestaufgabe 1.1 (Registerindirekte Adressierung) *Welche einfacheren Adressierungsarten lassen sich durch die registerindirekte Adressierung mit Verschiebung ersetzen?*

Lösung auf Seite 61

1.2.7 CISC- und RISC-Prinzipien

Bei der Entwicklung der Großrechner in den 60er und 70er Jahren hatten technologische Bedingungen wie der teure und langsame Hauptspeicher[5] zu einer immer größeren Komplexität der Rechnerarchitekturen geführt. Um den teuren Hauptspeicher optimal zu nutzen, wurden komplexe Maschinenbefehle entworfen, die mehrere Operationen mit einem Opcode codieren. Damit konnte auch der im Verhältnis zum Prozessor langsame Speicherzugriff überbrückt werden, denn ein Maschinenbefehl umfasste genügend Operationen, um die Zentraleinheit für mehrere, eventuell sogar mehrere Dutzend Prozessortakte zu beschäftigen.

Mikroprogrammierung

Eine auch heute noch übliche Implementierungstechnik, um den Ablauf komplexer 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 (vgl. KE4 von Kurs 1608).

CISC und RISC

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. Ähnliche Architekturcharakteristika zeigten sich auch bei den Intel-80x86- und den Motorola-680x0-Mikroprozessoren, die ab Ende der 70er Jahre entstanden. Etwa 1980 entwickelte sich ein 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 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. Das komplexe Steuerwerk, das notwendig war, um einen großen Befehlssatz in Hardware zu implementieren, benötigte viel Chip-Fläche und führte zu langen Entwicklungszeiten.

Das RISC-Architekturkonzept wurde Ende der 70er Jahre mit dem Ziel entwickelt, durch vereinfachte Architekturen Rechner schneller und preisgünstiger zu machen. Die Speichertechnologie war billiger geworden, erste Mikroprozessoren konnten (bereits seit Beginn der 1970er Jahre) auf Silizium-Chips implementiert werden. Einfache Maschinenbefehle ermöglichten es, bei der Befehlsausführung das Pipelining-Prinzip anzuwenden: Möglichst alle Befehle sollten dabei so implementierbar sein, dass pro Prozessortakt die Ausführung eines Maschinenbefehls in der Pipeline beendet wird. Durch die Implementierung mittels einer Befehlspipeline kann (für die meisten Befehle) auf die Mikroprogrammierung verzichtet werden. Stattdessen werden die Befehle (Opcodes) durch ein schnelles Schaltnetz in einem einzigen Taktzyklus decodiert. (Nur sehr kom-

[5] Es gab noch keine Cache-Speicher!

plexe Befehle, wie z.B. bei den PC-Prozessoren von Intel oder AMD, werden weiterhin durch ein Mikroprogramm-Steuerwerk implementiert.)

Dies war natürlich nur durch eine konsequente Verschlankung der Prozessorarchitektur möglich. 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 Registerzahl von mindestens 32 allgemein verwendbaren Registern ist vorhanden.

- Der Zugriff auf den Speicher 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 Implementierungstechnik des Befehls-Pipelining wurde damals zur Architektur hin offen gelegt. Eine klare Trennung von (Befehlssatz-)Architektur und Mikroarchitektur war für diese Rechner nicht möglich. Das gilt für heutige Mikroprozessoren – abgesehen von wenigen Ausnahmen – nicht mehr, jedoch ist die Beachtung der Mikroarchitektur für Compiler-Optimierungen auch weiterhin notwendig.

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 als Pipelines aufgebaut sind, können sie jedoch ebenfalls pro Takt ein Resultat liefern.

Skalar und superskalar

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* Befehlsausführung pro Takt stehen zu bleiben. Die Superskalartechnik ermöglicht es heute, pro Takt mehreren Ausführungseinheiten gleichzeitig bis zu sechs Befehle zuzuordnen und eine gleiche Anzahl von Befehlsausführungen pro Takt zu

beenden. Solche Prozessoren werden als **superskalare (RISC)-Prozessoren** bezeichnet, da die oben definierten RISC-Charakteristika auch heute noch weitgehend beibehalten werden. Solche hochperformante Prozessoren werden in der nächsten Kurseinheit ausführlich vorgestellt.

Im Folgenden werden wir zunächst die skalaren RISC-Prozessoren behandeln.

1.3 Beispiele für RISC-Architekturen

1.3.1 Das Berkeley RISC-Projekt

Das 1980 an der Universität in Berkeley begonnene RISC-Projekt ging von der Beobachtung aus, dass Compiler den umfangreichen Befehlssatz eines CISC-Rechners nicht optimal nutzen können. Daher wurden anhand von compilierten Pascal- und C-Programmen für VAX-, PDP-11- und Motorola-68000-Prozessoren zunächst die Häufigkeiten einzelner Befehle, Adressierungsmodi, lokaler Variablen, Verschachtelungstiefen usw. untersucht. Es zeigten sich zwei wesentliche Ergebnisse:

Ein hoher Aufwand entsteht bei häufigen Prozeduraufrufen oder Kontextwechseln durch Sicherung von Registern und Prozessorstatus und durch Parameterübergabe. Diese geschieht über den Laufzeitstapel im Haupt- bzw. Cache-Speicher und ist dementsprechend langsam. Wesentlich sinnvoller wäre jedoch eine Parameterübergabe in den Registern, was jedoch durch die Gesamtzahl der Register begrenzt ist. Für die Berkeley RISC-Architektur wurde daher eine neue Registerorganisation – die Technik der überlappenden Registerfenster – entwickelt, die sich heute noch in den SPARC-, SuperSPARC- und UltraSPARC-Prozessoren von Sun findet und den Datentransfer zwischen Prozessor und Speicher bei einem Prozeduraufruf minimiert. Der Registersatz mit 78 Registern ist durch seine Fensterstruktur (*Windowed Register Organization*) gekennzeichnet, wobei zu jedem Zeitpunkt jeweils nur eine Teilmenge des Registerblocks sichtbar ist. Als Konsequenz ergibt sich eine effiziente Parameterübergabe für eine kleine Anzahl von Unterprogrammaufrufen. Das Prinzip der überlappenden Registerfenster wird in Abschnitt 3.1.2 ausführlich beschrieben.

RISC-I-Prozessor

Der RISC-I-Prozessor wurde als 32-Bit-Architektur konzipiert; eine Unterstützung von Gleitkommaarithmetik und Betriebssystemfunktionen war nicht vorgesehen. Der Befehlssatz bestand aus nur 31 Befehlen mit einer einheitlichen Länge von 32 Bit und nur zwei Befehlsformaten. Die Befehle lassen sich folgendermaßen einteilen:

- 12 arithmetisch-logische Operationen,
- 9 Lade/Speicheroperationen und
- 10 Programmsteuerbefehle.

Der RISC I verfügt nur über drei Adressierungsarten:

- Registeradressierung `Rx`: Der Operand befindet sich im Register `x`.

- Unmittelbare Adressierung `#num`: Der Operand `#num` ist als 13-Bit-Zahl unmittelbar im Befehlswort angegeben.
- Registerindirekte Adressierung mit Verschiebung `Rx+#displ`: Der Inhalt eines Registers `Rx` wird als Adresse interpretiert, zu der eine 13-Bit-Zahl `#displ` addiert wird. Das Ergebnis der Addition ist die Speicheradresse, in der der Operand zu finden ist.

Durch die Verwendung von `R0+#displ` (Register `R0` enthält immer den Wert Null und kann nicht überschrieben werden) wird die direkte oder absolute Adressierung einer Speicherstelle gebildet. Setzt man für ein Register Ri, i≠0, die Verschiebung `#displ` zu Null, erhält man eine registerindirekte Adressierung.

Im Oktober 1982 war der erste Prototyp des RISC I fertig. Der Prozessor bestand aus etwa 44.500 Transistoren. Auffällig war der stark verminderte Aufwand von nur 6% Chip-Flächenanteil für die Steuerung gegenüber mehr als 50% bei CISC-Architekturen. Der mit einer Taktfrequenz von 1,5 MHz schnellste RISC-I-Chip, der gefertigt wurde, erreichte die Leistung kommerzieller Mikroprozessoren.

Die Entwicklung des Nachfolgemodells RISC II wurde 1983 abgeschlossen. Die Zahl der Transistoren wurde auf 41.000 und die Chip-Fläche sogar um 25% verringert. Die beiden Prozessoren RISC I und II waren natürlich keineswegs ausgereift.

Die wesentliche Weiterentwicklung der RISC-I- und RISC-II-Rechner geschah jedoch durch die Firma Sun, die Mitte der 80er Jahre die SPARC-Architektur definierte, welche mit den SuperSPARC- und heute den UltraSPARC-Prozessoren fortgeführt wird.

1.3.2 Die DLX-Architektur

In diesem Abschnitt wird die Architektur des DLX-Prozessors (DLX steht für „Deluxe“) eingeführt. Dieser ist ein hypothetischer Prozessor, der von Hennessy und Patterson für Lehrzwecke entwickelt wurde und auch im vorliegenden Kapitel als Referenzprozessor dient. Der DLX kann als idealer, einfacher RISC-Prozessor charakterisiert werden, der sehr eng mit dem MIPS-Prozessor verwandt ist.

Registersatz des DLX

Die Architektur enthält 32 jeweils 32 Bit breite Universalregister (*General-Purpose Register* – GPR) `R0,..., R31`, wobei der Wert von Register `R0` immer Null ist. Dazu gibt es einen Satz von Gleitkommaregistern (*Floating-Point Registers FPRs*), die als 32 Register einfacher (32 Bit breiter) Genauigkeit (`F0, F1,..., F31`) oder als 16 Register doppelter (64 Bit breiter) Genauigkeit (`F0, F2,..., F30`) nach IEEE 754-Format genutzt werden können. (Dabei werden jeweils zwei 32-bit-Register R(2i), R(2i+1) zu einem 64-bit-Register verbunden.) Es gibt einen Satz von Spezialregistern für den Zugriff auf Statusinformationen. Das Gleitkomma-Statusregister wird für Vergleiche und zum Anzeigen von Ausnahmesituationen verwendet. Alle Datentransporte zum bzw. vom Gleitkomma-Statusregister erfolgen über die Universalregister.

Speicherorganisation des DLX

Der Speicher ist byteadressierbar im Big-endian-Format mit 32-Bit-Adressen. Man beachte, dass daher der Befehlszähler immer automatisch nach jedem Befehl um 4 erhöht wird, auch im Falle der Verzweigungsbefehle. Alle Speicherzugriffe müssen ausgerichtet sein und erfolgen mittels der Lade-/Speicherbefehle zwischen Speicher und Universalregistern oder Speicher und Gleitkommaregistern. Weitere Transportbefehle erlauben es, Daten zwischen Universalregistern und Gleitkommaregistern zu verschieben. Der Zugriff auf die Universalregister kann byte-, halbwort- (16 Bit) oder wortweise (32 Bit) erfolgen. Auf die Gleitkommaregister kann mit einfacher oder doppelter Genauigkeit zugegriffen werden.

Befehlsformate des DLX

Alle Befehle sind 32 Bit lang und müssen im Speicher ausgerichtet sein. Dabei ist das Opcode-Feld 6 Bit breit; die Quell- `(rs1, rs2)` und Zielregisterangaben (`rd`) benötigen 5 Bit, um die 32 Register eines Registersatzes zu adressieren. Für Verschiebewerte (*Displacement*) und unmittelbare Konstanten (*immediate*) sind 16-Bit-Felder vorgesehen. Befehlszählerrelative Verzweigungsadressen (*PC-Offset*) können 26 Bits lang sein. Ein Befehl ist in einem der folgenden drei Befehlsformate codiert:

- I-Typ: zum Laden und Speichern von Bytes, Halbwörtern und Wörtern, für alle Operationen mit unmittelbaren Operanden, bedingte Verzweigungsbefehle sowie unbedingte Sprungbefehle mit Zieladresse in einem Universalregister (JR, JALR).

6 Bits	5 Bits	5 Bits	16 Bits
Opcode	rs1	rd	immediate

- R-Typ: für Register-Register-ALU-Operationen, wobei das 11-Bit-func-Feld die Operation codiert, und für die Transportbefehle.

6 Bits	5 Bits	5 Bits	5 Bits	11 Bits
Opcode	rs1	rs2	rd	func

- J-Typ: für Jump- (J), Jump-and-Link- (JAL), Trap- und RFE-Befehle.

6 Bits	26 Bits
Opcode	PC-Offset

Es gibt vier Klassen von Befehlen: Transportbefehle, arithmetisch-logische Befehle, Verzweigungen und Gleitkommabefehle.

Transportbefehle des DLX

Für die Transportbefehle, also die Lade-/Speicherbefehle (s. Tabelle 1.4) gibt es nur die Adressierungsart „registerindirekt mit Verschiebung“ in der Form „Universalregister + 16-Bit-Verschiebewert mit Vorzeichen“, doch können daraus die registerindirekten Adressierungsarten (Verschiebewert = 0) und die absoluten Adressierungsarten (Basisregister: R0) erzeugt werden.

Tabelle 1.4: Transportbefehle.

Opcode	Bedeutung
LB, LBU	Laden eines Bytes, Laden eines Bytes vorzeichenlos (U – unsigned),
SB	Speichern eines Bytes (B)
LH, LHU	Laden eines Halbworts (H), Laden eines Halbworts vorzeichenlos,
SH	Speichern eines Halbworts
LW, SW	Laden eines Worts (W), Speichern eines Worts (von/zu Universalregistern)
LF, LD	Laden eines einfach (F) bzw. doppelt (D) genauen Gleitkommaregisters,
SF, SD	Speichern eines einfach bzw. doppelt genauen Gleitkommaregisters
MOVI2S, MOVS2I	Transport von einem Universalregister (I) zu einem Spezialregister (S) bzw. umgekehrt
MOVF, MOVD	Transport von einem Gleitkommaregister (F) bzw. Gleitkommaregisterpaar (D) zu einem anderen Register bzw. Registerpaar
MOVFP2I, MOVI2fp	Transport von einem Gleitkommaregister (FP) zu einem Universalregister (I) bzw. umgekehrt

Arithmetisch–logische Befehle des DLX

Alle arithmetisch-logischen Befehle (s. Tabelle 1.5) sind Dreiadressbefehle mit zwei Quellregistern (oder einem Quellregister und einem vorzeichenerweiterten unmittelbaren Operanden) und einem Zielregister. Nur die einfachen arithmetischen und logischen Operationen Addition, Subtraktion, AND, OR, XOR und Schiebeoperationen sind vorhanden. Vergleichsbefehle vergleichen zwei Register auf Gleichheit, Ungleichheit, größer, größer-gleich, kleiner oder kleiner-gleich. Wenn die Bedingung erfüllt ist, wird das Zielregister auf Eins gesetzt, andernfalls auf Null.

Tabelle 1.5: Arithmetisch-logische Befehle.

Opcode	Bedeutung
ADD,ADDI, ADDU,ADDUI	Addition, Add. mit unmittelbarem (16-Bit-)Operanden (I) mit oder ohne Vorzeichen (U – *unsigned*)
SUB,SUBI, SUBU,SUBUI	Subtraktion, Subtraktion mit unmittelbarem (16-Bit-)Operanden mit oder ohne Vorzeichen
MULT,MULTU, DIV,DIVU	Multiplikation und Division mit und ohne Vorzeichen; alle Operanden sind Gleitkommaregister-Operanden und nehmen bzw. ergeben 32-Bit-Werte
AND,ANDI	logisches UND, logisches UND mit unmittelbaren Operanden
OR,ORI	logisches ODER, logisches ODER mit unmittelbaren Operanden;
XOR,XORI	Exklusiv-ODER, Exklusiv-ODER mit unmittelbaren Operanden

LHI	*Load High Immediate*-Operation: Laden der oberen Registerhälfte mit einem unmittelbaren Operanden, niedrige Hälfte wird auf Null gesetzt
SLL,SRL SRA,SRAI SLLI,SRLI	Schiebeoperationen: links-logisch (LL), rechts-logisch (RL), rechts-arithmetisch (RA) ohne und mit (I) unmittelbaren Operanden, links-logisch und rechts–logisch mit unmittelbaren Operanden
S_, S_I	*Set conditional*-Vergleichsoperation; _ kann sein: LT (less than), GT (greater than), LE (less equal), GE (greater equal), EQ (equal), NE (not equal)

Verzweigungs–befehle des DLX

Die ersten vier Verzweigungsbefehle (s. Tabelle 1.6) sind bedingt. Die Verzweigungsbedingung ist durch den Opcode des Befehls spezifiziert, wobei das Quellregister auf Null oder auf von Null verschieden getestet wird. Die Zieladresse der bedingten Sprungbefehle ist durch eine vorzeichenerweiterte 16-Bit-Verschiebung gegeben, die zum Inhalt des inkrementierten Befehlszählerregisters (PC + 4) addiert wird. Die *Jump-and-Link*-Befehle (JAL, JALR) speichern die Adresse des Befehls, der dem Sprungbefehl folgt, im Register R31. Dadurch wird ein Rücksprung an die „Unterbrechungsstelle“ im Programm erleichtert – ähnlich wie beim Aufruf eines Unterprogramms und Rücksprung in das Hauptprogramm.

Tabelle 1.6: Verzweigungsbefehle.

Opcode	Bedeutung
BEQZ, BNEZ	Verzweigen, falls das angegebene Universalregister gleich (EQZ) bzw. ungleich (NEZ) Null ist; befehlszählerrelative Adressierung (16-Bit-Verschiebung + Befehlszähler + 4)
BFPT, BFPF	Testen des Vergleichsbits im Gleitkomma-Statusregister und Verzweigen, falls Wert gleich „*true*“ (T) oder „*false*“ (F); befehlszählerrelative Adressierung (16-Bit-Verschiebung + Befehlszähler + 4)
J, JR	unbedingter Sprung mit befehlszählerrelativer Adressierung (26-Bit-Verschiebung + Befehlszähler + 4) oder mit befehlszählerindirekter Adressierung (Zieladresse in einem Universalregister)
JAL, JALR	Speichert PC+4 in R31, Ziel ist befehlszählerrelativ adressiert (JAL) oder ein Register (JALR)
TRAP	Ausnahmebefehl: Übergang zum Betriebssystem bei einer vektorisierten Adresse
RFE	Rückkehr zum Benutzerprogramm von einer Unterbrechung; Wiederherstellen des Benutzermodus

Gleitkomma–befehle des DLX

Alle Gleitkommabefehle (s. Tabelle 1.7) sind Dreiadressbefehle mit zwei Quell- und einem Zielregister. Bei jedem Befehl wird angegeben, ob es sich um einfach oder doppelt genaue Gleitkommaoperationen handelt. Die letzte-

ren können nur auf ein Registerpaar `(F0,F1), ..., (F30,F31)` angewendet werden.

Tabelle 1.7: Gleitkommabefehle.

Opcode	Bedeutung
ADDF, ADDD	Addition von einfach (F) bzw. doppelt (D) genauen Gleitkommazahlen
SUBF, SUBD	Subtraktion von einfach bzw. doppelt genauen Gleitkommazahlen
MULTF, MULTD	Multiplikation von einfach bzw. doppelt genauen Gleitkommazahlen
DIVF, DIVD	Division von einfach bzw. doppelt genauen Gleitkommazahlen
CVTF2D, CVTF2I, CVTD2F, CVTD2I, CVTI2F, CVTI2D	Konvertierungsbefehle: CVTx2y konvertiert von Typ x nach Typ y, wobei x und y einer der Grundtypen I (Integer), F (einfach-genaue Gleitkommazahl) oder D (doppelt-genaue Gleitkommazahl) ist.
_F, _D	Vergleichsoperation auf einfach-genauen (F) bzw. doppelt-genauen (D) Gleitkommazahlen: _ kann sein: LT (less than), GT (greater than), LE (less equal), GE (greater equal), EQ (equal), NE (not equal); Es wird jeweils ein Vergleichsbit im Gleitkomma-Statusregister gesetzt.

Selbsttestaufgabe 1.2 (DLX-Simulator)
Arbeiten Sie sich in den DLX-Simulator ein. Schreiben Sie ein einfaches Programm, welches das Quadrat einer Zahl berechnet, die sich im Register R5 befindet. Das Ergebnis soll im Register R6 abgespeichert werden. Schauen Sie sich dabei die Registerbelegungen im DLX-Simulator genau an.

Lösung auf Seite 61

Selbsttestaufgabe 1.3 (Eingabeaufforderung)
Über die Eingabeaufforderung sollen zunächst zwei Zahlen eingelesen werden. Schreiben Sie ein Programm:

a.) welches die Summe dieser zwei Integerzahlen berechnet und das Ergebnis ausgibt;

b.) welches die zweite Zahl von der ersten abzieht und das Ergebnis ausgibt;

c.) welches das Produkt der beiden Zahlen berechnet, wobei das Ergebnis in das Gleitkommazahlen-Format umgewandelt werden soll;

d.) welches die erste Zahl durch die zweite teilt, wobei die Zahlen vorher in Gleitkommazahlen umformatiert werden sollen.

e.) Schreiben Sie ein Programm, welches zu jeder Berechnung aus a) bis d) ein Unterprogramm besitzt, das über das Hauptprogramm aufgerufen wird.

Lösung auf Seite 61

Selbsttestaufgabe 1.4 (Programm-Eingabeaufforderung)
Die Fakultät einer Zahl ist definiert als:

$$a! = a \cdot (a-1) \cdot (a-2) \cdots 2 \cdot 1$$

Schreiben Sie ein Programm, welches eine Zahl über die Eingabeaufforderung einliest und die Fakultät dieser Zahl ausgibt.

Lösung auf Seite 65

1.4 Einfache Prozessoren und Prozessorkerne

1.4.1 Grundlegender Aufbau eines Mikroprozessors

Ein **Mikroprozessor** ist ein Prozessor, der auf einem, manchmal auch auf mehreren VLSI-Chips implementiert ist. Heutige Mikroprozessoren haben meist auch Cache-Speicher und verschiedene Steuerfunktionen auf dem Prozessor-Chip untergebracht.

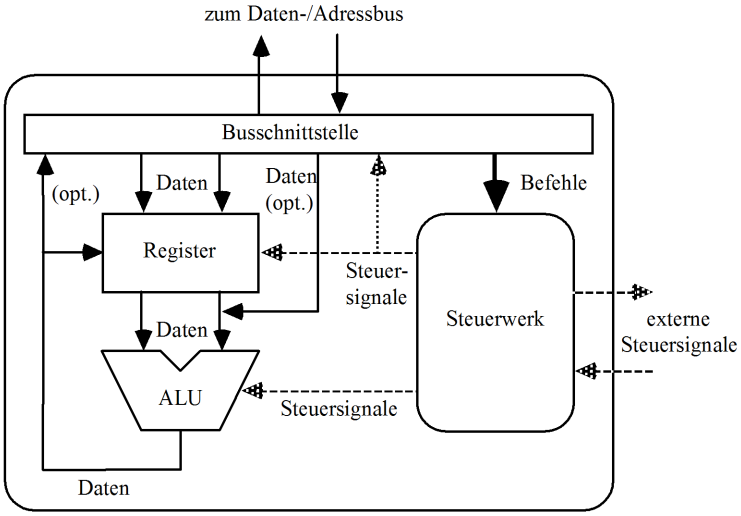


Abbildung 1.18: Struktur eines einfachen Mikroprozessors.

Abbildung 1.18 zeigt einen Mikroprozessor einfachster Bauart, wie er heute noch als Kern von einfachen Mikrocontrollern (Prozessoren zur Steuerung technischer Geräte) vorkommt. Er enthält ein Rechenwerk (oder arithmetisch-logische Einheit, *Arithmetic Logic Unit* – ALU), das seine Daten aus internen Speicherplätzen, den Registern, oder über die Busschnittstelle direkt aus dem Hauptspeicher empfängt. Das Rechenwerk führt auf den Eingabedaten arithmetisch-logische Operationen aus. Die Resultatwerte werden wieder in einem Register abgelegt oder über die Busschnittstelle in den Hauptspeicher

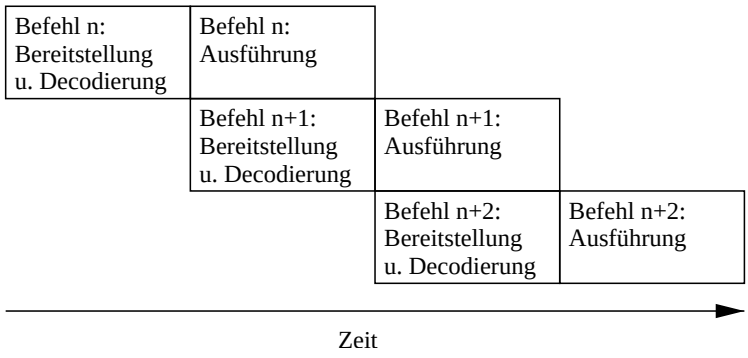


Abbildung 1.19: Überlappende Befehlsausführung.

transportiert. Die Rechenwerksoperationen werden durch Steuersignale vom Steuerwerk bestimmt. Das Steuerwerk erhält seine Befehle, adressiert durch das interne Befehlszählerregister, ebenfalls über die Busschnittstelle aus dem Hauptspeicher. Es setzt die Befehle in Abhängigkeit vom Prozessorzustand, der in einem internen Prozessor-Statusregister gespeichert ist, und eventuell auch abhängig von externen Steuersignalen in Steuersignale für das Rechenwerk, die Registerauswahl und die Busschnittstelle um.

1.4.2 Einfache Implementierungen

Ein Rechner mit von-Neumann-Prinzip benötigt zur Bearbeitung eines Befehls zwei grundlegende Phasen: eine Befehlsbereitstellungs- und eine Ausführungsphase. In der Ausführungsphase werden die Operanden bereitgestellt, in der arithmetisch-logischen Einheit (ALU) verarbeitet und schließlich wird das Resultat gespeichert.

Vom Prinzip her muss jede Befehlsbearbeitung beendet sein, bevor die nächste Befehlsbearbeitung beginnen kann. In der einfachsten Implementierung ist jede der beide Phasen abwechselnd aktiv – die beiden Phasen werden sequentiell zueinander ausgeführt. Jede dieser 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.

überlappende Befehlsverarbeitung

Jedoch können die beiden Phasen auch überlappend zueinander ausgeführt werden (s. Abbildung 1.19). Während der eine Befehl sich in seiner Ausführungsphase befindet, wird bereits der nach Ausführungsreihenfolge nächste Befehl aus dem Speicher geholt und decodiert. Diese Art der Befehlsverarbeitung ist erheblich komplizierter als diejenige ohne Überlappung, denn nun müssen Ressourcenkonflikte bedacht und gelöst werden.

Ressourcenkonflikt

Ein Ressourcenkonflikt tritt ein, wenn beide Phasen gleichzeitig dieselbe Ressource benötigen. Zum Beispiel muss in der Befehlsbereitstellungsphase über die Verbindungseinrichtung 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 er-

folgen 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 oder einer durchgeführten bedingten Verzweigung 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 verdoppelt werden.

1.4.3 Pipeline-Prinzip

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

Pipelining

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, bei der mehrere Befehle überlappt ausgeführt werden. Jede Stufe der Pipeline heißt **Pipeline-Stufe** oder **Pipeline-Segment**. Die einzelnen Pipeline-Stufen sind aus Schaltnetzen (kombinatorischen Schaltkreisen) aufgebaut, die die Funktion der Stufe realisieren.

Wichtige Merkregel: *Pipeline-Register* sollten nicht mit den Registern des Registersatzes des Prozessors verwechselt werden.

Die Pipeline-Stufen werden durch getaktete **Pipeline-Register** (auch *Latches* genannt) getrennt, welche die jeweiligen Zwischenergebnisse aufnehmen (s. Abbildung 1.20). Dabei werden alle Register vom selben Taktsignal gesteuert, sie arbeiten also synchron. Ein Pipeline-Register sollte nicht mit den Registern des Registersatzes des Prozessors verwechselt werden. Pipeline-Register sind Pipeline-interne Pufferspeicher, die das Schaltnetz, durch das eine Pipeline-Stufe realisiert wird, von der nächsten Pipeline-Stufe trennen. 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.

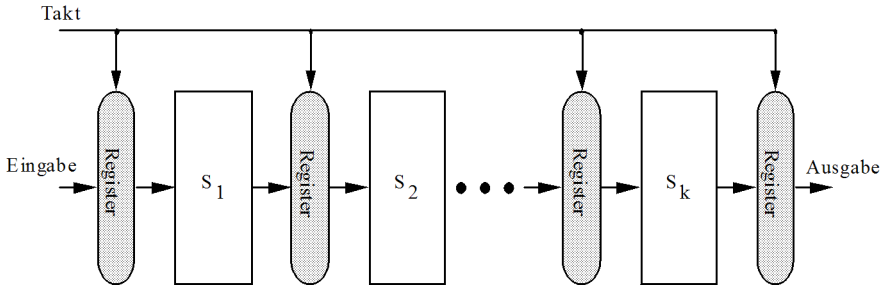


Abbildung 1.20: Pipeline-Stufen und Pipeline-Register.

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.

k-stufigen Pipeline

Man definiert die **Latenz** als die Zeit, die ein Befehl benötigt, um alle k Pipeline-Stufen zu durchlaufen.

Definition: Latenz und Durchsatz

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.

Definition: Beschleunigung

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 überlappten 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, sodass die Gattertiefe des Schaltnetzes, welches die Pipeline-Stufe implementiert, geringer wird und die Signale früher 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.

Definition: Befehls-Pipeline

Bei einer **Befehls-Pipeline** (*Instruction Pipeline*) wird die Bearbeitung eines Maschinenbefehls in verschiedene Phasen unterteilt. Aufeinanderfolgen-

de 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 pro Takt ausführen zu können.

1.5 Befehls-Pipelining

1.5.1 Grundlegende Stufen einer Befehls-Pipeline

Wie teilt sich die Verarbeitung eines Befehls in Phasen auf? Als Erweiterung des zweistufigen Konzepts aus Abschnitt 1.3.3 kann man eine Befehlsverarbeitung 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.

Randbedingungen für den Entwurf einer Befehls-Pipeline

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 Code-Cache-Speichers 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 Register-Register-Architektur (ebenfalls eine RISC-Eigenschaft), d.h. die Operanden der arithmetisch-logischen Befehle müssen nur aus den Registern des Registersatzes in die Pipeline-Register übertragen werden. Falls Speicheroperanden ebenfalls zugelassen sind, wie es für CISC-Architekturen der Fall ist, so dauert das Laden der Operanden eventuell mehrere Takte und der Pipeline-Fluss muss unterbrochen werden.

- Die Befehlsdecodierung ist für Befehlsformate einheitlicher Länge und Befehle mit geringer Komplexität so einfach, dass sie mit der Operandenbereitstellung aus den Registern eines Registersatzes zu einer Stufe zusammengefasst werden kann. In der ersten Takthälfte wird decodiert und die (Universal-)Register der Operanden werden angesprochen. In der zweiten Takthälfte werden 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 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 Resultatwerts 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.

1.5.2 Die DLX-Pipeline

Überblick über die Stufen der DLX-Pipeline

Als Beispiel einer Befehlsausführung mit Pipelining betrachten wir eine einfache Befehls-Pipeline mit den in Abbildung 1.21 dargestellten Stufen. Eine überlappte Ausführung dieser fünf Stufen 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 I-Cache-Speicher (*Instruction Cache*) in einen Befehlspuffer, das sog. Befehlsregister, geladen. Der Befehlszähler wird weitergeschaltet.

- **Decodier**- und Operandenbereitstellungsphase oder **ID-Phase** (*Instruction Decode/Register Fetch*): Aus dem Operationscode des Maschinenbefehls werden durch ein Decodierschaltnetz Pipeline-interne Steuersignale erzeugt. Die Operanden werden aus Registern (*Register File*) bereit gestellt.

- **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 D-Cache (*Data Cache*) wird durchgeführt.
- **Resultatspeicher**- oder **WB-Phase** (*Write Back*): Das Ergebnis wird in ein Register geschrieben.

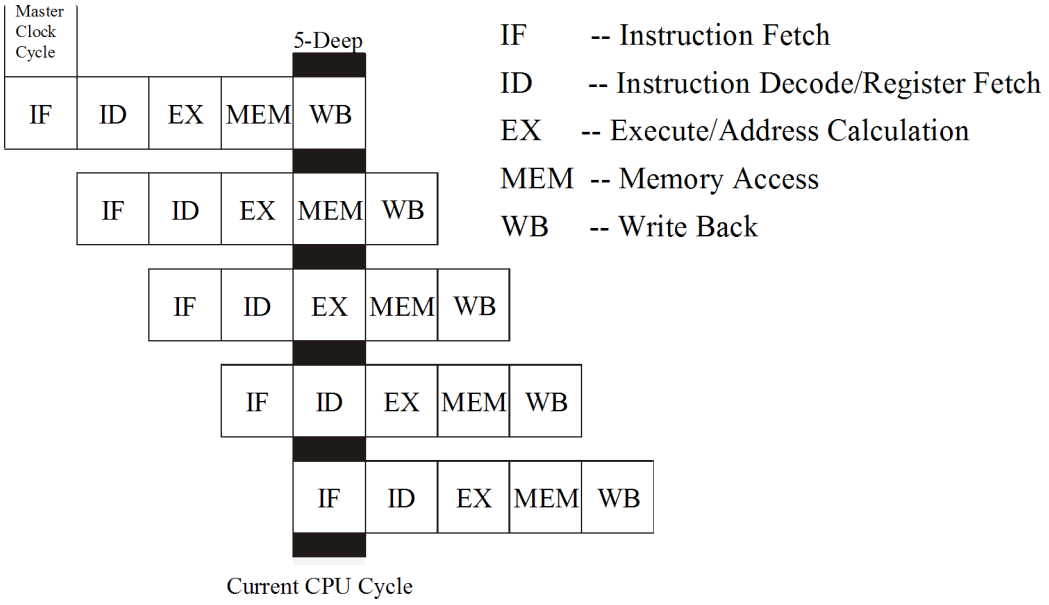


Abbildung 1.21: Grundlegendes Pipelining.

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 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.

Diese für den DLX-Prozessor vorgesehene Pipeline wurde z.B. auch im MIPS R3000-Prozessor implementiert. Heutzutage gibt es ähnlich einfache Pipelines noch in den Kernen von Digitalen Signalprozessoren (DSPs) und einigen Multimediaprozessoren.

Abbildung 1.22 zeigt detailliert die grundsätzlichen Stufen der Befehls-Pipeline, die auf eine gegenüber Abschnitt 1.3.2 vereinfachte DLX-Architektur (im Wesentlichen ohne die Gleitkommabefehle) angepasst ist.

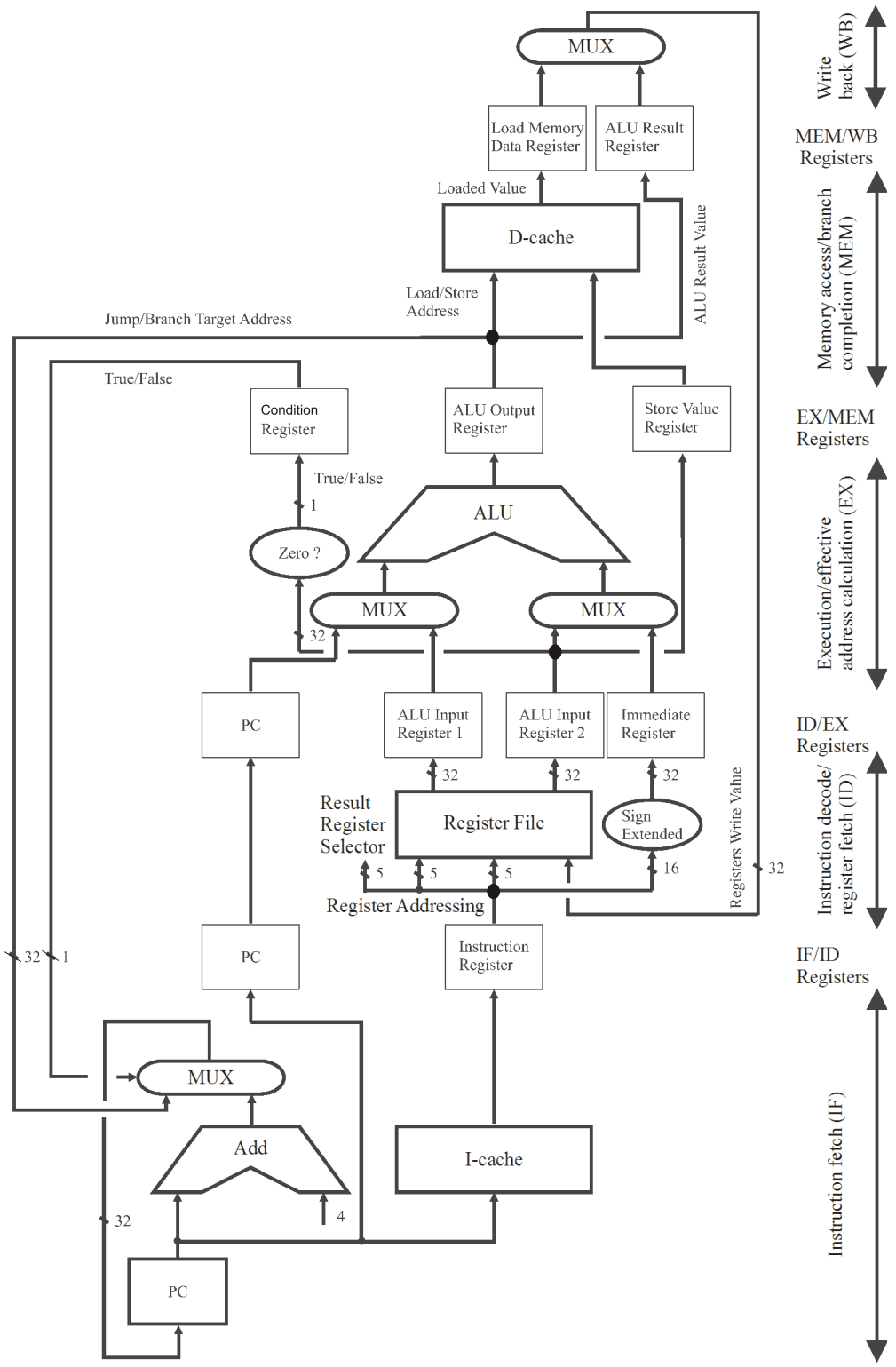


Abbildung 1.22: Implementierung einer DLX-Pipeline (ohne Gleitkommabefehle).

Funktion der Pipeline-Register

Die Pipeline-Stufen werden jeweils von mehreren **Pipeline-Registern** getrennt, die funktional verschiedene Zwischenwerte puffern. Nicht alle notwendigen Pipeline-Register sind in Abbildung 1.22 aufgeführt, um die Abbildung nicht noch komplexer werden zu lassen. Außerdem sei erwähnt, dass wegen des einheitlich 32 Bit breiten Befehlsformats und der Byteadressierbarkeit des DLX-Prozessors der Inhalt des Befehlszählers jeweils um vier erhöht werden muss, wenn auf den folgenden Befehl zugegriffen werden soll. Die in der Abbildung gezeigten, meist 32 Bit breiten Pipeline-Register besitzen die folgenden Funktionen:

- In der IF-Stufe befindet sich das Befehlszähler-Register (*Program Counter Register* – PC), das den zu holenden Befehl adressiert.
- Zwischen den IF- und ID-Stufen befindet sich ein weiteres PC-Register, das die um vier erhöhte Befehlsadresse des in der Stufe befindlichen Befehls puffert. Ein weiteres Pipeline-Register, das Befehlsregister (*Instruction Register*), enthält den Befehl selbst.
- Zwischen den ID- und EX-Stufen befindet sich erneut ein PC-Register mit der um vier erhöhten Adresse des in der Stufe befindlichen Befehls, da bei allen Mikroprozessoren diese erhöhte Befehlsadresse im Falle eines Sprungbefehls für die Berechnung der Sprungzieladresse verwendet wird. Weiterhin gibt es das erste und zweite ALU-Eingaberegister (*ALU Input Register 1* und *ALU Input Register 2*) zur Pufferung von Operanden aus dem Registersatz und ein Register (*Immediate Register*) zur Aufnahme von im Befehl angegebenen Konstanten, insbesondere auch der Anzahl der Bitstellen in Verschiebeboperationen. Durch die mit *Sign Extended* bezeichnete Einheit werden Konstanten, die kürzer sind als 32 Bit, vorzeichenrichtig auf 32 Bit erweitert.
- Zwischen den EX- und MEM-Stufen befinden sich das ein Bit breite Bedingungsregister (*Condition Register*), das zur Aufnahme eines Vergleichsergebnisses dient, das ALU-Ausgaberegister (*ALU Output Register*) zur Aufnahme des Resultats einer ALU-Operation und das Speicherwertregister (*Store Value Register*) zum Puffern des zu speichernden Registerwerts im Falle einer Speicheroperation.
- Zwischen den MEM- und WB-Stufen befindet sich das Ladewertregister (*Load Memory Data Register*) zur Aufnahme des Datenworts im Falle einer Ladeoperation und das ALU-Ergebnisregister (*ALU Result Register*) zur Zwischenspeicherung des ALU-Ausgaberegisterwertes aus der vorherigen Pipeline-Stufe.

Die Stufen der DLX-Pipeline im Detail

Während der Befehlsausführung werden folgende Schritte ausgeführt:

- In der *IF-Stufe* wird der Befehl, auf den der PC zeigt, aus dem Code-Cache (*I-Cache*, *Instruction Cache*) in das Befehlsregister geholt und der PC um vier erhöht, um auf den nächsten Befehl im Speicher zu zeigen. Im Fall eines vorangegangenen Sprungbefehls kann die Zieladresse (über den

Pfad *Jump/Branch Target Address*) aus der MEM-Stufe benutzt werden, um den PC auf die im nächsten Takt zu holende Anweisung zu setzen.

- In der *ID-Stufe* wird der im Befehlsregister stehende Befehl in der ersten Takthälfte decodiert. In der zweiten Hälfte der Stufe wird abhängig vom Opcode eine der folgenden Aktionen ausgeführt:

  - *Register-Register* (bei arithmetisch-logischen Befehlen): Die Operandenwerte werden von den (Universal-)Registern (*Register File*) in das erste und das zweite ALU-Eingaberegister verschoben.
  - *Speicherreferenz* (bei Lade- und Speicherbefehlen): Ein Registerwert wird von einem (Universal-)Register in das erste ALU-Eingaberegister übertragen. Konstanten (*immediate*) oder Verschiebewerte (*Displacement*) aus dem Befehl werden vorzeichenerweitert und in das *Immediate*-Register transferiert. Da die „registerindirekte Adressierung mit Verschiebung“ die komplexeste Adressierungsart des DLX-Prozessors ist, kann im nachfolgenden Pipeline-Schritt der Registeranteil mit dem Verschiebewert addiert werden, um die effektive Adresse zu berechnen. Im Falle der direkten oder absoluten Adressierung wird der Registerwert Null aus Register R0 in das erste ALU-Eingaberegister geladen. Im Falle der registerindirekten Adressierung (ohne Verschiebung) wird statt des vorzeichenerweiterten Verschiebewertes der Wert Null erzeugt und in das Verschieberegister geladen. (Eigentlich wird dafür ein weiterer MUX benötigt, der in der Abbildung nicht gezeigt ist.) Im Fall eines Speicherbefehls wird der zu speichernde Registerwert in das zweite ALU-Eingaberegister transferiert.
  - *Steuerungstransfer* (bei Sprungbefehlen): Der Verschiebewert innerhalb des Befehls wird vorzeichenerweitert und zur Berechnung der Sprungzieladresse in das Verschieberegister transferiert (wir erlauben nur den befehlszählerrelativen Adressierungsmodus mit 16 Bit breiten Verschiebewerten - abweichend von den 26 Bit PC-Offset-Werten aus Abschnitt 1.3.2). Im Falle eines bedingten Sprungs wird der Wert der Verzweigungsbedingung (*true* oder *false*) zum zweiten ALU-Eingaberegister transferiert. Eine vorangegangene Vergleichsoperation muss diesen Wert produziert und im Universalregister gespeichert haben. Der Ausgang des zweiten ALU-Eingaberegisters ist mit einem Vergleicherschaltnetz (*Zero?*) verbunden, das gepuffert über ein Bedingungsregister (*Conditional Register*) den Multiplexer vor dem PC-Register der IF-Stufe ansteuert. Wenn die Verzweigungsbedingung erfüllt wurde, wird der neue PC-Wert aus dem ALU-Ausgaberegister übernommen.

- In der *EX-Stufe* erhält die arithmetisch-logische Einheit ihre Operanden – je nach Befehl aus den ALU-Eingaberegistern, aus dem (ID/EX-)PC-Register oder aus dem *Immediate*-Register – und legt das Ergebnis der arithmetisch-logischen Operation in dem ALU-Ausgaberegister ab. Die

Inhalte dieses Registers hängen vom Befehlstyp ab, welcher die MUX-Eingänge auswählt und die Operation der ALU bestimmt. Je nach Befehl können folgende Operationen durchgeführt werden:

  - *Register-Register* (bei arithmetisch-logischen Befehlen): Die ALU führt die arithmetische oder logische Operation auf den Operanden aus dem ersten und dem zweiten ALU-Eingaberegister durch und legt das Ergebnis im ALU-Ausgaberegister ab.
  - *Speicherreferenz* (bei Lade- und Speicherbefehlen): Von der ALU wird die Berechnung der effektiven Adresse durchgeführt und das Ergebnis im ALU-Ausgaberegister abgelegt. Die Eingabeoperanden erhält die ALU aus dem ersten ALU-Eingaberegister und dem *Immediate*-Register. Im Fall eines Speicherbefehls wird der Inhalt des zweiten ALU-Eingaberegisters (das den zu speichernden Wert beinhaltet) unverändert in das Speicherwertregister verschoben.
  - *Steuerungstransfer* (bei Sprungbefehlen): Die ALU berechnet die Zieladresse des Sprungs aus dem (ID/EX-)PC-Register und dem *Immediate*-Register und legt die Sprungzieladresse im ALU-Ausgaberegister ab. Gleichzeitig wird die Sprungrichtung (die feststellt, ob der Sprung ausgeführt wird oder nicht) aus dem zweiten ALU-Eingaberegister auf Null getestet und das Boole'sche Ergebnis im Bedingungsregister gespeichert.

- Die *MEM-Stufe* wird nur für Lade-, Speicher- und bedingte Sprungbefehle benötigt. Folgende Operationen werden abhängig von den Befehlen unterschieden:

  - *Register-Register*: Das ALU-Ausgaberegister wird in das ALU-Ergebnisregister übertragen.
  - *Laden*: Das Speicherwort wird so, wie es vom ALU-Ausgaberegister adressiert wird, aus dem Daten-Cache (D-Cache) gelesen und im Ladewertregister platziert.
  - *Speichern*: Der Inhalt des Speicherwertregisters wird in den Daten-Cache geschrieben, wobei der Inhalt des ALU-Ausgaberegisters als Adresse benutzt wird.
  - *Steuerungstransfer*: Für genommene Sprünge wird das Befehlszähler-Register (PC) der IF-Stufe durch den Inhalt des ALU-Ausgaberegisters ersetzt. Für nicht genommene Sprünge bleibt der PC unverändert. Das Bedingungsregister trifft die MUX-Auswahl in der IF-Stufe.

- In der *WB-Stufe* wird während der ersten Takthälfte der Inhalt des Ladewertregisters (im Falle eines Ladebefehls) oder des ALU-Ergebnisregisters (in allen anderen Fällen) in das (Universal-)Register gespeichert. Der Resultatregisterselektor (*Result Register Selector*) aus dem Befehl, der das (Universal-)Register als Resultatregister bezeichnet, wird ebenfalls durch

die Pipeline weitergegeben. (Diese Weitergabe der Registerselektoren ist in Abbildung 1.22 jedoch nur angedeutet.)

In Abbildung 1.22 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 die Multiplexer und die Operation der ALU.

Dabei benutzen alle Pipeline-Stufen unterschiedliche Ressourcen. 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 weiter geleitet. 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 Befehlsbearbeitungen gefüllt sein muss.

Es gibt leider mehrere potentielle Probleme, die eine reibungslose Befehlsausführung in der Pipeline stören können. Wenn zum Beispiel sowohl der Befehls- als auch der Datencache nachgeladen werden müssen und nur ein Speicherkanal (*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, sodass 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 so genannten Pipeline-Konflikten führen.

1.5.3 Pipeline-Konflikte

Als **Pipeline-Konflikt** bezeichnet man die Unterbrechung des taktsynchronen Durchlaufs der Befehle durch die einzelnen Stufen der Befehls-Pipeline. Pipeline-Konflikte werden durch Daten- und Steuerflussabhängigkeiten im Programm oder durch die Nichtverfügbarkeit von Ressourcen (Ausführungseinheiten, Registern etc.) hervorgerufen. 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.

Es 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.

In den nächsten Abschnitten werden die Pipeline-Konflikte und Möglichkeiten, wie man sie eliminiert oder zumindest ihre Auswirkung mindert, diskutiert.

1.5.4 Datenkonflikte und deren Lösungsmöglichkeiten

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

Datenabhängigkeitsarten

- 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 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 R (oder einer Speicherstelle) liest, das anschließend von I_2 überschrieben wird. Durch eine Gegenabhängigkeit kann 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 R (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.

Als Beispiel betrachte man die folgende Befehlsfolge, deren Abhängigkeitsgraph in Abbildung 1.23 dargestellt ist:

```
S1: ADD R1,R2,2  ; R1 = R2+2
S2: ADD R4,R1,R3 ; R4 = R1+R3
S3: MULT R3,R5,3 ; R3 = R5*3
S4: MULT R3,R6,3 ; R3 = R6*3
```

In diesem Fall besteht:

- eine echte Datenabhängigkeit von `S1` nach `S2`, da `S2` den Wert von Register `R1` benutzt, der erst in `S1` berechnet wird;

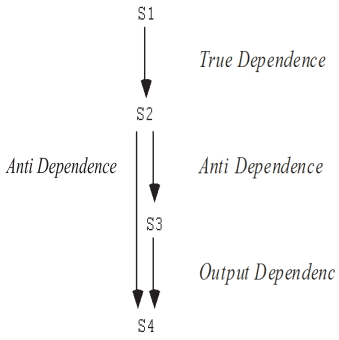


Abbildung 1.23: Abhängigkeitsgraph.

- eine Gegenabhängigkeit von `S2` nach `S3`, da `S2` den Wert von Register `R3` benutzt, bevor `R3` in `S3` einen neuen Wert zugewiesen bekommt; eine weitere Gegenabhängigkeit besteht von `S2` nach `S4`;
- eine Ausgabeabhängigkeit von `S3` nach `S4`, da `S3` und `S4` beide dem Register `R3` neue Werte zuweisen.

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 Variablenumbenennungen entfernt werden.

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

Selbsttestaufgabe 1.5 (Daten- und Steuerflussabhängigkeiten)
Es sei folgende Programmsequenz gegeben:

```
S1: ADDI    R1, R2, #2 ; R1 = R2 + 2
S2: SUB     R4, R1, R3 ; R4 =R1 -- R3
S3: SGE     R7, R4, R0 ; R4 >= 0 ? Status in R7
S4: BNEZ    R7, S7     ; wenn ja, gehe zu S7
S5: MULT    R3, R5, R6 ; R3 = R5 * R6
S6: J       S8         ; Goto S8
S7: ADDI    R3, R3, #2 ; R3 = R3 + 2
S8: ADDI    R4, R4, #1 ; R4 = R4 +1
```

Bestimmen Sie alle Daten- und Steuerflussabhängigkeiten in dieser Programmsequenz. Stellen Sie diese in einem Abhängigkeitsbaum dar.

Lösung auf Seite 66

Die ersten drei Datenabhängigkeiten in Abbildung 1.23 erzeugen Datenkonflikte in der Reihenfolge RAW, WAR, WAW. 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 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 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 R 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 R. 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 R schreiben. Deshalb liest, wenn nichts unternommen wird, I_2 in seiner ID-Stufe den alten Wert von R.

Lese-nach-Schreibe-Konflikte

Lese-nach-Schreibe-Konflikte (RAW) werden in Abbildung 1.24 für das Beispiel einer Befehlsfolge der Register-Register-Architektur DLX dargestellt.

Vor der Addition müssen beide Operanden des `add`-Befehls erst in Register geladen werden und nach der Addition wird das Ergebnis vom `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. (Scheinbare Datenabhängigkeiten sind in dieser Pipeline ohne Auswirkungen und werden deshalb weggelassen.) Die Tatsache, dass die Pfeile in die Rückrichtung der Zeitachse deuten, demonstriert, dass durch die Datenabhängigkeiten Datenkonflikte hervorgerufen werden, die bei Nichtbehandlung zu falschen Ergebnissen führen. Dieser Fall ist in Abbildung 1.25 für die letzten beiden Befehlsausführungen nochmals aufgezeigt.

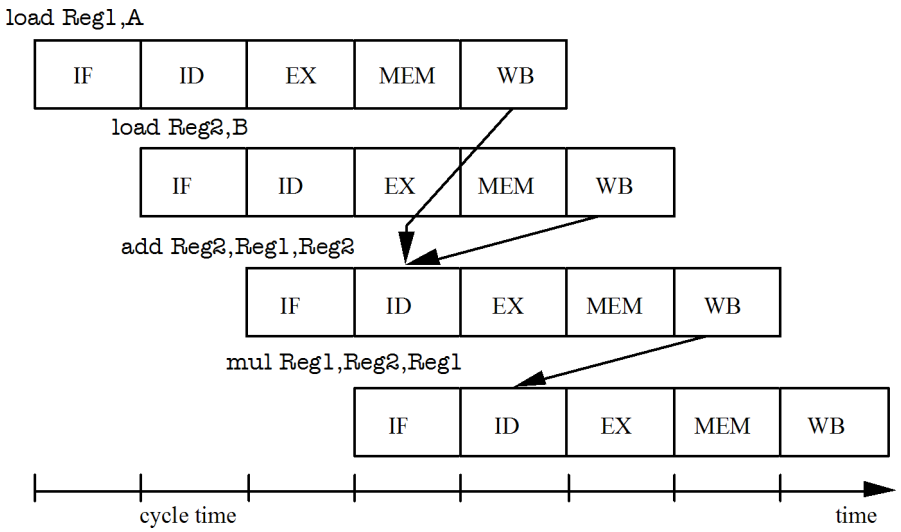


Abbildung 1.24: Datenkonflikte in einer Befehls-Pipeline.

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

Software-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.

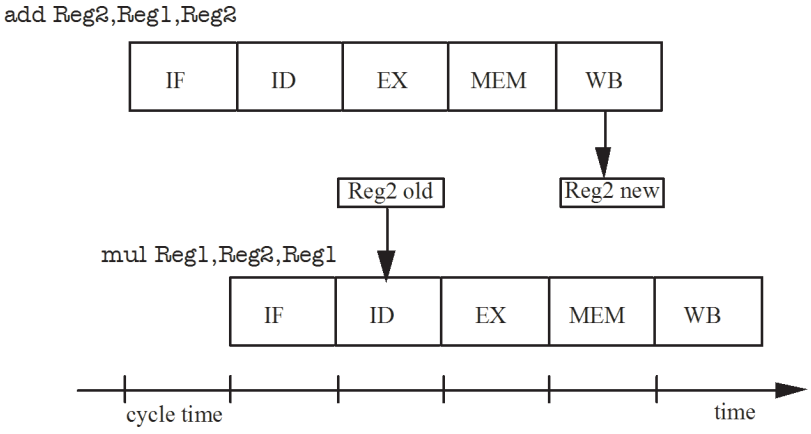


Abbildung 1.25: Durch einen nicht behandelten Datenkonflikt ausgelöste Fehlzuweisung; ein falsches Register wird gelesen.

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 `noop`-Befehle) nach jedem datenab-

hängigen 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, denn er kann den Programmcode mit dem Ziel neu anordnen, solche Leerbefehle möglichst zu eliminieren.

Im Beispiel aus Abbildung 1.24 sollten, wenn möglich, zwei Befehle, die keine neuen Datenkonflikte hervorrufen, zwischen dem `add`- und dem `mul`-Befehl eingefügt werden. Diese Methode wird **Befehlsanordnung** (*Instruction Scheduling* oder *Pipeline Scheduling*) genannt. Eine solche compilerbasierte Befehlsanordnung wurde bereits bei den ersten RISC-Prozessoren angewandt.

Hardware-Lösungen für das Datenkonflikt-Problem

Die *Hardware-Lösungen* erfordern das Erkennen der Datenkonflikte durch die Hardware und deren automatisches Behandeln. Dies ist heute Standard. An Hardware-Lösungen für das Datenkonflikt-Problem unterscheiden wir die folgenden:

- *Leerlauf der Pipeline*: Die einfachste Art, mit solchen Datenkonflikten umzugehen ist es, den Befehl I_2 in der Pipeline für 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 so genannte **Pipeline-Blasen** (*Pipeline Bubbles*), die den gleichen Effekt wie Leerbefehle hervorrufen, nämlich, die Ausführungsgeschwindigkeit deutlich herabzusetzen. Im Beispiel in Abbildung 1.24 erzeugt das Anhalten zwei Pipeline-Blasen (s. Abbildung 1.26). Dabei nehmen wir für unsere DLX-Pipeline an, dass das Zurückschreiben in die Register in der ersten Hälfte der WB-Stufe abgeschlossen wird und derselbe Registerwert bereits wieder während der Operandenholephase in der zweiten Hälfte der ID-Stufe gelesen werden kann. (Ohne diese Annahme müssten drei Pipeline-Blasen eingefügt werden, da die ID-Phase eigentlich erst nach vollständigem Abschluss der WB-Phase starten kann. Dieser Fall wird in einer Einsendeaufgabe näher betrachtet.)

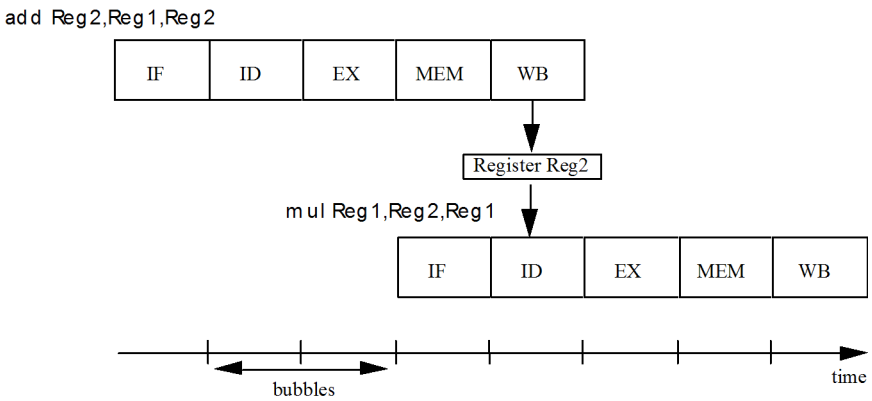


Abbildung 1.26: Datenkonflikt: Hardware-Lösung durch Interlocking.

Forwarding wird auch als *Bypass* bezeichnet.

- *Forwarding*: Es gibt eine bessere Lösung, die allerdings etwas mehr Hardwareaufwand erfordert und *Forwarding* genannt wird. Wenn ein Datenkonflikt erkannt wird, so sorgt eine Hardwareschaltung dafür, dass der betreffende Operand nicht aus dem Universalregister, sondern direkt aus dem ALU-Ausgaberegister der vorherigen Operation in das ALU-Eingaberegister übertragen wird. Übertragen auf unsere DLX-Pipeline und das Beispiel aus Abbildung 1.24 bedeutet dies, dass der `mul`-Befehl nicht warten muss, bis das Ergebnis des davor stehenden `add`-Befehls während der WB-Phase in das Universalregister R geschrieben wird. 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. In unserem Beispiel, in dem beide Befehle vom Register-Register-Typ sind, beseitigt das Forwarding alle Leertakte (s.Abbildung 1.27).

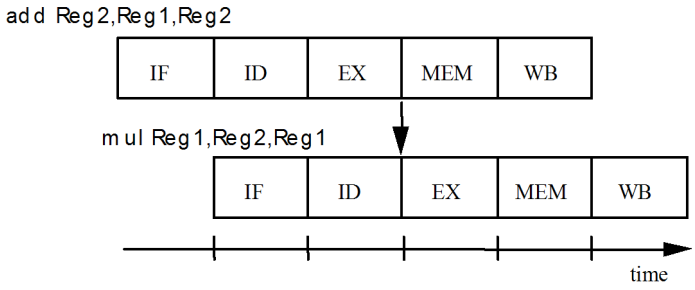


Abbildung 1.27: Datenkonflikt: Hardware-Lösung durch Forwarding.

- *Forwarding mit Interlocking*: 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 (s. Abbildung 1.28 für den Konflikt und Abbildung 1.29 für den verbleibenden Leertakt).

Wie kann Forwarding implementiert werden? In unserer einfachen DLX-Pipeline genügt es, die zwei Resultatregisterselektoren der beiden Befehle, die sich gerade in der EX- und in der MEM-Phase befinden, mit den beiden Operandenregisterselektoren des Befehls, der sich gerade in der ID-Phase befindet, zu vergleichen. Ist eine Übereinstimmung vorhanden, so wird ein Pfad geschaltet, der nicht den Wert aus dem Universalregister, sondern den Wert, der in der betreffenden Stufe (EX oder MEM) berechnet wird, direkt in das ALU-Eingaberegister übernimmt.

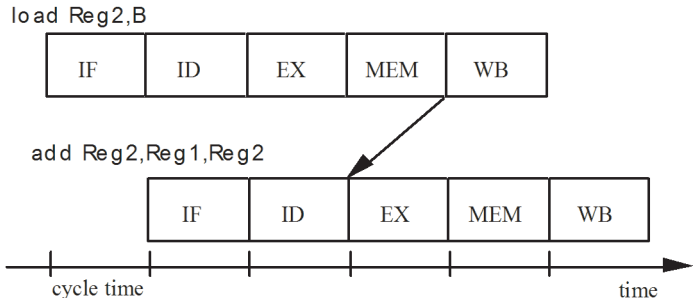


Abbildung 1.28: Pipeline-Konflikt durch eine Datenabhängigkeit, der nicht durch Forwarding verhindert werden kann.

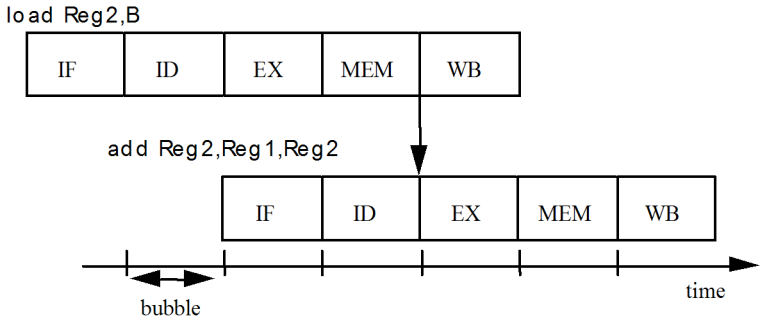


Abbildung 1.29: Leertakt aufgrund einer Datenabhängigkeit.

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.

1.5.5 Steuerflusskonflikte und deren Lösungsmöglichkeiten

Programmsteuerbefehle erzeugen Steuerflussänderungen und damit möglicherweise 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, sodass weitere Befehle des falschen Programmpfades in die Pipeline geraten, bevor der richtige Befehl vom PC adressiert und in die Pipeline geladen werden kann. Eine Besonderheit stellen die bedingten Sprungbefehle (Verzweigungen) 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 PC 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 gelöscht werden und der Befehl an der Sprungadresse geladen werden.

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 (s. Abbildung 1.30).

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 zum PC findet schon in der ID-Stufe statt (falls der Sprung genommen wird), so ergibt sich nur *ein* Wartezyklus.

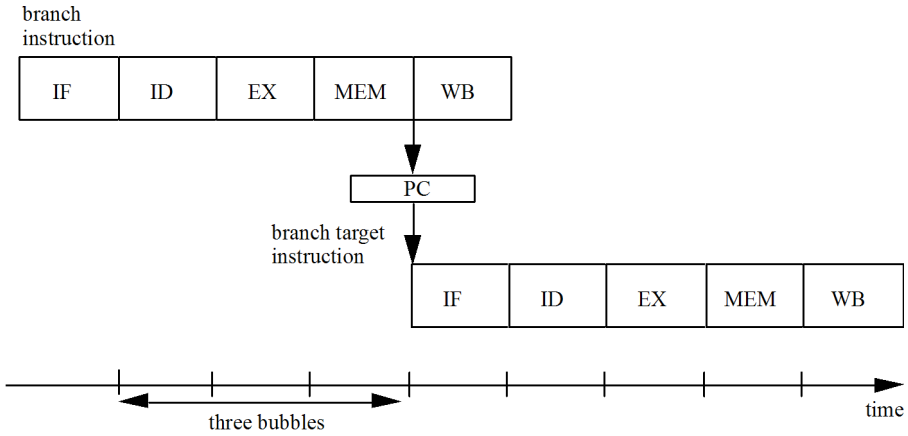


Abbildung 1.30: Leertakte nach einem genommenen bedingten Sprung.

Obwohl dies die Verzögerung auf einen einzelnen Takt verringert, kann nun ein neuer nicht behebbarer Pipeline-Konflikt entstehen. Ein ALU-Befehl gefolgt von einem bedingten Sprung, der vom Ergebnis dieses Befehls abhängt, wird einen Konflikt mit Verzögerung verursachen, auch wenn das Ergebnis von der EX- in die ID-Stufe weitergeleitet wird (ähnlich dem in Abbildung 1.28 dargestellten Datenkonflikt eines Ladebefehls mit einer nachfolgenden ALU-Operation, die den geladenen Wert benötigt).

Das Hauptproblem bei dieser Reorganisation der Pipeline ist jedoch, dass die Decodierung, die Sprungberechnung und das Rückschreiben des PC sequenziell in einer einzigen Pipeline-Stufe 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 längste Gatterpfad, den die Signale durchlaufen müssen.

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

Software-basierte Lösungen für Steuerflusskonflikte

Diese Steuerflusskonflikte können mit Hilfe von verschiedenen Software-basierten Techniken behandelt werden:

- *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.

- *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. Dies ist aus Hardware-Sicht die einfachste Lösung. Dadurch, dass geladene Befehle nicht gelöscht werden müssen, verringert sich die Komplexität der Hardware. (Unsere einfache Pipeline ist von diesem Typ.) Wenn es keine Befehle gibt, die in die Zeitschlitze verschoben werden können, müssen Leerbefehle eingefügt werden.

Entsprechend den Ergebnissen von Programmtestläufen ist die Wahrscheinlichkeit, dass *ein* Befehl in einen Verzögerungszeitschlitz verschoben werden kann, größer als 60%, die Wahrscheinlichkeit für zwei Befehle bei ungefähr 20% und die Wahrscheinlichkeit für drei kleiner als 10%.

Die verzögerte Sprungtechnik mit einem Zeitschlitz wurde bei den ersten Generationen skalarer RISC-Prozessoren angewandt wie dem IBM801, dem Berkeley RISC I und dem Stanford MIPS. In superskalaren Prozessoren, die mehr als einen Befehl holen und gleichzeitig verarbeiten können, erschwert die verzögerte Sprungtechnik die Befehlszuordnungslogik und die Implementierung präziser Unterbrechungen. (Unter einer präzisen Unterbrechung versteht man dabei ein Verfahren, bei dem die Resultate aller Befehle, die in der Programmreihenfolge vor dem Unterbrechungsereignis stehen, gültig sind und diejenigen aller nachfolgenden Befehle verworfen werden. Abhängig von der Architektur und der Art der Unterbrechung, wird das Resultat des verursachenden Befehls noch gültig gemacht oder verworfen, ohne weitere Auswirkungen zu haben.) Aus Kompatibilitätsgründen gibt es sie aber immer noch in den Architekturen einiger heutiger Mikroprozessoren, z.B. in den SPARC- oder MIPS-basierten Prozessoren.

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

- *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.

- *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 so genannten statischen Sprungvorhersagen dar. Leider ist sie auch die ineffizienteste, da bedingt durch die in Programmen häufig auftretenden Schleifen die Mehrzahl der Sprünge genommen wird.

Eine Verbesserung der Sprungbehandlung ist nur dann möglich, wenn auch als „genommen“ vorhergesagte bedingte Sprünge und damit auch alle anderen Programmsteuerbefehle, die immer zu Steuerflussänderungen führen, unterstützt werden. Das ist jedoch nur möglich, wenn die Sprungzieladresse nicht (erneut) berechnet werden muss, sondern bereits in der IF-Stufe der richtige Nachfolgebefehl geladen werden kann. Dafür wird ein kleiner Pufferspeicher in der IF-Stufe benötigt, der nach dem ersten Durchlauf der Befehlsfolge die Adresse und die Sprungzieladresse eines Programmsteuerbefehls puffert, um

beim nächsten Auftreten desselben Programmsteuerbefehls sofort an der Zieladresse weiter laden zu können. Mit Lösungen dieser Art werden wir uns in den folgenden Unterabschnitten beschäftigen.

1.5.6 Sprungzieladress-Cache

Die Sprungzieladresse wird zur gleichen Zeit gebraucht wie die Vorhersage der Sprungentscheidung selbst. Insbesondere sollte die Sprungzieladresse bereits in der IF-Stufe bekannt sein, damit nach einem geladenen Sprungbefehl schon im nächsten Takt von der Zieladresse geladen werden kann.

Der **Sprungzieladress-Cache** (*Branch Target Address Cache* – BTAC) oder **Sprungzielpuffer** (*Branch Target Buffer* – BTB) ist ein kleiner Cache-Speicher, auf den in der IF-Stufe der Pipeline zugegriffen wird. Der Sprungzieladress-Cache umfasst eine Menge von Tupeln, von denen jedes die folgenden Elemente enthält (s. Abbildung 1.31):

Sprungzieladress-Cache

- Feld 1: die Adresse (*Branch Address*) eines Sprungbefehls, der in der Vergangenheit ausgeführt wurde,
- Feld 2: die Zieladresse (*Target Address*) dieses Sprungs,
- Feld 3 (optional): 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.

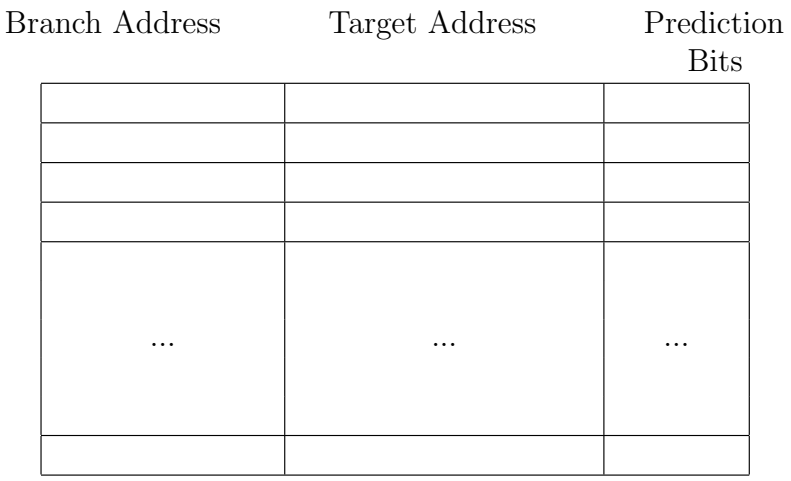


Abbildung 1.31: Sprungzieladress-Cache.

Der Sprungzieladress-Cache funktioniert wie folgt: Die IF-Stufe vergleicht den Inhalt des Befehlszählerregisters (PC) mit den Adressen der Sprungbefehle im Sprungzieladress-Cache (Feld 1). Falls ein passender Eintrag gefunden wird, wird die zugehörige Zieladresse als neuer PC benutzt und der Befehl am Sprungziel als nächstes geholt.

Falls kein Eintrag vorhanden ist, so wird ein Eintrag erzeugt, sobald die Sprungadresse berechnet ist, und dafür eventuell ein anderer Eintrag überschrieben. Der Sprungzieladress-Cache ist meist als vollassoziativer Cache-Speicher organisiert, d.h. der Vergleich des aktuellen Befehlszähler-Inhalts geschieht simultan in einem Takt mit allen im Cache gespeicherten Adressen von Sprungbefehlen (siehe Unterabschnitt 3.1.4.2). Er benötigt wie alle Cache-Speicher eine gewisse „Aufwärmzeit“, in der beim erstmaligen Ausführen der Sprungbefehle die Einträge erzeugt werden.

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.

Wenn 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 er als „genommen“ vorhergesagt, so wird die zugehörige Sprungzieladresse aus Feld 2 in den PC übertragen und mit ihrer Hilfe der Zielbefehl geholt.

Dynamische Sprungvorhersage

Natürlich kann eine falsche Vorhersage auftreten. Dann muss der Sprungzieladress-Cache die Vorhersagebits korrigieren, sobald die Sprungrichtung in der MEM-Stufe bekannt ist. Da die Hardware die Vorhersagerichtung aufgrund der Sprungverläufe während der Programmausführung modifiziert, ist diese Art der Sprungvorhersage ein Beispiel einer einfachen **dynamischen Sprungvorhersage**. Um die Größe des Sprungzieladress-Cache klein zu halten, kann die Implementierung so geschehen, dass von den bedingten Sprünge nur diejenigen, die als „genommen“ vorhergesagt werden, gespeichert werden.

Das Laden der Befehle ab einer Zieladresse geschieht ohne Verzögerung, wenn die Befehle schon im Code-Cache vorhanden sind. Eine Variante des Sprungzieladress-Cache war in älteren Prozessoren ohne On-Chip-Cache verbreitet: Der so genannte **Sprungziel-Cache** (*Branch Target Cache* – BTC) speicherte pro Eintrag noch zusätzlich ein paar Befehle am Sprungziel ab.

Rücksprungadressstapel

Zusätzlich gibt es unabhängig vom Sprungzieladress-Cache häufig einen kleinen **Rücksprungadressstapel** (*Return Address Stack* – RAS), auf dem bei Unterprogrammaufrufen (`call`-Befehlen) die Rücksprungadressen abgelegt werden. Dieser Speicher ist als Stapelspeicher organisiert, was dem `call`- und `return`-Verhalten bei der Programmausführung entspricht. Im Prinzip können auch die Rücksprungadressen der Interrupt-Aufrufe darauf abgelegt werden. Problematisch wird es nur, wenn die Anzahl der Pufferplätze nicht ausreicht. Dann muss jeweils zunächst die älteste gespeicherte Rücksprungadresse im Arbeitsspeicher abgelegt und bei Bedarf wieder von dort geholt werden.

Rücksprungadressstapel sind meist klein. Beispielsweise umfasst der Rücksprungstapel beim beim Athlon-Prozessor von AMD sechzehn Einträge.

1.5.7 Statische Sprungvorhersagetechniken

Statische vs. dynamische Sprungvorhersage

Die statische Sprungvorhersage ist eine sehr einfache Vorhersagetechnik, bei der die Hardware jeden bedingten Sprung nach einen festen Muster vorhersagt. In manchen Architekturen kann der Compiler durch ein Bit die Spekulations-

richtung für alle bedingten Sprungbefehle gemeinsam festlegen, indem er die hardwarebasierte Vorhersage umkehrt. In beiden Fällen kann die Vorhersage für einen speziellen Sprungbefehl sich nie ändern. Eine solche „dynamische“ Änderung der Vorhersagerichtung ist jedoch bei den o.g. dynamischen Sprungvorhersagetechniken der Fall, welche die Vorgeschichte des Sprungbefehls zur Sprungvorhersage nutzen.

Nach einer statischen Sprungvorhersage werden die Befehle auf dem spekulativen Pfad vom Prozessor in die Pipeline eingefüttert, bis die Sprungrichtung entschieden ist. Falls richtig spekuliert wurde, so können die Ergebnisse der Befehle gültig gemacht werden und es treten keine Verzögerungen auf. Falls die Spekulation fehlgeschlagen ist, so müssen diese Befehle wieder aus der Pipeline gelöscht werden.

Einfache Muster für die statische Vorhersage in Hardware sind:

- *Predict always not taken*: Dies ist das einfachste Schema, bei dem angenommen wird, dass kein bedingter 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 eine Fehlspekulation stattfindet, ist diese Technik nicht sehr effizient. (Die Technik sollte nicht mit der verzögerten Sprungtechnik verwechselt werden, bei der die Befehle in den Verzögerungszeitschlitzen immer ausgeführt werden.) Hier werden alle Ergebnisse der Befehle nach einem falsch spekulierten Sprung verworfen.

- *Predict always taken*: Hier wird angenommen, dass jeder Sprung genommen wird. Damit werden alle Rücksprünge während einer Schleife richtig vorhergesagt. Ein Sprungzieladress-Cache ist für ein verzögerungsfreies Befehlsladen nötig.

- *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.

Manche Befehlssätze ermöglichen es dem Compiler, mit einem Bit im Befehlscode der Sprunganweisung die Vorhersage direkt zu steuern (z.B. ‚Bit gesetzt‘ bedeutet *predict taken*, ‚Bit nicht gesetzt‘ bedeutet *predict not taken*) oder, bei Anwendung einer der obigen statischen Vorhersagetechniken, die statische Vorhersage für den betreffenden Sprungbefehl umzukehren.

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 eine Analyse des Verhaltens von früheren Programmabläufen (*Profiling*) kann das voraussichtliche Verhalten jedes Sprungs ermittelt werden.

Dabei ist die Technik des Profilings fast immer die beste, aber auch die aufwändigste.

1.5.8 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 unsere 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 nun 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, sodass ein Ressourcenkonflikt entsteht, sofern nur ein einzelner Schreibkanal auf die Register vorhanden ist (Abbildung 1.32).

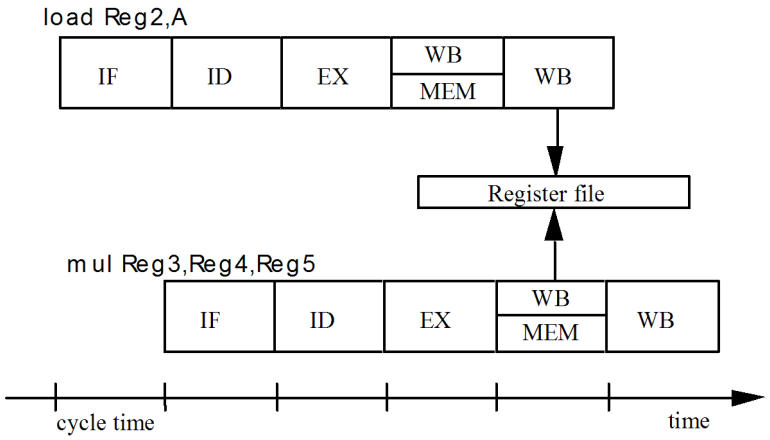


Abbildung 1.32: Beispiel eines Strukturkonflikts im Falle einer veränderten (falsch entworfenen) DLX-Pipeline.

Hardware-Lösungen zur Vermeidung von Strukturkonflikten

Zur Vermeidung von Strukturkonflikten gibt es folgende Hardware-Lösungen:

- *Arbitrierung mit Interlocking*: Strukturkonflikte können durch eine Arbitrierungslogik erkannt und aufgelöst werden. Die Arbitrierungslogik hält den im Programmfluss späteren der beiden um die Ressource konkurrierenden Befehle an. Wenn diese Technik dazu benutzt wird, um Strukturkonflikte zu lösen, kommt man nicht ohne Verzögerung aus. Im Beispiel in Abbildung 1.32 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 in einer Pipeline-Stufe auf die Ressource zugreifen und die Ressourcenanforderungen in der Ausführungsreihenfolge erfüllen.

  Als Beispiel betrachte man die Abänderung des load-Befehls in Abbildung 1.32 zu load Reg3,A.

- *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. Alternativ kann jedoch beim Registerschreibzugriff zweier direkt hintereinander stehender Befehle der Wert, der vom ersten Befehl geschrieben werden soll, gestrichen und statt dessen nur der Wert des zweiten schreibenden Befehls ins Register zurückgeschrieben werden. Das heißt, übertragen auf unser Beispiel in Abbildung 1.32, der Wert, der vom `load`-Befehl produziert wurde, wird gestrichen und der Wert des ALU-Ausgaberegisters des `mul`-Befehls wird ausgewählt und in das Zielregister geschrieben. Dieses sehr effiziente Verfahren beruht auf der Beobachtung, dass der erste Resultatwert ja im Programmablauf nie verwendet wird, da er ja vom zweiten Befehl sofort überschrieben wird.

1.5.9 Ausführung in mehreren Takten

Mehrtaktbefehle in der Pipeline

Betrachten wir eine Folge von zwei Befehlen I_1 und I_2, bei der I_1 vor I_2 geholt wird, und nehmen an, dass I_1 ein lange rechnender Befehl (z.B. ein Gleitkommabefehl) sei. Weitere Beispiele für Befehle, deren Operationen nicht in einem Pipeline-Takt ausgeführt werden können, sind die Lade- und die Speicherbefehle. Ein solcher Befehl benötigt (mindestens) zwei Takte zur Ausführung: einen Takt zur Berechnung der effektiven Adresse und einen zweiten Takt für den Zugriff auf den Daten-Cache. Zur Verarbeitung eines lang laufenden Befehls I_1 wäre es unpraktisch zu fordern, dass alle Befehle ihre EX-Stufe in einem Takt beenden. Denn dann würde entweder eine geringe Taktfrequenz entstehen oder man müsste die Hardware-Logik stark vergrößern bzw. beides.

Stattdessen wird es der EX-Stufe erlaubt, so viele Takte zu verbrauchen, wie sie benötigt, um I_1 abzuschließen. Dies verursacht jedoch einen Strukturkonflikt in der EX-Stufe, weil der nachfolgende Befehl I_2 die ALU im nächsten Takt nicht nutzen kann. Einige mögliche Lösungen sind:

- *Pipeline anhalten*: Die einfachste Weise, mit einem solchen Strukturkonflikt umzugehen, ist es, I_2 in der Pipeline anzuhalten, bis I_1 die EX-Stufe verlässt.

- *Ressourcen-Pipelining*: Falls die EX-Stufe selbst aus einer Pipeline besteht, wird der Strukturkonflikt vermieden, weil die EX-Stufe zu jedem Takt einen neuen Befehl entgegen nehmen kann (der Durchsatz ist 1).

- *Ressourcenreplizierung*: Es kann mehrere Ausführungseinheiten geben, sodass I_2 in einer anderen Ausführungseinheit verarbeitet werden kann und eine EX-Stufe mit der EX-Stufe von I_1 überlappt.

Das Anhalten der Pipeline ist natürlich ineffizient, da es Leertakte erzeugt und so die Ausführungsgeschwindigkeit senkt. Die Unterteilung der EX-Stufe in zwei Pipeline-Stufen ist die Lösung, die für die Ausführung von Lade-/Speicherbefehlen in unserer Beispiel-Pipeline durch separate EX- und MEM-Stufen (anstatt einer einzelnen kombinierten EX/MEM-Stufe) gewählt wurde. Das Streben nach einer einfachen Hardware-Implementierung unter Voraussetzung der zweitaktigen Lade-/Speicherbefehle war der Grund dafür, die Ergebnisse von eintaktigen arithmetisch-logischen Befehlen durch die MEM-Stufe weiterzuleiten. Dies verzögert das Zurückschreiben der Ergebnisse um einen Takt, vermeidet aber Schreibe-nach-Schreibe-Konflikte in der Pipeline.

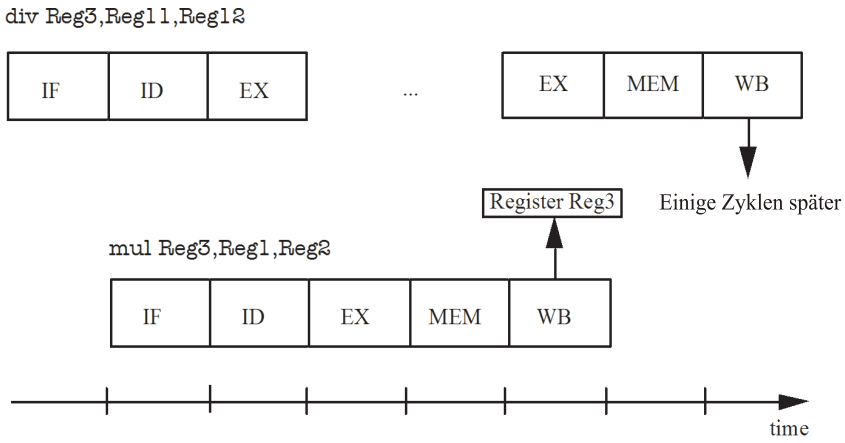


Abbildung 1.33: Beispiel eines Schreibe-nach-Schreibe-Konflikts.

Eine komplexere Lösung ist die Nutzung mehrerer Ausführungseinheiten und die gleichzeitige Ausführung. Diese Lösung beinhaltet jedoch, dass Befehle nicht in der ursprünglichen Reihenfolge abgeschlossen werden (s. Abbildung 1.33): Da die EX-Stufen des `div`-Befehls von einem bis zu einigen dutzend Takten dauern können, setzt der `mul`-Befehl mit der WB-Stufe vor dem `div`-Befehl fort. Leider kann eine solche Ausführung außerhalb der Reihenfolge dann einen Schreibe-nach-Schreibe-Konflikt verursachen, wenn es eine Ausgabeabhängigkeit zwischen den beiden Befehlen gibt. Es gibt zwei Lösungen, um den Schreibe-nach-Schreibe-Konflikt in Abbildung 1.33 zu lösen:

- Der `mul`-Befehl wartet mit dem Zurückschreiben, bis der `div`-Befehl sein Ergebnis in das Register geschrieben hat, das anschließend überschrieben wird.

- Die elegantere Lösung ist das sofortige Zurückschreiben des Ergebnisses des `mul`-Befehls und das Verwerfen des Ergebnisses des `div`-Befehls, das im Beispiel in Abbildung 1.33 von keinem anderen Befehl benutzt wird. Leider stellt sich nun die Frage, wie man eine präzise Unterbrechung implementiert, z.B. im Falle einer Division durch Null.

Bisher haben wir einen einfachen Pipeline-Prozessor betrachtet, der nur eine Ausführung in Programmreihenfolge unterstützt, d.h. die Befehle werden an die Ausführungseinheit in genau der gleichen Reihenfolge wie im Programm zugewiesen und von dieser ausgeführt. Wenn mehrere Ausführungseinheiten vorhanden sind, so ist eine Ausführung außerhalb der Reihenfolge der nächste Schritt. Im Fall einer Ausführung außerhalb der Reihenfolge müssen Schreibe-nach-Schreibe-Konflikte gelöst werden. Sogar eine Gegenabhängigkeit kann einen Schreibe-nach-Lese-Konflikt verursachen, falls ein nachfolgender Befehl seine Ausführung beginnt und sein Ergebnis zurückschreibt, bevor ein vorhergehender Befehl seine Operanden bekommt. Lösungen zu diesem Problem werden detailliert in [9] und in [16] beschrieben. Sie können aus Platzgründen in diesem Kurs nicht behandelt werden.

1.6 Weitere Aspekte des Befehls-Pipelinings

Super-Pipelining

Eine Weiterentwicklung des Befehls-Pipelinings ist das so genannte ***Super-Pipelining***, das heute meist mit dem Vorhandensein einer „langen“ Befehls-Pipeline gleichgesetzt wird. Nach der ursprünglichen, im Rahmen des MIPS-R4000-Prozessors verwendeten Bedeutung des Begriffs „Super-Pipelining“ wurden die Stufen einer Befehls-Pipeline in feinere Pipelinestufen unterteilt und mit einem schnellen, chip-internen Takt – schneller als der Takt des Gesamtrechnersystems – ausgeführt. Die Anzahl der Pipeline-Stufen erhöht sich damit von fünf beim R3000- auf acht beim R4000-Prozessor. Der Zugriff auf den Primär-Cache-Speicher auf dem Prozessor-Chip geschieht dann mit der hohen Taktrate des Prozessors. Außerhalb des Prozessor-Chips wird meist ein niedrigerer Takt verwendet, der um ein Vielfaches langsamer als der Prozessortakt ist. Dies erlaubt eine sehr hohe Taktfrequenz für den Prozessor-Chip und niedrigere Taktfrequenzen (und damit billigere Chips) für das Gesamtsystem. Da mittlerweile die Taktraten heutiger Prozessoren so hoch sind, dass kein Bussystem und häufig nicht einmal der Sekundär-Cache-Speicher mithalten kann, ist die Technik, den Prozessor selbst mit einer wesentlich höheren Taktfrequenz als das Gesamtsystem zu betreiben, allgemein verbreitet.

Kurze vs. lange Befehls-Pipelines

Während noch zu Beginn der 80er Jahre Befehls-Pipelining praktisch ausschließlich Großrechnern und Supercomputern vorbehalten war, sind Befehls-Pipelines zunächst bei RISC-Prozessoren Anfang der 80er Jahre in die Mikroprozessortechnik eingeführt worden und heute in allen Mikroprozessoren Stand der Technik. Dabei gibt es zwei Trends: kurze Befehls-Pipelines von 4 – 6 Stufen (Bsp. PowerPC-Prozessoren) und lange Befehls-Pipelines von 7 – 20 Stufen (Bsp. MIPS-Prozessoren, SuperSPARC, UltraSPARC und Pentium 4).

Betrachtet man die Ausführungsphase einer Befehls-Pipeline genauer, stellt man fest, dass praktisch alle arithmetisch-logischen Befehle auf Festpunktoperanden (abgesehen von der Division) in einem Takt ausführbar sind, während Gleitkommaoperationen derart komplex sind, dass ihre Eingliederung als *eine* Phase einer Befehls-Pipeline das fein abgestimmte Gleichgewicht der einzelnen Pipelinestufen zerstören würde.

Eine komplexe Operation wie die Gleitkommamultiplikation oder die Gleit-

kommaaddition wird deshalb ebenfalls wieder in verschiedene Stufen zerlegt, die an der Stelle der Ausführungsstufe in eine Befehls-Pipeline eingegliedert werden. Derartige, meist dreistufige Pipelines für Gleitkommaoperationen werden **Gleitkommaeinheiten** genannt. Mit jedem Takt kann von der vorherigen Stufe der Befehls-Pipeline eine Gleitkommaoperation in eine solche Gleitkommaeinheit eingefüttert werden. Das Resultat steht jedoch erst nach drei Takten zur Verfügung.

Gleitkommaeinheit

Wegen der andersartigen Ausführungsstufen für Festpunkt-, Gleitkomma-, Lade-/Speicher- und Verzweigungsbefehle existieren auf heutigen superskalaren Mikroprozessoren mehrere, funktional verschiedenartige Befehls-Pipelines, die nur noch die Befehlsbereitstellungs-, die Decodier-, eine so genannte Befehlszuordnungs- und eine Rückordnungsstufe gemeinsam besitzen.

Selbsttestaufgabe 1.6 (DLX-Befehls-Pipeline)
Wie in Selbsttestaufgabe 1.4 sei folgende Programmsequenz gegeben:

```
S1: ADDI    R1, R2, #2 ; R1 = R2 + 2
S2: SUB     R4, R1, R3 ; R4 = R1 -- R3
S3: SGE     R7, R4, R0 ; R4 >= 0 ? Status in R7
S4: BNEZ    R7, S7     ; wenn ja, gehe zu S7
S5: MULT    R3, R5, R6 ; R3 = R5 * R6
S6: J       S8         ; Goto S8
S7: ADDI    R3, R3, #2 ; R3 = R3 + 2
S8: ADDI    R4, R4, #1 ; R4 = R4 +1
```

Es sei angenommen, dass bei der Ausführung der oben abgebildeten Programmsequenz die einzelnen Befehle in einer fünfstufigen DLX-Befehls-Pipeline verarbeitet werden. (Hinweis: Bei einem Sprungbefehl wird der Program Counter bereits nach der 4. Stufe auf das neue Sprungziel gesetzt.)

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

Lösung auf Seite 67

Selbsttestaufgabe 1.7 (Kontrollfragen)

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?

Lösung auf Seite 68

1.7 Lösungen zu den Selbsttestaufgaben

Selbsttestaufgabe 1.1 von Seite 22

Welche einfacheren Adressierungsarten lassen sich durch die registerindirekte Adressierung mit Verschiebung ersetzen?

Lösung:

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.

Selbsttestaufgabe 1.2 von Seite 30

Arbeiten Sie sich in den DLX-Simulator ein. Schreiben Sie ein einfaches Programm, welches das Quadrat einer Zahl berechnet, die sich im Register R5 befindet. Das Ergebnis soll im Register R6 abgespeichert werden. Schauen Sie sich dabei die Registerbelegungen im DLX-Simulator genau an.

Lösung:

```
;------------------------------------------------------------------------
; Programmanfang bei Symbol main ; Berechnet das Quadrat vom Inhalt
R5 und schreibt das Ergebnis in R6
;------------------------------------------------------------------------

    .text
    .global main

main:
    addi R5, R0, #25    ;Lade den Wert 25 in R5
    mult R6, R5, R5     ;Berechne Quadrat und speichere Ergebnis in R6
    trap 0              ;Exit
```

Selbsttestaufgabe 1.3 von Seite 30

Über die Eingabeaufforderung sollen zunächst zwei Integerzahlen eingelesen werden. Schreiben Sie ein Programm:

a) welches die Summe dieser zwei Integerzahlen berechnet und das Ergebnis ausgibt.

Lösung:

```
    .data
;Variablendeklaration Prompt1: .asciiz "\nEnter first number:"
    .align 4
Prompt2: .asciiz "\nEnter second number:"
    .align 4

;Das Ausgabeformat. %d steht für Dezimalzahl (%f oder %g für Gleitkommazahl).
PrintFt: .asciiz "The sum of %d and %d = %d"
    .align 6
;Pointer auf PrintFt PrintPar: .word PrintFt Int1: .space 4 Int2:
.space 4 PrintSum: .space 4
```

```
    .text
    .global main
main:
    addi r1, r0, Prompt1    ;Kopiere die Adresse von Prompt1 in R1. Wird
                            ;von InputUnsigned benutzt um die
                            ;Eingabeaufforderung auszugeben.
    jal InputUnsigned       ;Aufruf von InputUnsigned in Input.s. R1 hat am
                            ;Ende den Wert der Eingabe.
    sw Int1, r1             ;Speichere den Eingabewert in Int1.
    add r2, r0, r1          ;Speichere den Eingabewert in R2 (für die
                            ;Addition).
    addi r1, r0, Prompt2    ;Kopiere die Adresse von Prompt2 in R1.
    jal InputUnsigned       ;Aufruf von InputUnsigned in Input.s. Eingabe der
                            ;zweiten Zahl.
    sw Int2, r1             ;Speichere den Eingabewert in Int2.
    add r3, r1, r2          ;Addiere nun die beiden Zahlen. Ergebnis in R3.
    sw PrintSum,r3          ;Speichere Summe in PrintSum.
    addi r14,r0,PrintPar    ;Lade nun die Ausgabeparameter in R14.
    trap 5                  ;Starte Ausgabe.
    trap 0                  ;Exit.
```

b) welches die zweite Zahl von der ersten abzieht und das Ergebnis ausgibt.

Lösung:

```
    .data
Prompt1: .asciiz "\nEnter first number:"
    .align 4
Prompt2: .asciiz "\nEnter second number:"
    .align 4
StrPtr: .space 4
PrintFt: .asciiz "The result of (%d - %d) is %d"
    .align 6
PrintPar: .word PrintFt Int1: .space 4 Int2: .space 4 PrintRes:
.space 4

    .text
    .global main
main:
    addi r1, r0, Prompt1    ;Kopiere die Adresse von Prompt1 in R1. Wird
                            ;von InputUnsigned benutzt um die
                            ;Eingabeaufforderung auszugeben.

    jal InputUnsigned       ;Aufruf von InputUnsigned in Input.s. R1 hat am
                            ;Ende den Wert der Eingabe.
    sw Int1, r1             ;Speichere den Eingabewert in Int1.
    add r2, r0, r1          ;Speichere den Eingabewert in R2 (für die
                            ;Differenz).
    addi r1, r0, Prompt2    ;Kopiere die Adresse von Prompt2 in R1.
    jal InputUnsigned       ;Aufruf von InputUnsigned in Input.s. Eingabe der zweiten Zahl.
    sw Int2, r1             ;Speichere den Eingabewert in Int2.
    sub r3, r2, r1          ;Subtrahiere nun die beiden Zahlen. Ergebnis in
                            ;R3.
    sw PrintRes,r3          ;Speichere Ergebnis in PrintRes.
    addi r14,r0,PrintPar    ;Lade nun die Ausgabeparameter in R14.
    trap 5                  ;Starte Ausgabe.
    trap 0                  ;Exit.
```

c) welches das Produkt der beiden Zahlen berechnet, wobei das Ergebnis ins Gleitkommazahlen-Format umgewandelt werden soll.

Lösung:

```
    .data
Prompt1: .asciiz "\nEnter first number:"
    .align 4
Prompt2: .asciiz "\nEnter second number:"
    .align 4
PrintFt: .asciiz "The result of (%d * %d) in float is %g."
    .align 6
PrintPar: .word PrintFt Int1: .space 4 Int2: .space 4 PrintRes:
.space 8

    .text
    .global main
main:
    addi r1, r0, Prompt1
    jal InputUnsigned       ;Eingabe der ersten Zahl.
    sw Int1, r1             ;Speichere erste Zahl in Int1.
    add r2, r0, r1          ;Speichere erste Zahl in R2.
    addi r1, r0, Prompt2
    jal InputUnsigned       ;Eingabe der zweiten Zahl.
    sw Int2, r1             ;Speichere zweite Zahl in Int2.
    mult r3, r1, r2         ;Multipliziere beide Zahlen. Ergebnis in R3.
    movi2fp f2, r3          ;Bewege Ergebnis in Gleitkommaregister F2.
    cvti2d f2, f2           ;Konvertiere Zahl in F2 in Gleitkomma mit
                            ; doppelter Genauigkeit.
    sd PrintRes, f2         ;Speichere Gleitkommazahl in PrintRes.
    addi r14, r0, PrintPar  ;Ausgabeparameter laden.
    trap 5                  ;Starte Ausgabe.
    trap 0                  ;Exit.
```

d) welches die erste Zahl durch die zweite teilt, wobei die Zahlen vorher in Gleitkommazahlen umformatiert werden sollen.

Lösung:

```
    .data
Prompt1: .asciiz "\nEnter first number:"
    .align 4
Prompt2: .asciiz "\nEnter second number:"
    .align 4
PrintFt: .asciiz "The result of (float)%g / (float)%g is %g.\n"
    .align 6
PrintPar: .word PrintFt Val1: .space 8 Val2: .space 8 PrintRes:
.space 8

    .text
    .global main
main:
    addi r1, r0, Prompt1
    jal InputUnsigned       ;Eingabe der ersten Zahl.
    movi2fp f0, r1          ;Bewege Zahl in Gleitkommaregister F0.
    cvti2d f0, f0           ;Konvertiere Zahl in F0 in doppelt-genaue
                            ;Gleitkomma.
    sd Val1, f0             ;Speichere Zahl in F0 in Val1.
    addi r1, r0, Prompt2    ;Eingabe der zweiten Zahl.
    jal InputUnsigned
    movi2fp f2, r1          ;Bewege Zahl in Gleitkommaregister F2.
    cvti2d f2, f2           ;Konvertiere Zahl in F2 in doppelt-genaue
                            ;Gleitkomma.
    sd Val2, f2             ;Speichere Zahl in F2 in Val2.
    divd f4, f0, f2         ;Führe Division durch. Ergebnis in F4.
    sd PrintRes, f4         ;Speichere Ergebnis in PrintRes.
    addi r14,r0,PrintPar    ;Ausgabeparameter laden.
    trap 5                  ;Starte Ausgabe.
    trap 0                  ;Exit.
```

e) Schreiben Sie ein Programm, welches zu jeder Berechnung aus a) bis d) ein Unterprogramm besitzt, die über das Hauptprogramm aufgerufen werden.

Lösung:

```
    .data
Prompt1: .asciiz "\nEnter first number:"
    .align 4
Prompt2: .asciiz "\nEnter second number:"
    .align 4
StrPtr: .space 4

    .text
;Definiere Label für die Unterprogramme
    .global main
    .global sum
    .global subtr
    .global prod
    .global divis

    ;*********************************************************************
    ;
main:
    addi r1, r0, Prompt1    ;Eingabe der ersten Zahl.
    jal InputUnsigned
    add r2, r0, r1          ;Speichere erste Zahl in R2.
    addi r1, r0, Prompt2
    jal InputUnsigned       ;Eingabe der zweiten Zahl.
    add r3, r0, r1          ;Speichere zweite Zahl in R3.
    jal sum                 ;Rufe Unterprogramm sum auf.
                            ;Rücksprungadresse wird in R31 gespeichert.
    jal subtr               ;Rufe Unterprogramm subtr auf.
    jal prod                ;Rufe Unterprogramm prod auf.
    jal divis               ;Rufe Unterprogramm divis auf.
    trap 0                  ;Exit.

    ;********** Unterprogramm: Sum (Addiert zwei Zahlen in R2 und R3) **********

sum:
    .data
PrintFt: .asciiz "The result of %d + %d is %d.\n"
    .align 6
PrintPar: .word PrintFt Int1: .space 4 Int2: .space 4 PrintSum:
.space 4

    .text
    sw Int1, r2             ;Speichere erste Zahl in Int1.
    sw Int2, r3             ;Speichere zweite Zahl in Int2.
    add r4, r2, r3          ;Addiere Zahlen. Ergebnis in R4.
    sw PrintSum, r4         ;Speichere Ergebnis in PrintSum.
    addi r14,r0,PrintPar    ;Ausgabeparameter laden.
    trap 5                  ;Starte Ausgabe.
    jr r31                  ;Springe zurück auf Adresse in R31.

    ;********** Unterprogramm : Subtr (Subtrahiert zwei Zahlen in R2 und R3) *******

subtr:
    .data
PrintFt2: .asciiz "The result of %d - %d is %d.\n"
    .align 6
PrintPar2: .word PrintFt2 Sub1: .space 4 Sub2: .space 4 PrintRes:
.space 4

    .text
    sw Sub1, r2             ;Speichere erste Zahl in Sub1.
    sw Sub2, r3             ;Speichere zweite Zahl in Sub2.
    sub r4, r2, r3          ;Subtrahiere Zahlen. Ergebnis in R4.
    sw PrintRes, r4         ;Speichere Ergebnis in PrintRes.
    addi r14,r0,PrintPar2   ;Ausgabeparameter laden.
    trap 5                  ;Starte Ausgabe.
    jr r31                  ;Springe zurück auf Adresse in R31.
```

```
    ;****************** Unterprogramm : Prod (Multipliziert zwei Zahlen in R2 und R3) ********

prod:
    .data
PrintFt3: .asciiz "The result of (%d * %d) in float is %g.\n"
    .align 6
PrintPar3: .word PrintFt3 Prod1: .space 4 Prod2: .space 4 PrintRes2:
.space 8

    .text

    sw Prod1, r2            ;Speichere erste Zahl in Prod1.
    sw Prod2, r3            ;Speichere zweite Zahl in Prod2.
    mult r4, r1, r2         ;Multipliziere Zahlen. Ergebnis in R4.
    movi2fp f2, r4          ;Bewege Ergebnis in Gleitkommaregister F2.
    cvti2d f2, f2           ;Konvertiere F2 in doppeltgenaue Gleitkomma.
    sd PrintRes2, f2        ;Speichere Ergebnis in PrintRes2.
    addi r14,r0,PrintPar3   ;Ausgabeparameter laden.
    trap 5                  ;Starte Ausgabe.
    jr r31                  ;Springe zurück auf Adresse in R31.

    ;************* Unterprogramm : Divis (Dividiert zwei Zahlen in R2 und R3) *******

divis:
    .data

PrintFt4: .asciiz "The result of (float)%g / (float)%g is %g.\n"
    .align 6
PrintPar4: .word PrintFt4 Div1: .space 8 Div2: .space 8 PrintRes3:
.space 8

    .text

    movi2fp f0, r2          ;Bewege erste Zahl in Gleitkommaregister F0.
    cvti2d f0, f0           ;Konvertiere F0 in doppeltgenaue Gleitkomma.
    sd Div1, f0             ;Speichere F0 in Div1.
    movi2fp f2, r3          ;Bewege zweite Zahl in Gleitkommaregister F2.
    cvti2d f2, f2           ;Konvertiere F2 in doppeltgenaue Gleitkomma.
    sd Div2, f2             ;Speichere F2 in Div2.
    divd f4, f0, f2         ;Führe Division durch. Ergebnis in F4.
    sd PrintRes3, f4        ;Speichere F4 in PrintRes3.
    addi r14,r0,PrintPar4   ;Ausgabeparameter laden.
    trap 5                  ;Starte Ausgabe.
    jr r31                  ;Springe zurück auf Adresse in R31.
```

Selbsttestaufgabe 1.4 von Seite 31

Die Fakultät einer Zahl ist definiert als:

$$a! = a \cdot (a-1) \cdot (a-2) \cdots 2 \cdot 1$$

Schreiben Sie ein Programm, welches eine Zahl über die Eingabeaufforderung einliest und die Fakultät dieser Zahl ausgibt.

Lösung:

```
;------------------------------------------------------------------------
; Programmanfang bei Symbol main ; Benötigt Modul INPUT ; Liest eine
Zahl von stdin ein und berechnet im Typ double die Faktorielle ;
Abschließend wird das Ergebnis ausgegeben
;------------------------------------------------------------------------

    .data
Prompt: .asciiz     "Ein ganzzahliger Wert >1:"

PrintfFormat: .asciiz   "Die Faktorielle = %g\n\n"
    .align 2
PrintfPar:  .word PrintfFormat PrintfValue: .space 8


    .text
    .global main
main:
    ;*** Wert von stdin in R1 einlesen
    addi r1,r0,Prompt
    jal InputUnsigned

    ;*** Werte initialisieren
    movi2fp f10,r1          ;R1 -> D0 D0..Zählregister
    cvti2d f0,f10
    addi r2,r0,1            ;1 -> D2 D2..Ergebnis
    movi2fp f11,r2
    cvti2d f2,f11
    movd f4,f2              ;1-> D4 D4..Konstante 1

    ;*** Berechnungsschleife abbrechen, wenn D0 1 erreicht
Loop: led f0,f4             ;D0<=1 ?
    bfpt Finish

    ;*** Multiplikation durchführen und nächster Schleifendurchgang
    multd f2,f2,f0
    subd f0,f0,f4
    j Loop

Finish: ;*** Ergebnis ausgeben
    sd PrintfValue,f2
    addi r14,r0,PrintfPar
    trap 5

    ;*** Programmende
    trap 0
```

Selbsttestaufgabe 1.5 von Seite 44

Es sei folgende Programmsequenz gegeben:

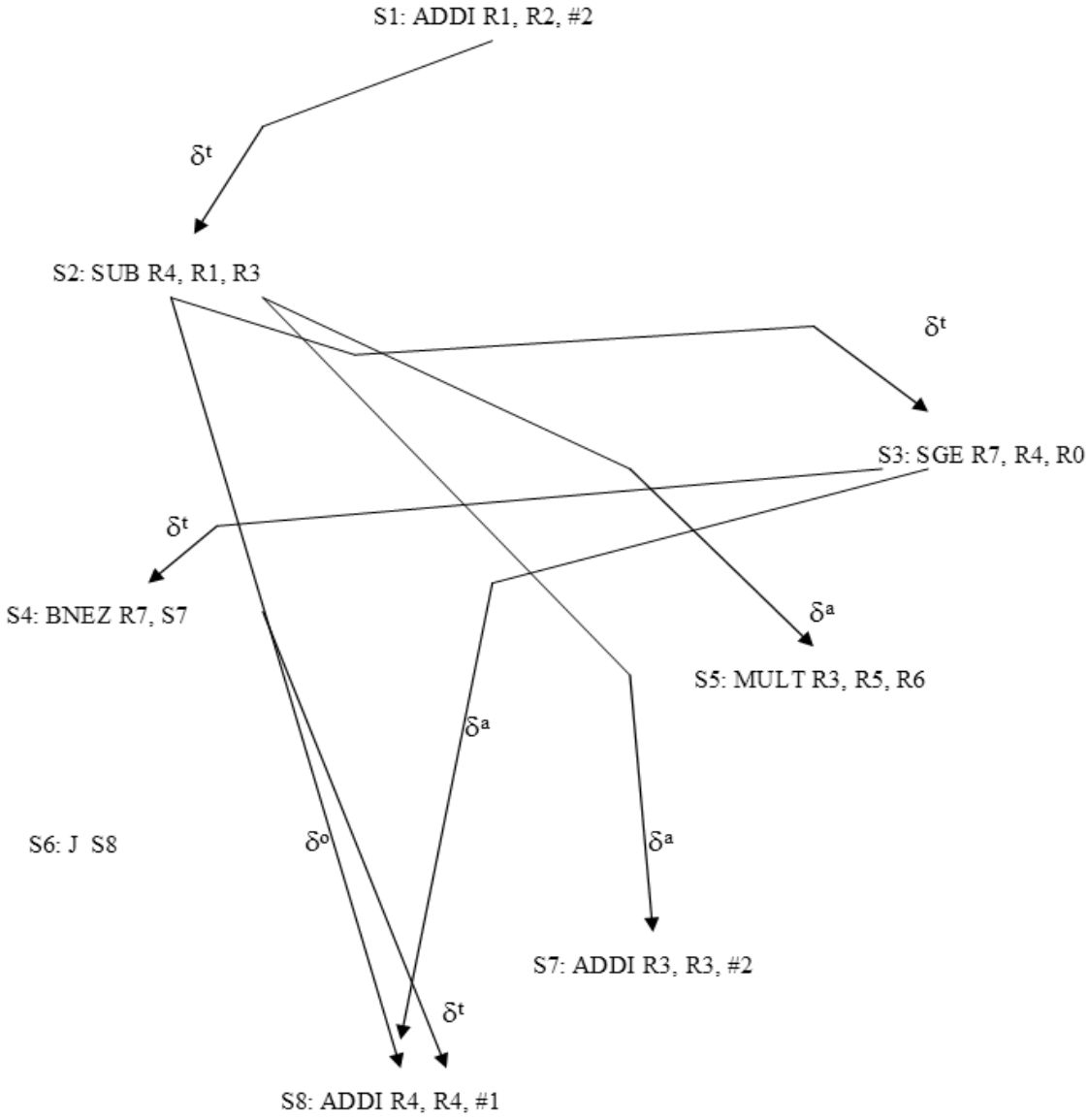
```
S1: ADDI    R1, R2, #2 ; R1 = R2 + 2
S2: SUB     R4, R1, R3 ; R4 = R1 -- R3
S3: SGE     R7, R4, R0 ; R4 >= 0 ? Status in R7
S4: BNEZ    R7, S7     ; wenn ja, gehe zu S7
S5: MULT    R3, R5, R6 ; R3 = R5 * R6
S6: J       S8         ;Goto S8
S7: ADDI    R3, R3, #2 ; R3 = R3 + 2
S8: ADDI R4, R4, #1    ; R4 = R4 +1
```

Bestimmen Sie alle Daten- und Steuerflussabhängigkeiten in dieser Programmsequenz. Stellen Sie diese in einem Abhängigkeitsbaum dar.

Lösung:

δ^t: echte Datenabhängigkeit, δ^a: Gegenabhängigkeit, δ^o: Ausgabeabhängigkeit

Datenabhängigkeiten:



Steuerflussabhängigkeiten: Nur die bedingte Sprunganweisung S4 erzeugt eine Kontrollabhängigkeit. Allerdings gibt es noch eine weitere Steuerflussänderung bei der unbedingten Sprunganweisung S6.

Selbsttestaufgabe 1.6 von Seite 60

Wie in Selbsttestaufgabe 1.4 sei folgende Programmsequenz gegeben:

```
S1: ADDI    R1, R2, #2  ;R1 = R2 + 2
S2: SUB     R4, R1, R3  ;R4 = R1 -- R3
S3: SGE     R7, R4, R0  ;R4 >= 0 ? Status in R7
S4: BNEZ    R7, S7      ;wenn ja, gehe zu S7
S5: MULT    R3, R5, R6  ;R3 = R5 * R6
S6: J       S8          ;Goto S8
```

```
S7: ADDI    R3, R3, #2  ;R3 = R3 + 2
S8: ADDI    R4, R4, #1  ;R4 = R4 +1
```

Es sei angenommen, dass bei der Ausführung der oben abgebildeten Programmsequenz die einzelnen Befehle in einer fünfstufigen DLX-Befehls-Pipeline (siehe Vorlesung) verarbeitet werden (Hinweis: Bei einem Sprungbefehl wird der Program-Counter bereits nach der 4. Stufe auf das neue Sprungziel gesetzt).

Fügen Sie die zur Beseitigung aller Pipelinekonflikte die minimale Anzahl von erforderlichen NOP-Befehlen in die Programmsequenz ein.

Lösung:

```
S1: ADDI    R1, R2, #2 ; R1 = R2 + 2
    NOP
    NOP
S2: SUB     R4, R1, R3 ; R4 = R1 - R3
    NOP
    NOP
S3: SGE     R7, R4, R0 ; R4 >= 0 ? Status in R7
    NOP
    NOP
S4: BNEZ    R7, S7     ; wenn ja, gehe zu S7
    NOP
    NOP
    NOP
S5: MULT    R3, R5, R6 ; R3 = R5 * R6
S6: J       S8         ;Goto S8
    NOP
    NOP
    NOP
S7: ADDI    R3, R3, #2 ; R3 = R3 + 2
S8: ADDI    R4, R4, #1 ; R4 = R4 +1
```

Selbsttestaufgabe 1.7 von Seite 60

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

Lösung: Pipeline-Konflikte: Datenkonflikte, Struktur-/Ressourcenkonflikte und Steuerflusskonflikte.

Datenkonflikte: Lese-nach-Schreib-Konflikt, Schreib-nach-Lese-Konflikt und Schreibe-nach-Schreib-Konflikt

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

Lösung: Keine Pipelinekonflikte ergeben sich bei *Anti Dependence* und *Output Dependence*.

Bei der *Anti Dependence* 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, sodass auch die *Output Dependence* zu keinem Konflikt führt.

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

Lösung: 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) Was sind die charakteristischen Eigenschaften einer RISC-Architektur?

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

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

Lösung: Alle komplexen Adressierungsarten können durch einfache Adressierungsarten ersetzt werden. Allerdings sind dafür mehrere Befehle notwendig.

Kapitel 2

Hochperformante Prozessoren

Kapitelinhalt

Zusammenfassung

Vertiefend zu den Pipelining-Techniken im Kapitel 1 werden in diesem Kapitel die Implementierungstechniken der Prozessoren behandelt, die in heutigen PCs und Workstations eingesetzt werden. Kapitel 2 gliedert sich in zwei grundlegende Abschnitte:

Der erste Abschnitt gibt eine Einführung in die Grundtechniken heutiger Prozessoren. Insbesondere wird die Superskalartechnik behandelt, die in allen heutigen PC- und Workstation-Prozessoren (mit Ausnahme des Intel Itanium) eingesetzt wird. Die VLIW-Technik (*Very Long Instruction Word*) steuert hauptsächlich Signalprozessoren. Die EPIC-Technik (*Explicit Parallel Instruction Computing*), die aus der VLIW-Technik hervorgeht, kommt vor allem in den VLIW-/Superskalarprozessoren der Intel Itanium-Familie zum Einsatz.

Der zweite Abschnitt behandelt die Pipeline-Stufen der Superskalartechnik im Detail. Für jede Pipeline-Stufe werden die Entwurfsalternativen vorgestellt und gegeneinander abgewogen. Die Techniken der dynamischen Sprungvorhersage nehmen breiten Raum ein, da eine exzellente Sprungvorhersage für die Verarbeitungsleistung eines Superskalarprozessors außerordentlich wichtig ist und außerdem diese Sprungvorhersagetechniken beispielhaft für die Spekulations- und Vorhersagetechniken stehen, die in vielen Gebieten der Informatik derzeit neu entwickelt werden.

Lernziele

Die Lernziele dieses Kapitels sind das Kennenlernen von:

- Grundtechniken heutiger Prozessoren,
- Aufbau superskalarer Pipelines,
- Befehlzuordnung und -verplanung,
- Befehlsbearbeitung außerhalb der Programmreihenfolge,
- Sprungvorhersage und spekulative Ausführung,
- Einfluss scheinbarer Datenabhängigkeiten,
- Bedeutung der Decodierung und Registerumbenennung,
- Gewährleistung der sequentiellen Programmsemantik.

2.1 Grundtechniken heutiger Prozessoren

2.1.1 Von skalaren RISC- zu Superskalarprozessoren

Die Grundlage dafür, dass (Einchip-)Prozessoren überhaupt möglich wurden, ist die stetig steigende Integrationsdichte bei den Halbleiter-Technologien. Der erste (Mikro-)Prozessor Intel 4004 entstand 1971 und enthielt nur 2300 Transistoren. Der schon wesentlich komplexere Intel 8086-Mikroprozessor von 1978 bestand bereits aus 20.000 Transistoren. Die Anzahl der Transistoren pro Prozessor-Chip steigerte sich nun rapide. Anfang der 80er Jahre waren dies 134.000 Transistoren mit einer Taktrate von 16 MHz (Intel 80286) und Mitte der 80er Jahre bereits 275.000 Transistoren mit 32 MHz Taktrate (Intel 80386).

Es gab in den 80er Jahren zwei Linien von Mikroprozessoren:

- Die CISC-Mikroprozessoren mit den Familien der Intel 80x86- und der Motorola 680x0-Prozessoren als wichtigste Vertreter. Diese orientierten sich in ihren Architekturen an den Großrechnerarchitekturen der 70er Jahre. Im Laufe der 80er Jahre konnten bei diesen Prozessorfamilien insbesondere die Gleitkommaeinheit und Steuerfunktionen für die Verwaltung des Arbeitsspeichers und die Unterstützung des Betriebssystems zusätzlich auf dem Prozessorchip untergebracht werden.

- Die (skalaren) RISC-Mikroprozessoren mit den Familien der MIPS- und Sun SPARC-Prozessoren. Diese legten besonderen Wert auf Pipelining und auf im Vergleich zu den CISC-Mikroprozessoren relativ große Registersätze.

Beide Prozessorlinien nutzten zwar Pipelining in verschiedenen Ausprägungen, konnten jedoch nur maximal einen Befehl pro Pipelinestufe verarbeiten. Ende der 80er Jahre war ein Integrationsgrad von 1,2 Millionen (M) Transistoren pro Prozessor-Chip (Beispiel: Intel 80486) erreicht, der ab Anfang der 90er Jahre den Siegeszug der superskalaren Prozessoren ermöglichte. Die Grundidee dabei war, die Einschränkung der skalaren RISC-Prozessoren mit Einmalzuweisung (*Single Issue*) zu überwinden und mehr als einen Befehl pro Takt zu holen, zu decodieren, den Ausführungseinheiten zuzuweisen, auszuführen und die Ergebnisse zurückzuschreiben.

Abbildung 2.1 zeigt die Entwicklung der Technologie der Mikroprozessoren in den 90er Jahren anhand der bedeutesten Mikroprozessoren, die den PC- und Workstation-Bereich dominierten. Darin bedeuten: M – Mega oder Million und $\mu = \mu\text{m} = 10^{-6}$ m als Einheit der Kanallänge eines Transistors.

Man erkennt in der Abbildung, dass sich die Anzahl der Transistoren dieser wichtigen Prozessoren (nur) linear von ca. 1,3 bis 9,5 Millionen entwickelte. Gleichzeitig wurde jedoch die Arbeitsfrequenz stärker von 66 auf 550 MHz gesteigert und die Strukturbreite (Kanallänge) in diskreten Schritten von 0,8 auf 0,25 μm reduziert.

Die kommerziell dominierenden superskalaren Mikroprozessoren von Intel wie der zweifach superskalare Pentium von 1993, der PentiumPro von 1995, der Pentium II, Pentium III und Pentium 4 setzen die Entwicklungslinie der Intel

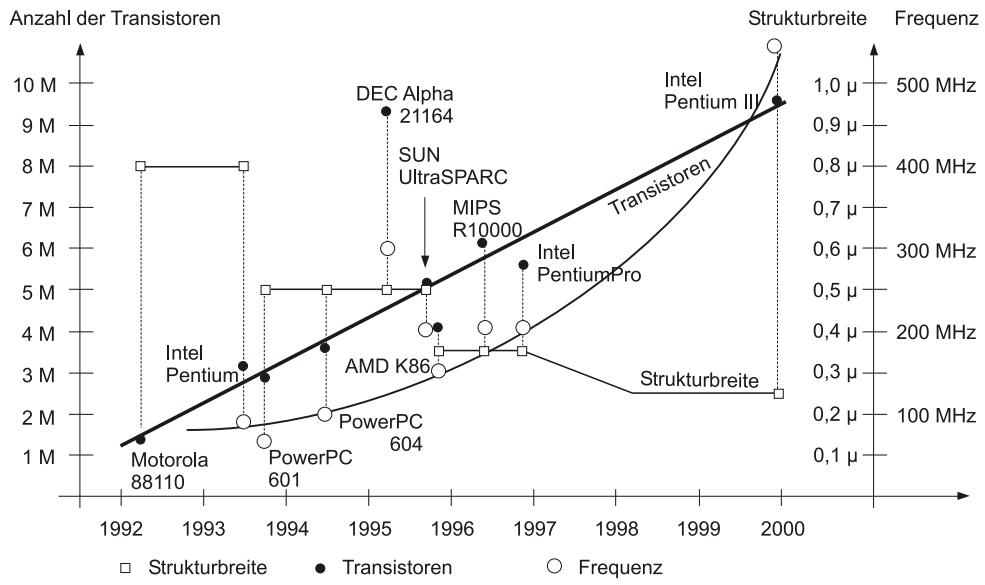


Abbildung 2.1: Prozessorentwicklung in den 90er Jahren.

x86-Architektur fort. Diese Prozessoren werden deshalb als CISC-Mikroprozessoren eingeordnet. Einige weitere Firmen – insbesondere AMD (Advanced Micro-Devices) mit der Athlon-Familie – entwickelten Intel-kompatible Prozessoren. Diese CISC-Mikroprozessoren benötigen eine etwas komplexere Pipeline als die superskalaren RISC-Prozessoren mit zusätzlichen Stufen für die Decodierung der x86-Befehle in so genannte *Micro-Ops*. All diese Intel-Mikroprozessoren besitzen eine 32-Bit-Architektur und sind für den Einsatz in Personal Computern gedacht. Die neueren superskalaren RISC-Prozessoren weisen eine 64-Bit-Architektur auf und sind für den Server-Betrieb konzipiert. Intel brachte mit dem Itanium einen ersten Prozessor mit IA-64-Architektur (IA steht für *Intel Architecture*) auf den Markt, der ebenfalls eine Wortbreite von 64 Bit implementiert.

Der heutige Stand der Chip-Technologie ist gekennzeichnet durch ca. 800 M Transistoren, 0,045 μm (d.h. 45 nm – Nanometer) Strukturbreite und bis zu 3,5 GHz Taktrate.

Heutige Hochleistungsmikroprozessoren und Signalprozessoren arbeiten mit Mehrfachzuweisungstechniken (*Multiple Issue*). Zu diesen Techniken gehören die Superskalartechnik, die VLIW-Technik (*Very Long Instruction Word*), die im Wesentlichen bei Signalprozessoren und bei Multimediaprozessoren angewandt wird, und die aus VLIW entwickelte EPIC-Technik (*Explicitly Parallel Instruction Computing*). Das von Hewlett-Packard und Intel entwickelte EPIC-Format ist das neue Befehlsformat der IA-64 und ist im Itanium-Prozessor verwirklicht.

2.1.2 Komponenten eines superskalaren Prozessors

Heutige Mikroprozessoren nutzen Befehlsebenen-Parallelität durch eine vielstufige Prozessor-Pipeline und durch die Superskalar- oder die VLIW-/EPIC-Technik. Ein superskalarer Prozessor kann aus einem sequenziellen Befehls-

strom mehrere Befehle pro Takt den Verarbeitungseinheiten zuordnen und ausführen, ein VLIW-/EPIC-Prozessor erreicht die gleiche Zuordnungsbandbreite mittels per Compiler gebündelter Befehle.

Solch ein superskalarer RISC-Prozessor besitzt eine Lade/Speicher-Architektur mit einem festen Befehlsformat und einer Befehlslänge von 32 Bit. In Abbildung 2.2 wird ein RISC-Prozessor am Beispiel eines PowerPC-Prozessors dargestellt.

Superskalarer RISC-Prozessor

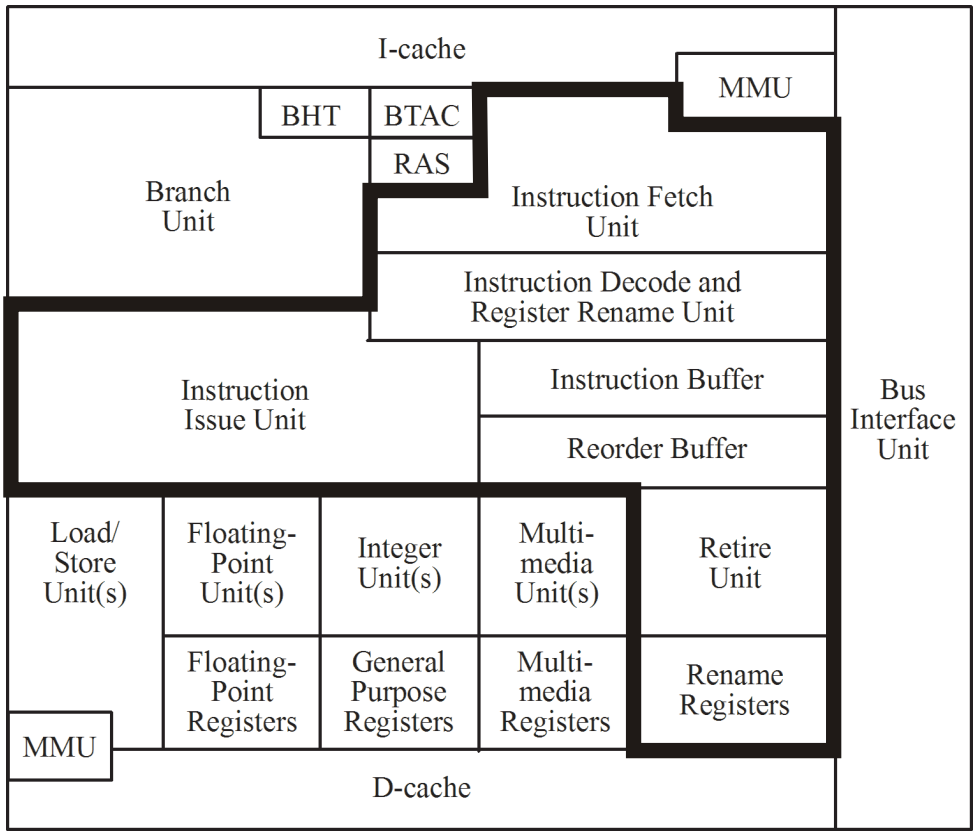


Abbildung 2.2: Komponenten eines superskalaren Prozessors.

Die gezeigten Komponenten haben die folgenden Funktionen:

- Die internen **Speichereinheiten** (*D-Cache, I-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 Arbeitsspeicher, aus denen insbesondere die auszuführenden Befehle geholt werden müssen. Diese Befehle werden dann im internen Code-Cache-Speicher (I-Cache) abgelegt. Die Berechnung der physikalischen Befehlsadresse aus der logischen Speicheradresse wird durch eine Speicherverwaltungseinheit (*Memory Management Unit* – MMU) unterstützt (vgl. Kapitel 3).

- Eine **Verzweigungseinheit** (*Branch Unit*) überwacht die Ausführung von Sprungbefehlen. Nach dem Holen eines Sprungbefehls aus dem Code-Cache-Speicher (*I-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.

- 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 (*D-Cache*) in eines der Arbeitsregister 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 (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. Dazu gehören:

  - Eine oder mehrere **Integer-Einheiten** (*Integer Units*) führen die arithmetischen und logischen Befehle auf den allgemeinen Registern (*General Purpose Register*) 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. Manchmal gibt es auch spezielle Divisions- oder Wurzelberechnungseinheiten, die wegen ihrer langen Latenz von 17 oder mehr Takten nicht als Pipeline realisiert sind.
  - 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 sogar 256 Bit breite Multimediaregister (*Multimedia Register*) 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 (*Floating-Point Registers*). Üblicherweise wird eine 64-Bit-Gleitkommaarithmetik nach IEEE-754-Standard angewandt (vgl. Unterabschnitt 1.2.2). Die Gleitkommaeinheit ist meist als Pipeline mit einer Latenz von drei und einem Durchsatz von eins implementiert. Manchmal bleiben komplexere Befehle – wie eine Kombination aus Multiplizieren und anschließendem Addieren – länger in der Pipeline und führen damit zu einem geringeren Durchsatz.

Am internen Bussystem hängen auch der Befehlspuffer (*Instruction Buffer*) und der sog. Rückordnungspuffer (*Reorder Buffer*). Auf diese Komponenten sowie alle weiteren Einheiten, die in Abbildung 2.2 durch eine breite Umrahmung hervorgehoben sind, aber noch nicht beschrieben wurden, wird im folgenden Unterabschnitt ausführlich eingegangen.

2.1.3 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. Außerdem werden zusätzlich eine Zuordnungs- (*Issue*) und eine Rückordnungs- und Rückschreibestufe (*Retire and Write Back*) sowie weitere Puffer zur Entkopplung der Pipelinestufen benötigt (Abbildung 2.3).

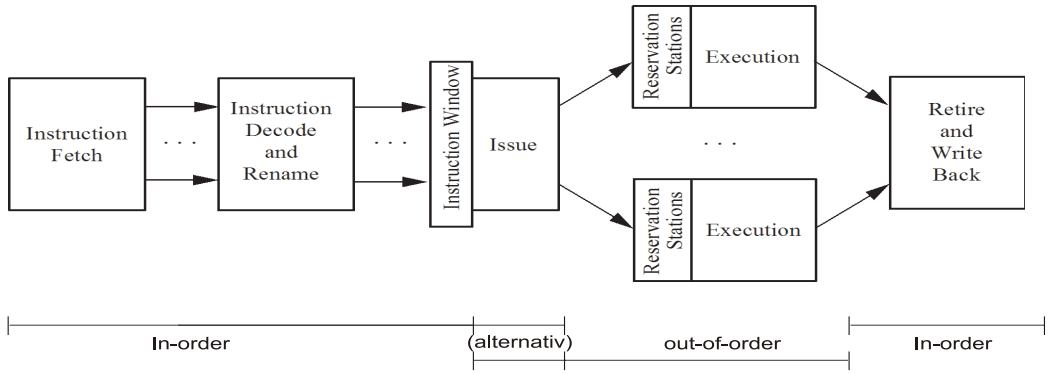


Abbildung 2.3: Superskalare Pipeline.

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

- Die erste *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 so genannte *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.

Nun werden die Aufgaben der einzelnen Pipelinestufen erklärt.

- Die am Beginn der Pipeline stehende Befehlsholestufe (*Instruction Fetch* – IF) lädt gleichzeitig mehrere Befehle aus dem Code-Cache. Die Einheit, die dies ausführt, wird als ***Instruction Fetch Unit*** bezeichnet. 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, die in der Befehlsholestufe zum Tragen kommt. Um die Pipeline bei Sprüngen nicht anhalten zu müssen, werden im Sprungzieladress-Cache die Adressen der Sprungbefehle und die dazugehörigen Sprungziele gespeichert. Ein Befehlsholepuffer entkoppelt die Befehlsholestufe von der Decodierstufe.

- In der Decodierstufe (*Instruction Decode* – ID) wird eine Anzahl von Befehlen decodiert (typischerweise genauso viele wie die maximale Zuordnungsbandbreite beträgt). Die Operanden- und Resultatregister werden umbenannt (*Rename*). Dazu werden die in den Befehlen spezifizierten Architekturregister auf die physikalisch vorhandenen Register (bzw. die Umbenennungspufferregister) abgebildet. Die Einheit, die diese Aufgaben erfüllt, wird ***Instruction Decode and Register Renaming Unit*** genannt. Danach werden die Befehle in einen Befehlspuffer geschrieben, der oft als Befehlsfenster (*Instruction Window* oder *Instruction Pool*) bezeichnet wird. Die Befehle in dem 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. Sie wird auch als ***Instruction Issue Unit*** bezeichnet. Die Programmreihenfolge der zugewiesenen Befehle wird im Rückordnungspuffer, dem ***Reorder Buffer***, abgelegt. Die Befehle können in der sequenziellen Programmreihenfolge (*in Order*) oder außerhalb der Reihenfolge (*out of Order*) zugewiesen werden. Die Prüfungen auf Daten- und Strukturkonflikte können in einer Pipeline-Stufe geschehen oder sie können auf

verschiedene Pipeline-Stufen aufgeteilt werden. Wenn die Strukturkonflikte vor den Datenkonflikten geprüft werden, geschieht die Zuweisung der Befehle in so genannte **Umordnungspuffer** (*Reservation Stations*), die sich vor den Ausführungseinheiten befinden, in Abbildung 2.2 aber nicht gezeichnet wurden.

- In den Umordnungspuffern verbleiben die Befehle, bis die Operanden verfügbar und die Ausführungseinheiten nicht mehr beschäftigt (frei) sind. Abhängig vom Prozessor gehören die Umordnungspuffer zu einer Gruppe von Ausführungseinheiten (Beispiele sind die Intel Pentium-Prozessoren) oder jede Ausführungseinheit hat seine eigenen Umordnungspuffer (PowerPC-Prozessoren). Im letzten Fall kommt es zu einem Strukturkonflikt, wenn mehr als ein Befehl an einen der Umordnungspuffer der gleichen Ausführungseinheit zugewiesen werden soll.

- Der Übergang der Befehle vom Warten zur Ausführung wird als *Dispatch* bezeichnet. *Dispatch* und Ausführung der Befehle geschehen außerhalb der Programmreihenfolge. Sollten bei der Zuordnung schon alle Operanden verfügbar sein und die Ausführungseinheit ist frei, so kann die Ausführung des Befehls schon direkt im folgenden Takt begonnen werden. Deshalb spricht man beim *Dispatch* auch nicht von einer eigenen Pipeline-Stufe. Ein Befehl kann daher keinen oder mehrere Takte im Umordnungspuffer verbringen.

- 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 (*Completion*) geschieht außerhalb der Programmreihenfolge. Während der Vervollständigung werden die Umordnungspuffer bereinigt und der Zustand der Ausführung im Rückordnungspuffer vermerkt. Der Zustand eines Eintrags im Rückordnungspuffer kann eine aufgetretene Unterbrechung anzeigen oder auch einen *vollständigen* Befehl, der jedoch noch von einer Spekulation abhängt.

- Nach der Vervollständigung werden die Befehlsresultate in der Programmreihenfolge gültig gemacht (*committed*). Ein Befehlsresultat kann gültig gemacht werden, 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 in den Architekturregistern *dauerhaft* gemacht, gewöhnlich durch das Rückschreiben aus den Umbenennungsregistern. Oft

wird dies in einer eigenen Stufe gemacht, was dazu führt, dass die Umbenennungsregister erst einen Takt nach der Vervollständigung freigegeben werden.

- Wenn eine Unterbrechung auftritt, werden die Resultate aller Befehle, die in der Programmreihenfolge vor dem Ereignis stehen, gültig gemacht und diejenigen aller nachfolgenden Befehle verworfen. Damit wird eine *präzise Unterbrechung* (vgl. Unterabschnitt 1.5.5 in Kapitel 1) garantiert. Abhängig von der Prozessorarchitektur und der Art der Unterbrechung, wird das Resultat des verursachenden Befehls noch gültig gemacht oder verworfen, ohne weitere Auswirkungen zu haben.

- Das Freigeben eines Platzes im Rückordnungspuffer wird als Rückordnung (*Retirement*) bezeichnet, unabhängig davon, ob das Befehlsresultat gültig gemacht oder verworfen wurde. Die Einheit, die diese Aufgabe erfüllt, wird ***Retire Unit*** genannt.[1] Das Gültigmachen eines Resultats geschieht entweder durch Ändern der Registerabbildung oder durch Rückschreiben des Resultats aus dem Umbenennungspufferregister in das Architekturregister.

2.1.4 Präzisierung des Begriffs „superskalar“

Der Begriff „superskalar“ wurde erstmals 1987 erwähnt. Das Zitat hier stammt von [5]: „*Superscalar machines are distinguished by their ability to (dynamically) issue*[2] *multiple instructions each clock cycle from a conventional linear instruction stream.*“

Die Bedeutung des Begriffs superskalar kann wie folgt präzisiert 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).

[1] Die deutschen Begriffe Zuordnung (*Issue*), zweite Zuordnungsstufe (*Dispatch*), Vervollständigung (*Completion*), Gültigmachung (*Commitment*) und Rückordnung (*Retirement*) sind keine wörtlichen Übersetzungen der englischen Termini, sondern beschreiben das funktionelle Verhalten. In den englischsprachigen wissenschaftlichen Arbeiten werden die genannten Begriffe uneinheitlich verwendet. Zum Beispiel wird in den am PowerPC-orientierten Arbeiten die Zuordnung *Dispatch* statt *Issue* und die Gültigmachung *Completion* statt *Commitment* genannt.

[2] Der Begriff *Issue* ersetzt hier den Begriff *Dispatch* aus dem Original.

- 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. Dies ist z.B. der Fall beim skalaren microSPARC-II- und den superskalaren SuperSPARC- und UltraSPARC-Prozessoren von Sun.

Der Begriff „superskalar“ wird oft in einer etwas weniger genauen Form benutzt, um Prozessoren mit mehreren parallelen Pipelines oder mehreren Ausführungseinheiten zu beschreiben[3]. Beide Varianten erlauben es jedoch nicht, zwischen der Superskalar- und der VLIW-Technik zu unterscheiden.

feinkörnige Parallelität

Das Befehls-Pipelining und die Superskalartechnik nutzen beide die so genannte feinkörnige Parallelität (*Fine-grain Parallelism* oder *Instruction-level Parallelism*), d.h. Parallelität zwischen einzelnen Befehlen.

zeitliche vs. räumliche Parallelität

Das Pipelining nutzt dabei zeitliche Parallelität (*Temporal Parallelism*) und die Superskalartechnik die räumliche Parallelität (*Spatial Parallelism*). Eine Leistungssteigerung durch zeitliche Parallelität kann mit einer längeren Pipeline und „schnelleren“ Transistoren (höherer Taktfrequenz) erreicht werden. Falls genügend feinkörnige Parallelität vorhanden ist, kann die Leistung durch räumliche Parallelität im superskalaren Fall mit Hilfe von mehr Ausführungseinheiten und einer höheren Zuweisungsbandbreite erreicht werden.

2.1.5 Die VLIW-Technik

Mit **VLIW** (*Very Long Instruction Word*) wird eine Architekturtechnik bezeichnet, bei der ein Compiler eine feste Anzahl von einfachen, voneinander unabhängigen Befehlen zu einem Befehlspaket zusammenpackt und in *einem* Maschinenbefehlswort meist fester Länge speichert. Das Maschinenbefehlsformat eines VLIW-Befehlspakets kann mehrere hundert Bits lang sein, in der Praxis sind dies zwischen 128 und 1024 Bits.

Alle Befehle innerhalb eines VLIW-Befehlspakets müssen unabhängig voneinander sein und eigene Opcodes und Operandenbezeichner enthalten. Damit

[3] [11] definierte „superskalar“ wie folgt: „*A superscalar processor reduces the average number of cycles per instruction beyond what is possible in a pipelined, scalar RISC processor by allowing concurrent execution of instructions in the same pipeline stage, as well as concurrent execution of instructions in different pipeline stages. The term superscalar emphasizes multiple, concurrent operations on scalar quantities, as distinguished from multiple, concurrent operations on vectors or arrays as is common in scientific computing.*“

unterscheidet sich ein VLIW-Befehlspaket von einem CISC-Befehl, der mit einem Opcode mehrere, eventuell sequenziell nacheinander ablaufende Operationen codieren kann. Weiterhin sind die Operationen innerhalb eines VLIW-Befehlspakets in der Regel verschiedenartig. Das unterscheidet VLIW-Befehlspakete von den SIMD-Befehlen (*Single Instruction Multiple Data*) wie beispielsweise den Multimediabefehlen, bei denen ein *einziger* Opcode eine gleichartige Operation auf einer Anzahl von Operanden(paaren) auslöst.

Die Anzahl der Befehle in einem VLIW-Befehlspaket ist in der Regel fest. Wenn die volle Bandbreite eines VLIW-Befehlspakets nicht ausgenutzt werden kann, muss es mit Leerbefehlen aufgefüllt werden. Neuere VLIW-Architekturen sind in der Lage, durch ein komprimiertes Befehlsformat auf das Auffüllen mit Leerbefehlen zu verzichten.

VLIW-Prozessor

Ein **VLIW-Prozessor** besteht aus einer Anzahl von Ausführungseinheiten, die jeweils eine Maschinenoperation taktsynchron zu den anderen Maschinenoperationen eines VLIW-Befehlspakets ausführen können, wobei ein VLIW-Befehlspaket so viele einfache Befehle umfasst, wie Ausführungseinheiten in dem VLIW-Prozessor vorhanden sind. Der Prozessor startet im Idealfall in jedem Takt ein VLIW-Befehlspaket. Die Befehle in einem solchen VLIW-Befehlswort werden dann gleichzeitig geholt, decodiert, zugewiesen und ausgeführt. In Abhängigkeit von der Anzahl n der Befehle, die gemeinsam durch die Pipeline fließen, also auch entsprechend der maximalen Anzahl n von Befehlen in einem Befehlspaket spricht man von einem ***n*-fachen VLIW-Prozessor**.

Ein VLIW-Prozessor führt keine dynamische Befehlszuordnung durch, sondern ist auf die (statische) Befehlsanordnung im Befehlswort durch den Compiler angewiesen. Diese Befehlsanordnung wird von der Zuordnungseinheit nicht geändert, mit der Folge, dass die Hardware-Komplexität der Zuordnungseinheit, verglichen mit derjenigen bei der Superskalartechnik, wesentlich geringer ist. So fällt zum Beispiel das Überprüfen von Datenkonflikten und die Erkennung der Parallelität auf Befehlsebene durch die Zuordnungs-Hardware weg. Die Anordnung der einzelnen Operationen einschließlich der Speicherzugriffe erfolgt bereits durch den Compiler. Die Anwendung einer Speicherhierarchie aus Cache- und Hauptspeichern wird damit erschwert. Dynamische Ereignisse, wie z.B. Cache-Fehlzugriffe, führen zum Stillstand der nachfolgenden Pipeline-Stufen. Es wird ferner vorausgesetzt, dass alle Operationen die gleiche Ausführungszeit haben. Eine spekulative Ausführung von Befehlen nach bedingten Sprüngen wird nicht von der Hardware organisiert, sondern ist auf Compiler-

Trace Scheduling

techniken wie das so genannte *Trace Scheduling* angewiesen. Die Ablaufreihenfolge der VLIW-Befehlspakete ist fest, eine Ausführung auch außerhalb der Programmreihenfolge ist nicht möglich.

Der Datenverkehr zwischen Registersatz und Datenspeicher erfolgt über Lade-/Speicherbefehle. Die Operanden werden einem allen Ausführungseinheiten zugänglichen Registersatz entnommen und in diesen werden auch die Resultate gespeichert. Der Registersatz ist zu diesem Zweck als Mehrkanalspeicher (*Multi-Port Memory*) ausgeführt.

Die bekanntesten VLIW-Prozessoren waren die Prozessoren der Multiflow TRACE-Rechnerfamilie aus der zweiten Hälfte der 80er Jahre, die Befehlswörter von 256, 512 und 1024 Bit Breite besaßen. Mit einem 1024 Bit breiten Be-

fehlswort wurden bis zu 28 Operationen pro Takt zur Ausführung angestoßen. Die im Markt zunächst ganz gut eingeführte Multiflow TRACE-Rechnerfamilie war wegen ihrer im Vergleich zum Hardware-Aufwand schwachen Leistung rasch wieder verschwunden. Die 90er Jahre waren geprägt von sehr leistungsfähigen Superskalarprozessoren im Bereich der Universalprozessoren und dem Wiederentdecken der VLIW-Technik für Digitale Signalprozessoren.

Bis in die Mitte der 90er Jahre galt im Bereich der Universalprozessoren die Superskalartechnik gegenüber der VLIW-Technik als überlegen. Dem ist jedoch nicht mehr so. Der Grund dafür ist die hohe Komplexität der Befehlszuordnungseinheit bei Superskalarprozessoren, die einer weiteren Erhöhung der superskalaren Zuordnungsbandbreite *und* der Taktfrequenz entgegensteht. Hier erlaubt die VLIW-Technik durch den einfacheren Hardware-Aufbau der Prozessoren eine größere Zuordnungsbandbreite an mehr Ausführungseinheiten und eine höhere erreichbare Taktfrequenz als vergleichbare Superskalarprozessoren.

heutige VLIWs

Heutige VLIWs existieren vorzugsweise im Bereich der Signalprozessoren wie z.B. die 8-fachen VLIW-Prozessoren TMS-320C6xxx-Prozessor von Texas Instruments. Die gegenüber Superskalarprozessoren geringere Hardware-Komplexität der VLIW-Technik motivierte ein im Jahr 1994 gestartetes, gemeinsames Projekt von Intel und Hewlett Packard (HP), das zur Weiterentwicklung der VLIW-Technik zum EPIC genannten Format der mit IA-64 bezeichneten 64-Bit-Architektur der beiden Firmen führte.

2.1.6 Die EPIC-Technik

EPIC-Format

Das von HP und Intel als **EPIC** (*Explicit Parallel Instruction Computing*) bezeichnete Befehlsformat des IA-64-Befehlssatzes (*Intel Architecture*) ist ein erweitertes Dreibefehlsformat, ähnlich einem dreifachen VLIW-Format. Ziel des EPIC-Ansatzes ist es, den Entwurf von Mikroarchitekturen zu unterstützen, welche die Einfachheit und hohe Taktrate eines VLIW-Prozessors mit den Vorteilen des dynamischen Schedulings verbinden. Dies wird in der IA-64-Architektur durch das EPIC-Format in Verbindung mit expliziten Befehlen zur Unterstützung von Spekulation (Daten- und Sprungspekulation) erreicht. Das EPIC-Format erlaubt es dem Compiler, dem Prozessor die Befehlsparallelität direkt mitzuteilen. Ein EPIC-Prozessor muss im Idealfall keine Überprüfung von Daten- und Steuerflussabhängigkeiten durchführen und unterstützt keine Veränderung der Ausführungsreihenfolge. Damit wird die Mikroarchitektur gegenüber einem Superskalarprozessor stark vereinfacht. EPIC verbessert die Fähigkeit des Compilers, auf statische Weise gute Befehlsanordnungen zu erzeugen.

IA-64-Architektur

Zum IA-64-Architekturansatz gehört weiterhin ein voll prädikativer Befehlssatz, d.h. die Ausführung jedes Befehls kann von einer Bedingung abhängig gemacht werden und bedingte Sprungbefehle lassen sich dadurch vermeiden (s. Abschnitt 2.2.2.6). Außderdem ist er gekennzeichnet durch viele Register – 128 allgemeine Register, 128 Gleitkommaregister, 64 Prädikatregister und 8 Sprungregister – und spekulative Ladebefehle. Die Spekulationsbefehle ermöglichen dem Compiler verschiedene Formen der Codeverschiebungen über Grundblöcke hinaus, die bei konventionellen Prozessoren unzulässig wären.

Ein einzelner EPIC-Befehl der IA-64-Architektur hat eine Länge von 41 Bit und besteht aus einem Opcode, einem Prädikatfeld, zwei Adressen der Quellregister, der Adresse des Zielregisters und weiteren Spezialfeldern. Die IA-64-Befehle werden vom Compiler gebündelt. Ein EPIC-Befehlsbündel besteht bei der IA-64-Architektur aus einem compilererzeugten 128 Bit breiten Befehls-„Bündel“ mit drei IA-64-Befehlen und so genannten *Template*-Bits. Für diese sind im Bündel 5 Bits reserviert, die Informationen zur Gruppierung von Befehlen beinhalten. Es gibt keine Leerbefehle, sondern die Parallelität wird durch die Template-Bits angegeben. Sie geben an, ob ein Befehl mit einem anderen parallel ausgeführt werden kann. Das kann sich auf Befehle innerhalb des gleichen EPIC-Befehlsbündels beziehen, aber auch auf nachfolgende EPIC-Befehlsbündel.

Da auch voneinander daten- oder steuerflussabhängige Befehle vom Compiler in einem Bündel zusammengefasst werden können, ist das EPIC-Format wesentlich flexibler als die VLIW-Formate. Die EPIC-Architektur kann durch variabel breite VLIW- und durch hybride VLIW-/Superskalarprozessoren wie z.B. den Intel/HP Itanium implementiert werden.

Auch eine Skalierbarkeit der Zuordnungsbandbreiten zukünftiger EPIC-Prozessoren ist durch Konkatenation mehrerer Befehlsbündel mit voneinander unabhängigen Befehlen möglich. Ein EPIC-Befehlsbündel mit drei einzelnen Befehlen spricht drei Ausführungseinheiten an. Wenn ein IA-64-Prozessor über n mal drei Ausführungseinheiten verfügt, dann ist es möglich, n EPIC-Befehlsbündel zu verbinden und die enthaltenen Befehle gleichzeitig auszuführen, falls diese unabhängig voneinander sind.

Itanium-Prozessor

Der Itanium-Prozessor ist ein sechsfacher EPIC-Prozessor mit zehnstufiger Pipeline. Im Itanium-Prozessor stehen neun Ausführungseinheiten bereit, dies sind vier ALU/MMX-Einheiten, zwei Fließkommaeinheiten, zwei Lade-/Speichereinheiten und eine Sprungeinheit. Der Itanium konkateniert bei der Ausführung zwei Befehlsbündel mit voneinander unabhängigen Befehlen und führt diese Befehle parallel zueinander in der Pipeline aus. Zukünftige EPIC-Prozessoren können dann beispielsweise drei oder mehr Befehlsbündel miteinander kombinieren. Auf diese Weise ist bei der EPIC-Architektur eine Skalierbarkeit auf zukünftige Prozessoren möglich.

2.1.7 Vergleich der Superskalar- mit VLIW- und EPIC-Technik

Die Superskalar-, VLIW- und EPIC-Techniken verfolgen das gleiche Ziel, nämlich durch die Parallelarbeit einer Anzahl von Ausführungseinheiten eine Leistungssteigerung zu erzielen. Dabei sollen möglichst mit jedem Takt so viele Operationen ausgeführt werden, wie die Maschine Ausführungseinheiten besitzt. Die folgenden Punkte vergleichen die drei Techniken bezüglich verschiedener Kriterien:

- *Architekturtechnik versus Mikroarchitekturtechnik*: VLIW und EPIC sind Architekturtechniken, wohingegen die Superskalartechnik eine Mikroarchitekturtechnik ist. Dies wird z.B. darin deutlich, dass Befehlspakete

mit einem sechsfachen VLIW-Format nicht auf einem vierfachen VLIW-Prozessor ausgeführt werden können. Damit ergibt sich bei der VLIW-Technik für Weiterentwicklungen eines Prozessors das Problem der Objektcode-Kompatibilität. Das gegenüber VLIW flexiblere EPIC-Format lässt in gewissen Umfang eine Skalierbarkeit der Prozessoren zu, da mehrere Befehlsbündel mit unabhängigen Befehlen gleichzeitig ausführbar sind.

- *Befehlsablaufplanung und Konfliktvermeidung*: Compiler für VLIW-Architekturen müssen die Befehlszuordnung an die Ausführungseinheiten planen und die Konfliktvermeidung zwischen Befehlen vornehmen, während dies beim superskalaren Prozessor durch die Hardware geschieht. Aus diesen Gründen sind die Anforderungen an einen Compiler bei der VLIW-Architektur noch komplexer als beim superskalaren Prozessor. Das gilt in noch höherem Maße für die EPIC-Architektur. Ferner ist die VLIW-Maschine wegen ihrer völlig synchronen Arbeitsweise wesentlich starrer als der superskalare Prozessor. Insbesondere kann sie nicht auf Konflikte reagieren, die zur Laufzeit auftreten.

- *Compileroptimierungen*: Für eine optimale Leistung setzen alle drei Architekturformen voraus, dass es einen Compiler gibt, der die Befehle im Sinne einer möglichst guten Ausnutzung der Ausführungseinheiten optimal anordnet. Der Compiler der VLIW- und der EPIC-Prozessoren muss außer den Operationen auch noch den Zeitbedarf der Speicherzugriffe in die Befehlsablaufplanung einbeziehen, während die Speicherzugriffe des superskalaren Prozessors automatisch von seiner Lade-/Speichereinheit vorgenommen werden. Es hat sich gezeigt, dass häufig die gleichen Optimierungsstrategien bei der Codeerzeugung in allen drei Fällen mit Erfolg angewandt werden können.

- *Befehlsanordnung*: Mit der VLIW- und der EPIC-Technik wird eine ähnliche Maschinenparallelität wie durch die Superskalartechnik ermöglicht. Betrachtet man die Ebene der „einfachen“ Maschinenbefehle, dann speist ein superskalarer Prozessor seine Ausführungseinheiten aus nur *einem* Befehlsstrom *einfacher* Befehle, während dies bei einem VLIW-Prozessor ein Befehlsstrom von VLIW-Befehlspaketen, also von Tupeln einfacher Befehle ist. Das EPIC-Format kann auch voneinander abhängige Befehle in einem Bündel vereinen. Diese Abhängigkeiten sind in den Template-Bits codiert und müssen vom Prozessor geprüft werden. Mehrere Bündel mit voneinander unabhängigen Befehlen können gleichzeitig ausgeführt werden. Ein EPIC-Prozessor kann somit als Hybrid aus VLIW- und Superskalarprozessor entworfen werden.

- *Reaktion auf Laufzeitereignisse*: Ein VLIW-Prozessor kann auf statisch nicht vorhersehbare Ereignisse wie z.B. Cache-Fehlzugriffe oder Steuerflussänderungen nicht so flexibel reagieren wie ein Superskalarprozessor. Die starre Ausführung führt dazu, dass Verzögerungen einzelner Operationen durch ein unvorhersehbares Laufzeitereignis auf die Ausführungszeit des gesamten Befehlspakets durchschlagen.

- *Speicherorganisation*: Die Speicherorganisation ist bei der VLIW-Maschine ungünstiger, da der Superskalarprozessor eine Hierarchie aus Cache- und Hauptspeichern verwenden kann, was bei der VLIW-Maschine kaum möglich ist.

- *Sprungvorhersage und Sprungspekulation*: Die ausgefeilten und sehr effizienten Verfahren der dynamischen Sprungvorhersage, die bei heutigen Superskalarprozessoren angewandt werden, können bei VLIW-Prozessoren nicht und bei EPIC-Prozessoren nur erschwert zum Einsatz kommen. Die für die VLIW-Architekturen entwickelten compilerbasierten Verfahren der Sprungspekulation bestehen darin, dass der Compiler zusätzlichen Code in die Befehlspakete einfügt, der beide Sprungrichtungen ausführt und dann, wenn die Sprungrichtung entschieden ist, die Auswirkungen des falsch spekulierten Sprungpfades wieder rückgängig macht. Diese Verfahren sind weniger effizient als die dynamische Sprungspekulationen in Superskalarprozessoren. Eine Abhilfe leistet die Prädikationstechnik (s. Abschnitt 2.2.2.6), die nach einem bedingten Sprung beide Sprungrichtungen bedingt ausführt und nur die Befehlsausführungen auf dem richtigen Sprungpfad gültig macht. Diese Technik wird in heutigen VLIW- und EPIC-Prozessoren eingesetzt, kann jedoch nicht in allen Fällen die bedingten Sprungbefehle ersetzen.

- *Codedichte*: Durch das feste VLIW-Befehlsformat ist die Codedichte immer dann geringer als bei Maschinenprogrammen für Superskalarprozessoren, wenn der Grad der zur Verfügung stehenden Befehlsebenenparallelität die Anzahl der Befehle in einem VLIW-Befehlspaket unterschreitet. Beim EPIC-Format ist dies nicht der Fall, jedoch kommen die Template-Bits pro EPIC-Bündel hinzu.

- *Erzielbare Verarbeitungsleistung und Anwendungsfelder*: Im Idealfall wird in allen drei Fällen eine vergleichbare Verarbeitungsleistung erzielt. Die Einfachheit der VLIW-Prozessoren ermöglicht eine gegenüber der Superskalartechnik höhere Taktrate. Die VLIW-Technik ist bei Code mit sehr hohem Parallelitätsgrad vorteilhaft, da die gegenüber Superskalarprozessoren einfachere Prozessorstruktur VLIW-Prozessoren mit höherer Maschinenparallelität zulässt. Superskalarprozessoren können auf dynamische Ereignisse und Programme mit häufig wechselndem Sprungverhalten besser reagieren. EPIC-Prozessoren stehen hier zwischen Superskalar- und VLIW-Prozessoren. Eine hohe Parallelität auf Befehlsebene ist oft bei Signalverarbeitungsanwendungen und in numerischen Programmen der Fall. Allgemeine Anwendungsprogramme wie Textverarbeitungsprogramme, Tabellenkalkulation, Compiler und Spiele besitzen im Vergleich dazu eine geringe Befehlsebenenparallelität und zeigen ein sehr dynamisches Verhalten, was den Einsatz der Superskalartechnik in Universalmikroprozessoren als geeigneter erscheinen lässt.

Wichtiger Merksatz

VLIW und EPIC sind Architekturtechniken, wohingegen die Superskalartechnik eine Mikroarchitekturtechnik ist.

Die Kombination der VLIW- mit der Superskalartechnik, wie sie durch die Intel-Prozessoren mit EPIC-Architektur geschieht, ist eine interessante Option, um eine sehr hohe Maschinenparallelität in einem Prozessor zu erreichen. Zum einen wird durch die Superskalartechnik die Starrheit des VLIW-Prinzips vermieden und zum anderen die Komplexität der Zuordnungseinheit durch das VLIW-artige EPIC-Format verringert.

2.1.8 Chipsätze

Die Leistungstärke eines Computersystems wird aber nicht nur durch die Prozessoren sondern auch sehr stark durch die Komponenten bestimmt, die den Prozessor mit dem Speicher und den Ein-/Ausgabeeinheiten verbinden. In diesem Abschnitt werden wir kurz in die grundlegende Struktur von Desktop-Systemen oder PCs (*Personal Computer*) einführen. Der Kern eines PCs besteht aus der Hauptplatine, auch *Motherboard* genannt, auf dem der Prozessor, Speicher, Ein-/Ausgabebausteine und -schnittstellen untergebracht sind. Da diese drei Haupteinheiten mit unterschiedlichen Geschwindigkeiten arbeiten, benötigt man *Brückenbausteine* (*Bridges*), welche die vorhandenen Geschwindigkeitsunterschiede ausgleichen und für einen optimalen Datenaustausch zwischen den Komponenten sorgen.

Brückenbausteine

Die Brückenbausteine müssen auf die Zeitsignale (*Timing*) des Prozessors abgestimmt werden. Die Gesamtheit der zu einem Prozessor passenden Brückenbausteine wird als *Chipsatz* bezeichnet. Die Prozessorhersteller (vor allem AMD und Intel) sowie auf die Entwicklung von Chipsätzen spezialisierte Firmen bieten kurz nach dem Erscheinen eines neuen Prozessors auch die dazu passenden Chipsätze an.

Chipsatz

Ein Chipsatz besteht meist aus ein bis zwei Chips, die benötigt werden, um den Prozessor mit dem Speichersystem und Ein-/Ausgabebussen zu koppeln (vgl. Kurseinheit 4, Kurs 1608). Der Chipsatz hat also großen Einfluss auf die Leistungsfähigkeit eines Computersystems und muss daher optimal auf den Prozessor abgestimmt sein. Obwohl ein Chipsatz für eine bestimmte Prozessorfamilie entwickelt wird, kann er meist auch eine Vielzahl kompatibler Prozessoren unterstützen.

Oft findet man beim Chipsatz eine Aufteilung in *North Bridge* und *South Bridge*. Die North Bridge liegt bei normaler Aufstellung des PCs unmittelbar beim Prozessor und oberhalb der South Bridge. Heutige Chipsätze[4] unterstützen die so genannte *Intel Hub Architecture* (IHA), bei der die North Bridge durch den *Memory Controller Hub* (MCH) und die South Bridge durch den *I/O Controller Hub 2* (ICH2) ersetzt wird (s. Abbildung 2.4). Die beiden Bereiche sind durch eine schnelle dedizierte Verbindung, dem *Hub Interface* mit bis zu 4 GB/s, miteinander gekoppelt. Daher ist die IHA auch viel schneller als der frühere North-/South-Bridge-Ansatz, bei dem alle Ein-/Ausgabeschnittstellen über den langsamen PCI-Bus angebunden wurden.

Wie wir aus der Abbildung sehen können, verarbeitet der MCH sehr hohe Taktraten und muss daher gut gekühlt werden. Dagegen kommt der ICH meist ohne Kühlkörper aus. In den heutigen Chipsätzen wird eine neue skalierbare

[4] ab dem Intel i820.

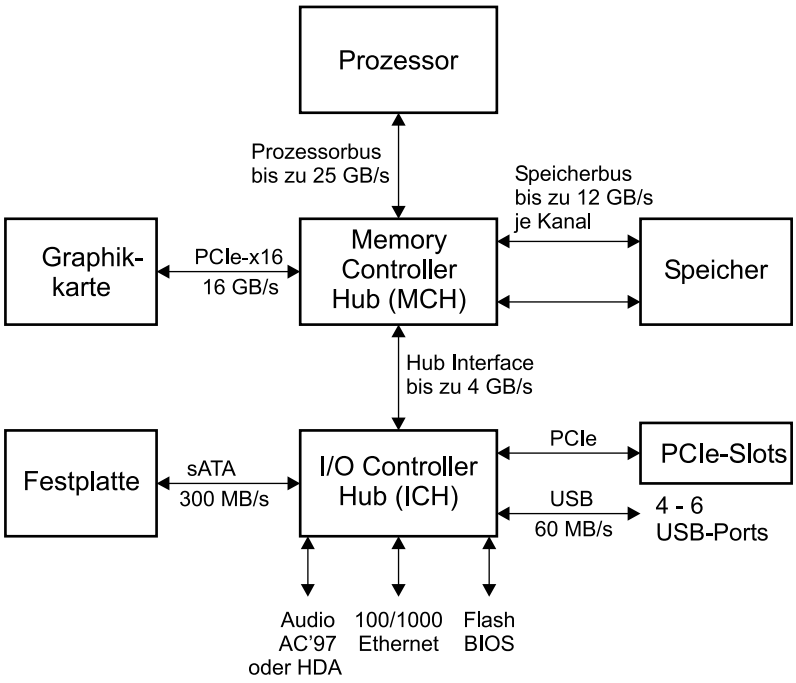


Abbildung 2.4: Chipsätze synchronisieren den Prozessor mit Speicher- und Ein-/Ausgabebussen. Dargestellt ist der Aufbau eines Desktop-Computers nach der Intel Hub Architecture.

Variante des PCI-Busses, der so genannte *PCI-Express* oder PCIe, eingeführt. Die PCIe-Verbindung wird nicht mehr als paralleler Bus ausgeführt, sondern besteht aus einer oder mehreren bidirektionalen seriellen Verbindungen, die als *Lanes* bezeichnet werden. Dadurch spart man sehr viele Verbindungsleitungen. Anstatt bisher 84 Leitungen bei einem herkömmlichen 32 Bit-PCI-Bus werden für eine Lane insgesamt nur vier Leitungen benötigt. Durch diesen drastischen Wegfall von Leitungen vereinfachen sich auch die Steckverbindungen. Durch mehrere parallele PCIe-Lanes kann die Bandbreite leicht den jeweiligen Erfordernissen angepasst werden. Auf diese Weise konnte auch der früher übliche AGP (*Accelerated Graphics Port*) zum Anschluss einer Graphikkarte durch einen PCIe-8x-Port ersetzt werden. Um dem Leistungsbedarf künftiger Graphikanwendungen gerecht zu werden, gab es jedoch bei PCIe von Anfang an nur Graphikports mit 16-facher Geschwindigkeit (PCIe-16x), die eine theoretische Bandbreite von 8 GB/s je Richtung, insgesamt also rund 16 GB/s erreichen.

Im Gegensatz zum PCI-Bus gibt es bei PCIe keine Einsteckplätze (*Slots*) sondern *geschaltete Ports*. Dies bedeutet, dass den einzelnen PCI-Express-Karten stets die volle Bandbreite zur Verfügung steht, da hier keine Zugriffskonflikte wie bei einem Bus auftreten können.

Weitere Informationen über den Aufbau eines PCs und seine Komponenten finden Sie im Kurs 1744 „PC-Technologie". Dort werden auch die in Abbildung 2.4 aufgeführten Schnittstellen und Busse ausführlich beschrieben.

Selbsttestaufgabe 2.1 (Kontrollfragen)

a) Worin besteht der Unterschied zwischen Superpipeline-Prozessoren und superskalaren Prozessoren?

b) Warum ist der Compilerbau für VLIW-Maschinen aufwändiger als für superskalare Prozessoren?

Lösung auf Seite 121

2.2 Die Superskalartechnik

2.2.1 Befehlsbereitstellung

Ein Superskalarprozessor besteht aus einem Befehlsbereitstellungsteil und einem Ausführungsteil. Die beiden Teile werden durch einen Pufferspeicher, das so genannte **Befehlsfenster**, voneinander entkoppelt. Die Aufgabe des Befehlsbereitstellungsteils ist es, den Ausführungsteil mit genügend Befehlen zu versorgen. Der Ausführungsteil wird von der Anzahl der Ausführungseinheiten und der Zuordnungsbandbreite bestimmt. 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 (*Program Counter* – PC) 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.

2.2.1.1 Code-Cache-Speicher

Natürlich muss in jedem Takt ein solcher Befehlsblock bereitgestellt werden. Die dafür verwendete Speichertechnik ist diejenige einer **Harvard-Cache-Architektur**, die sich durch separate Code- und Daten-Cache-Speicher, jeweils eigene Speicherverwaltungseinheiten und separate Zugriffspfade für Befehle und Daten auszeichnet. Separate Code- und Daten-Cache-Speicher werden auf der Ebene der Primär-Cache-Speicher (*L1-Cache*), die auf dem Prozessor-Chip untergebracht sind, angewandt. Strukturkonflikte beim gleichzeitigen Speicherzugriff der Lade-/Speicher- und der Befehlsholeeinheit können damit für die Primär-Cache-Speicher vermieden werden. Die nächste Ebene der Speicherhierarchie eines Universalmikroprozessors, der Sekundär-Cache-Speicher (*L2-Cache*), vereint üblicherweise dann wieder Code und Daten in einem Cache-Speicher.

Die Organisation eines Code-Cache-Speichers ist einfacher als diejenige 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 Cache-Kohärenz beachtet werden muss. Selbstmodifizierender Code ist deshalb auch auf heutigen Superskalarprozessoren nicht effizient implementierbar.

Üblicherweise hat ein Code-Cache-Speicher eine Kapazität von 8 – 64 kB. Die Cache-Blöcke, die zwischen Code-Cache und Sekundär-Cache oder Hauptspeicher ausgetauscht werden, sind meist 32 Bytes groß und umfassen damit acht 32-Bit-Befehle.

Meist wird pro Takt ein Befehlsholeblock (*Fetch Block*), abkürzend als Befehlsblock bezeichnet, von der Länge des Cache-Blocks aus dem Code-Cache-Speicher in die Pipeline geladen. Falls das Befehlsformat variabel lange Befehle zulässt – wie es beim IA-32-Format (*Intel Architecture*) der Fall ist –, enthält ein Befehlsblock eine variable Anzahl von Befehlen und der Anfang eines jeden Befehls muss erst festgestellt werden. Das bedingt nicht nur eine komplexere Decodierstufe (mehrere, unterschiedlich lange Befehle müssen pro Takt decodiert werden), sondern auch eine komplexere Befehlsholestufe.

Um die nachfolgenden Pipeline-Stufen zu vereinfachen, werden daher bei manchen Mikroprozessoren die Befehle beim Übertragen aus dem Speicher oder den Sekundär-Cache-Speicher in den Code-Cache-Speicher schon vorab decodiert. Insbesondere wird dabei der Anfang eines jeden Befehls im Befehlsstrom ermittelt und markiert.

2.2.1.2 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.

Simulationen gezeigt, dass aus diesen Gründen ein achtfach superskalarer Prozessor mit einer einfachen Befehlsholestufe im Durchschnitt weniger als vier Befehle pro Takt den nachfolgenden Pipeline-Stufen zur Verfügung stellen kann.

Eine Möglichkeit, um 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.

Alle diese Techniken können mit einem Vorabladen der Befehle aus dem Cache-Speicher kombiniert werden. Das Vorabladen der Befehle in einen Befehlspuffer zur Decodierstufe entkoppelt die Befehlsholestufe von der Decodierstufe.

Schwankungen in der Anzahl der bereitgestellten Befehle können 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. Eine solche Befehlsbereitstellung erscheint auch notwendig zu sein, um Techniken wie die spekulative beidseitige Ausführung nach einem bedingten Sprungbefehl zu unterstützen.

Die Lösungen dafür können verschränkte Code-Cache-Speicher oder Mehrkanal-Cache-Speicher, kombiniert mit mehreren Befehlsholeeinheiten sein. Eine andere Lösung besteht in der Ergänzung des Code-Cache-Speichers durch einen so genannten Trace-Cache-Speicher, der im nächsten Abschnitt behandelt wird.

2.2.1.3 Trace Cache

Eine hohe Zuordnungs- und Ausführungsbandbreite zukünftiger Prozessoren erfordert eine entsprechend effiziente Befehlsbereitstellung. Ein Problem entsteht durch die vielen Sprungbefehle in einem konventionellen sequenziellen Befehlsstrom[5]. Bei einer Zuordnungsbandbreite von acht oder höher sind somit 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 *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 Befehlsablaufs starten und sich über mehrere Grundblöcke, d.h. über mehrere Sprungbefehle hinweg, erstrecken kann. Die Anzahl der Befehle eines solchen Traces wird durch die maximale Länge eines Trace-Cache-Blocks beschränkt.

Wird die entsprechende Befehlsfolge erneut ausgeführt, so werden die Befehle nicht mehr dem Code Cache, sondern dem Trace Cache entnommen. Da die dynamisch erzeugte Befehlsfolge im Trace Cache bereits Befehle auf einem spekulativen Pfad enthält, ermöglicht es der Trace Cache, den Ausführungsteil der Prozessor-Pipeline mit einem fortlaufenden Befehlsstrom zu versorgen. Ein Nachladen an mehreren Stellen, wie oben beschrieben, ist nicht mehr nötig.

Ein Trace Cache enthält die dynamischen Befehlsablauffolgen, während der Code-Cache-Speicher die statischen, also vom Compiler erzeugten, Befehlsfolgen speichert. Abbildung 2.5 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. Ein gesamter Trace-Cache-Block kann in einem Takt in den Befehlspuffer der Befehlsholeeinheit übertragen werden.

Auf den Trace Cache kann mit dem Befehlszählerinhalt oder mit der als

[5] In typischen Programmen verändert ca. jeder 4. oder 5. Befehl den Programmfluss, jeder 5. bis 7. ist ein *bedingter* Sprungbefehl, s.u.

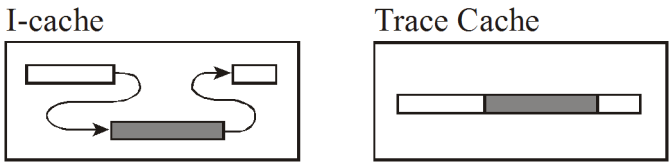


Abbildung 2.5: Code-Cache-Speicher (*I-Cache*) und Trace Cache.

nächstes zu holenden Befehlsadresse des zuvor geladenen Trace-Cache-Blocks zugegriffen werden.

Der Trace Cache wird während der Programmausführung von der Befehlsholeeinheit mit den aus dem Code-Cache-Speicher geholten Befehlsfolgen unter Berücksichtigung von Sprungspekulationen bis zum Maximum eines Trace-Cache-Blocks gefüllt. Dieses Auffüllen kann unabhängig und zeitgleich zu der normalen Pipeline-Verarbeitung geschehen. Dadurch ist die Trace-Cache-Füllung nicht auf einem kritischen Pfad in der Pipeline. Dieser wird durch einen Trace Cache auch nicht verlängert. Um ein korrektes Rücksetzen nach einer Sprungfehlspekulation und um präzise Unterbrechungen zu gewährleisten, müssen geeignete Zusatzinformationen mit jedem Befehl im Trace Cache gespeichert werden.

Ein Trace im Trace Cache kann sogar bereits Befehle aus mehreren Sprungvorhersagen umfassen. Konsequenterweise kann damit die Sprungspekulation durch eine *Next-Trace*-Spekulation, die zwischen verschiedenen Trace-Cache-Blöcken des Trace Cache auswählt, ergänzt werden.

Eine erste Implementierung eines Trace Cache fand sich in Intels Pentium-4-Prozessor, der sogar auf den Code-Cache verzichtete und im Falle eines Trace-Cache-Fehlzugriffs die Befehle direkt aus dem Sekundär-Cache ludt.

2.2.2 Sprungvorhersage und spekulative Ausführung

2.2.2.1 Grundlagen

Für heutige und zukünftige Mikroprozessoren ist eine effiziente Behandlung von Sprungbefehlen sehr wichtig. In den einzelnen Stufen eines Superskalarprozessors befinden sich viele Befehle in verschiedenen Ausführungszuständen. Der Ausführungsteil eines Prozessors arbeitet am besten, wenn der Zuordnungseinheit sehr viele Befehle zur Auswahl stehen, d.h. das Befehlsfenster groß ist. Betrachtet man nun den Befehlsstrom eines typischen Anwenderprogramms, so stellt man fest, dass etwa jeder fünfte bis siebte Befehl ein *bedingter* Sprungbefehl ist, der den kontinuierlichen Befehlsfluss durch die Pipeline *möglicherweise* unterbrechen kann. Unter Berücksichtigung der spekulativen Ausführung von Befehlen befinden sich bei heutigen Superskalarprozessoren in der Regel sogar mehrere Sprungbefehle in der Pipeline.

Um eine hohe Leistung bei der Sprungbehandlung zu erzielen, sind folgende Bedingungen 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, d.h. falls die Sprungvorhersage (*Branch Prediction*) einen genommenen Sprung unterstellt, sofort in den Befehlszähler geholt werden, sodass ohne Verzögerung die Befehle ab der Sprungzieladresse spekulativ in die Pipeline geladen werden können.

- Die Sprungvorhersage sollte sich durch eine sehr hohe Genauigkeit und die Möglichkeit, Befehle spekulativ auszuführen, auszeichnen.

- Häufig muss ein weiterer Sprung vorhergesagt werden, obwohl der Test der Bedingung des vorhergehenden Sprungs noch nicht ausgeführt wurde. Der Prozessor muss daher mehrere Ebenen der Spekulation verwalten können.

- Wenn ein Sprung falsch vorhergesagt wurde, ist außerdem ein schneller Rückrollmechanismus mit geringem Fehlspekulationsaufwand (*Misprediction Penalty*) wichtig.

Eine möglichst frühe Bestimmung des Sprungausgangs wird durch die direkte Auswertung (*Forwarding*) des Ergebnisses vom vorangegangenen Vergleichsbefehl erzielt, ohne das dieses Ergebnis zunächst im Registersatz abgelegt werden muss. Der Test der Sprungbedingung kann dadurch bis in die Decodierstufe vorgezogen werden.

Die Gesamtleistung 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 1.5.7 beschriebenen statischen Sprungvorhersagetechniken haben sich bei heutigen Superskalarprozessoren die wesentlich genaueren dynamischen Sprungvorhersagetechniken durchgesetzt. Der Einsatz größerer Informationstabellen über die bisherigen Sprungverläufe führt bei diesen Techniken auch zu weniger Fehlspekulationen. Mit den „bisherigen Sprungverläufen“, der sog. „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 vielstufe Pipeline verursacht mehr Kosten als eine kurze Pipeline.

Bei manchen Prozessoren ist es nicht möglich, Befehle aus internen Puffern zu löschen, d.h., auch wenn Befehle als ungültig erkannt werden, müssen diese ausgeführt und dann von der Rückordnungsstufe verworfen werden. Weiterhin gibt es noch dynamische Einflüsse auf die Kosten für das Rückrollen, wie z.B. die Anzahl der spekulativen Befehle im Befehlsfenster oder dem Rückordnungspuffer (*Reorder Buffer*). Typischerweise können aus diesen meist recht großen Puffern immer nur eine kleine Anzahl Befehle pro Takt gelöscht werden. Dies führt dazu, dass die Kosten im Allgemeinen hoch sind, wie z.B. 11 oder mehr Takte beim Pentium II. Für eine gute Gesamtleistung bei der Sprungvorhersage ist es daher sehr wichtig, dass die Genauigkeit der Vorhersage hoch ist.

Eine andere Technik, mit Sprüngen umzugehen, ist die Prädikation (*Predication*),die so genannte 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 Anwendung der Prädikation. Bei heutigen Superskalarprozessoren ist diese Technik jedoch nicht einsetzbar, da jeder Befehl mindestens ein Prädikationsbit enthalten muss und dies nur in wenigen Fällen gegeben ist. Kompatibilitätsgründe und die bereits voll ausgeschöpften 32-Bit-Befehlsformate stehen einer solchen Erweiterung entgegen. Voll prädikative Befehlssätze sind jedoch der Intel IA-64-Befehlssatz des Itanium-Prozessors und die Befehlssätze einiger Digitaler Signalprozessoren.

2.2.2.2 Dynamische Sprungvorhersagetechniken

dynamische Sprungvorhersagetechnik

Bei der dynamischen Sprungvorhersagetechnik wird die Entscheidung über die Spekulationsrichtung eines Sprungs in Abhängigkeit vom bisherigen Programmablauf getroffen. Die Sprungverläufe werden beim Programmablauf in Sprungverlaufstabellen gesammelt und auf der Grundlage der Tabelleneinträge werden aktuelle Sprungvorhersagen getroffen. Bei Fehlspekulationen werden die Tabellen per Hardware abgeändert.

Im Gegensatz dazu steht die Sprungvorhersage bei den statischen Sprungvorhersagetechniken aus Abschnitt 1.5.7 immer fest und kann sich im Verlauf einer Programmausführung nie ändern. Auch wenn ein Sprung ständig falsch vorhergesagt wird, kann die Hardware darauf nicht reagieren, im Gegensatz zu einer dynamischen Sprungvorhersage.

Natürlich wird wie bei allen Prädiktoren zusätzlich die Sprungzieladresse benötigt, was einen Sprungzieladress-Cache (s. Abschnitt 1.4.6) notwendig macht, der die Sprungadressen und die Sprungzieladressen speichert. Eine mögliche Implementierungstechnik ist deshalb die Erweiterung des Sprungzieladress-Caches um die zusätzlichen Bits der Sprungverlaufstabelle (vgl. die Vorhersagebits in Abbildung 1.32 in Abschnitt 1.5.6).

Im Normalfall bleiben der Sprungzieladress-Cache und die Sprungverlaufstabelle getrennte Hardware-Einrichtungen. Man beachte dabei die unterschiedliche Organisation: der Sprungzieladress-Cache ist ein meist vollassoziativ organisierter Cache-Speicher (s. Kapitel 3), der die gesamte Sprungbefehlsadresse zur Identifikation eines Eintrags speichert. Eine Sprungverlaufstabelle wird über einen Teil der Sprungbefehlsadresse adressiert und enthält dann nur das oder die Vorhersagebit(s) als Einträge. Üblicherweise werden die niederwertigen Bits der Befehlsadresse genommen, um einen Eintrag zu adressieren.

Gründe für Fehlspekultationen

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. Zu Beginn ist eine statische Vorhersage oft besser. Im Allgemeinen sind jedoch nach der Warmlaufphase die dynamischen Techniken den statischen überlegen, wobei dies mit erhöhter Hardwarekomplexität erkauft wird.

- Die Spekulation für den 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.

- Durch die Indizierung der im Speicherplatz beschränkten Sprungverlaufstabelle wird die Verlaufsgeschichte eines *anderen* Sprungbefehls miterfasst. Der Index zur Adressierung eines Tabelleneintrags besteht aus einem Teil der Sprungbefehlsadresse, und wenn zwei Sprungbefehle in dem betreffenden Adressteil dasselbe Bitmuster aufweisen, werden sie bei den einfachen Verfahren auf den gleichen Eintrag abgebildet. Eine solche **Wechselwirkung** oder **Interferenz** (*Branch Interference*, *Aliasing*) zwischen zwei Sprungbefehlen ist unerwünscht. Sie führt besonders bei kleinen Sprungverlaufstabellen häufig zu einer Fehlvorhersage. Die Anzahl der Interferenzen lässt sich verringern und damit die Spekulationsgenauigkeit signifikant verbessern, wenn die Sprungverlaufstabelle vergrößert wird. Die Interferenzen würden sogar ganz vermieden, wenn jeder Sprungbefehl einen eigenen Eintrag in der Sprungverlaufstabelle erhält, dies ist jedoch für beliebig große Programme wegen des beschränkten Platzes auf dem Prozessor-Chip nicht möglich.

Wichtig: Wechselwirkung oder Interferenz zwischen zwei Sprungbefehlen

Als dynamische Sprungvorhersagetechniken kommen heute neben dem einfachen Zwei-Bit-Prädiktor zunehmend Korrelations- und Hybrid-Prädiktoren zum Einsatz. Diese Techniken werden in den nachfolgenden Unterabschnitten beschrieben.

2.2.2.3 Ein- und Zwei-Bit-Prädiktoren

Die einfachste dynamische Sprungvorhersagetechnik ist der **Ein-Bit-Prädiktor**, der für jeden Sprungbefehl die zwei Zustände „genommen“ oder „nicht genommen“ in einem Bit speichert. Diese Zustände beziehen sich dabei immer auf die zeitlich letzte Ausführung des Sprungbefehls. Die Zustände eines Ein-Bit-Prädiktors sind in Abbildung 2.6 dargestellt. Dabei steht T für „genommen“ (*taken*) und NT für „nicht genommen“ (*not taken*).

Ein-Bit-Prädiktor

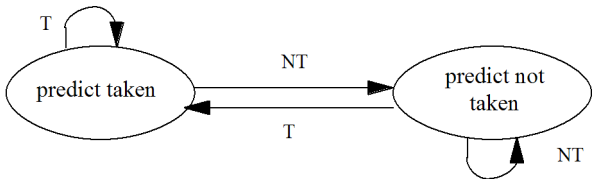


Abbildung 2.6: Die Zustände des Ein-Bit-Prädiktors.

Der Ein-Bit-Prädiktor wird in der Befehlsholestufe implementiert und benötigt eine Sprungverlaufstabelle (*Branch History Table* – BHT, auch *Branch*

Prediction Buffer genannt), 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, wenn es gelöscht ist, als „nicht genommen“. Im Falle einer Fehlspekulation wird das Bit invertiert und damit die Richtung der Vorhersage umgekehrt.

Eine Implementierungsvariante des Ein-Bit-Prädiktors besteht darin, nur die als „genommen“ vorhergesagten Sprünge in den Sprungzieladress-Cache einzutragen und auf die Vorhersagebits zu verzichten.

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-Prädiktor

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“ (*strongly taken*), „vielleicht genommen“ (*weakly taken*), „vielleicht nicht genommen“ (*weakly not taken*) und „sicher nicht genommen“ (*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 2.7 dargestellt. Die zweite, **Hysteresezähler** genannte Variante verdeutlicht Abbildung 2.8.

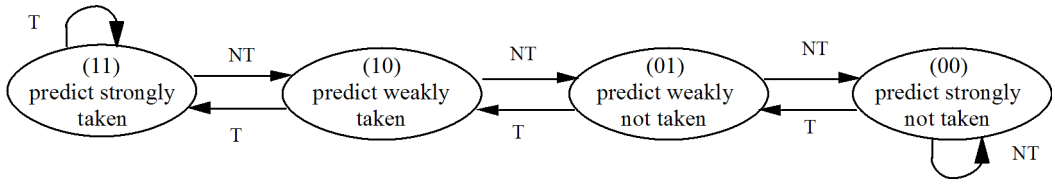


Abbildung 2.7: Die Zustände des Zwei-Bit-Prädiktors mit Sättigungszähler

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.

Die zweite Variante, die Hysteresemethode, unterscheidet sich von der des Sättigungszählers dadurch, dass direkt von einem „unsicheren“ (*weakly*) Zu-

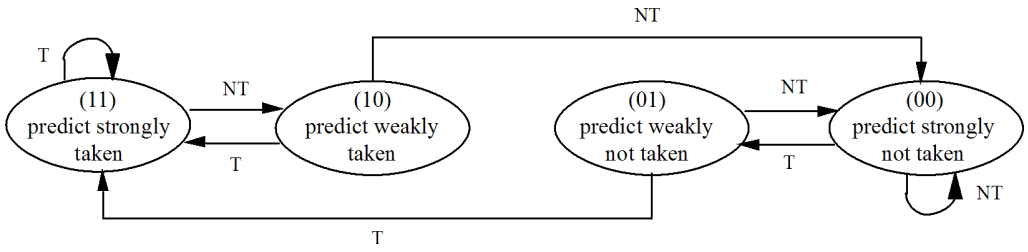


Abbildung 2.8: Die Zustände des Zwei-Bit-Prädiktors mit Hysteresezähler.

stand 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.[6] Der Zustand von noch nicht aufgetretenen Sprüngen wird dabei auf „vielleicht nicht genommen“ initialisiert.

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 Sprungzieladress-Cache implementiert werden, wobei jeder Eintrag um zwei Vorhersagebits erweitert wird. Eine weitere Möglichkeit besteht darin, den Sprungzieladress-Cache für die Zieladressen und eine separate Sprungverlaufstabelle für die Vorhersage zu nutzen.

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, wenn 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 in C-Code geschriebenen Programmstücks voneinander abhängig (*correlated*):

```
if (d == 0) // Sprung s1
    d = 1;
if (d == 1) // Sprung s2 ...
```

Wie man sieht, wird der zweite Sprung immer dann genommen, wenn der erste genommen wurde. Das ist eine Korrelationsinformation, die von Ein- und Zwei-Bit-Prädiktoren nicht genutzt werden kann. Bei diesem Programmstück kann es passieren, dass von einem Ein-Bit- oder Zwei-Bit-Prädiktor jeder Sprung falsch vorhergesagt wird. Dazu betrachten wir eine mögliche Übersetzung der if-then-Konstrukte in Maschinensprache (die Variable d ist im Register R1 abgelegt):

```
    bnez R1, L1          ; Sprung s1 falls (d != 0)
    addi R1, R0, #1      ; d = 1
L1: sub R3, R1, #1
    bnez R3, L2          ; Sprung s2 falls (d != 1)
    ...
L2: ...
```

[6] Diese Technik wurde im UltraSPARC-I-Prozessor implementiert.

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. Tabelle 2.1 verdeutlicht diesen Sachverhalt.

Tabelle 2.1: Verkürzter Programmablauf für obiges Beispiel.

Anfangswert für d bei Iterationsbeginn	d=0 ?	Sprung s1	d vor s2	d=1 ?	Sprung 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 2.5 mit Anfangszustand „vielleicht genommen“. Der Zwei-Bit-Prädiktor mit Hysteresezähler aus Abbildung 2.6 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.

Ausblick

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 *Korrelationsprädiktoren*, die neben der eigenen Vergangenheit eines Sprungbefehls auch die Historie benachbarter, im Programmlauf vorhergegangener Sprünge berücksichtigen (vgl. den folgenden Unterabschnitt), und die *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. Auf die zuletzt genannten Verfahren können wir aus Platzgründen hier nicht eingehen. Näheres findet man in [13] und [17].

Selbsttestaufgabe 2.2 (Vor-/Rückwärtszähler)

Man betrachte das obige Programmbeispiel mit dem in Tabelle 2.1 gegebenen Programmablauf. Vervollständigen Sie 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“) und

Sprung	*Initialzustand*	*d=0*	*d=2*	*d=0*
s1	*WT*			
s2	*WT*			

Lösung auf Seite 121

Selbsttestaufgabe 2.3 (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 ; (1) R3 <= R0 ? Status in R10
        BNEZ R10, Ende1 ; (2) wenn ja, dann gehe zu Ende1
Loop2:  SLE R11, R4, R0 ; (3) R4 <= R0 ? Status in R11
        BNEZ R11,Ende2  ; (4) wenn ja, dann gehe zu Ende2
    .
    .
    .
        SUB R4, R4, R1  ; (19) R4 = R4 -- R1
        J Loop2         ; (20) Goto Loop2
Ende2:  SUB R3, R3, R1  ; (21) R3 = R3 -- R1
        J Loop1         ; (22) Goto Loop1
Ende1:  ...             ; (23)
```

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.

Lösung auf Seite 122

2.2.2.4 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.

Korrelationsprädiktor

Die **Korrelationsprädiktoren** (*Correlation-based Predictors* oder *Correlating 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 (entsprechend Abschnitt 2.2.2.3) 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 Sprungverlaufstabelle darstellt, die für partiell verschiedene Sprungbefehlsadressen die Zustände eines n-Bit-Prädiktors speichert. Der PHT-Eintrag selbst wird wie beim Zwei-Bit-Prädiktor über eine bestimmte Zahl k der niederwertigen Bits der Sprungbefehlsadresse selektiert. Man beachte, dass Sprungbefehlsadressen mit gleichen niederwertigen aber verschiedenen höherwertigen Adressbits auf die gleiche Speicherstelle abgebildet werden. Aufgrund der Lokalitätseigenschaften geht man davon aus, dass eine Unterscheidung in einem Bereich von der 2^k Sprungbefehlsadressen ausreichend ist.

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 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.

Abbildung 2.9 zeigt die Implementierung eines (2,2)-Prädiktors mit vier PHTs mit jeweils 1024 Einträgen (1 k Einträge). Über das BHR wird eine spezielle PHT ausgewählt (*select*). Die Einträge in der PHT werden im Allgemeinen über die niederwertigen Bits der Sprungadresse ausgewählt (hier $k = 10$ Bits zur Adressierung von 1024 Einträgen).

Man kann die vier PHTs der Größe 1 k Einträge auch als eine einzige große PHT mit 4 k Einträgen betrachten. Um einen Eintrag zu finden benötigt man

damit zwölf Bits, was der Konkatenation des BHR-Inhalts mit dem Adressteil des Sprungbefehls entspricht.

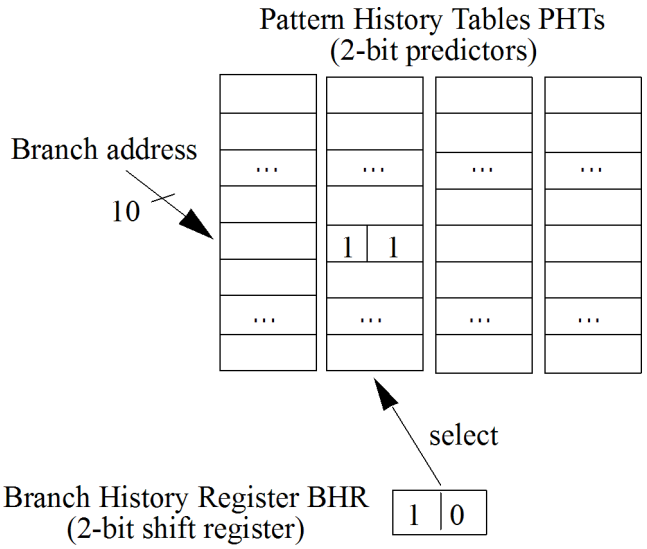


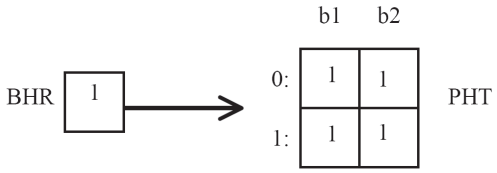
Abbildung 2.9: Implementierung eines (2,2)-Prädiktors.

Selbsttestaufgabe 2.4 (Vor-/Rückwärtszähler)

Man betrachte erneut das Programmbeispiel aus Selbsttestaufgabe 2.2 mit dem in Tabelle 2.1 gegebenen Programmablauf. Im Text heißt es: „Ein (1,1)-Korrelationsprädiktor kann den Zusammenhang zwischen beiden Sprüngen erkennen und nutzen. Er trifft nur in der ersten Iteration eine falsche Vorhersage.“

Vervollständigen Sie die Zustände der Prädiktortabelleneinträge für die beiden Sprünge bei Verwendung eines (1,1)-Korrelationsprädiktors (Initialzustand: „predict taken“) und begründen Sie damit die obige Aussage.

Initialzustand:



Sprung	*Initialzustand*	*d=0*	*d=2*
s1	*T*		
s2	*T*		

Lösung auf Seite 124

2.2.2.5 Prädikation

Prädikation als Alternative zur *Prädiktion*

VLIW-Prozessoren verwenden häufig statt einer Sprungvorhersage mit spekulativer Ausführung die so genannte **Prädikation** (*Predication*), die es ermöglicht, statt einer Sprungspekulation die Befehle beider Sprungrichtungen auszuführen und im Nachhinein nur die fälschlicherweise ausgeführten Befehle zu verwerfen.

Um Prädikation anwenden zu können, sind zwei Architekturerweiterungen notwendig. Es müssen ein oder mehrere ein Bit breite Prädikatsregister vorhanden sein und möglichst alle Befehle des Befehlssatzes müssen diese Prädikatsregister im Befehlsformat ansprechen können. Ein Prädikatsregister wird von einem Vergleichsbefehl gesetzt. Im einfachsten Fall genügt ein einzelnes Ein-Bit-Prädikatsregister und ein zusätzliches Bit im Befehlsformat, das es dem Compiler erlaubt, die gültige Befehlsausführung von der Übereinstimmung des Prädikatsbits im Befehl mit dem Bit im Prädikationsregister abhängig zu machen.

Falls alle Befehle eines Befehlssatzes prädikativ sind, so spricht man von einem **voll prädikativen Befehlssatz**.

Das folgenden Programmstück in C wird zur Demonstration einer Implementierung mit Prädikation verwendet:

```
if (x==0)   {// branch b1
    a=b+c;
    d=e-f;  }
g=h*i;       // instruction independent of branch b1 ...
```

Falls das Programmstück in Maschinencode mit einem bedingten Sprungbefehl übersetzt wird, so erfolgt zur Laufzeit eine Sprungspekulation. Im Falle einer Fehlspekulation werden nicht nur die spekulativ ausgeführten Befehle `a=b+c` und `d=e-f`, sondern *alle* spekulativen Ausführungsresultate nachfolgender Befehle gelöscht. Das gilt auch für den von dem Sprungbefehl und seinen beiden Nachfolgebefehlen unabhängigen Befehl `g=h*i`.

Das lässt sich jedoch vermeiden, wenn der Quellcode in eine Codefolge mit prädikativen Befehlen übersetzt wird (jede Zeile steht für einen einzelnen Maschinenbefehl in Pseudo-C-Code):

```
(Pred = (x==0))       // replaces branch b1:
                      // Pred is set to true if x==0
if Pred then a=b+c;   // The operations are only performed
if Pred then d=e-f;   // if Pred is set to true
g=h*i;
```

Die Anweisung `if Pred then a=b+c` steht für einen prädikativen Maschinenbefehl. Im Beispiel werden der zweite und der dritte Befehl nur dann ausgeführt (bzw. die Resultate gültig gemacht), wenn das Prädikat Pred zu *true* ausgewertet wurde. Falls Pred zu *false* ausgewertet wird, so werden nur die betreffenden beiden Befehlsausführungen gelöscht und alle Ausführungen nachfolgender Befehle werden weiterverwendet. Damit sind zwei Befehle überflüssigerweise ausgeführt worden, deren Ausführung im Falle einer Implementierung mit einem bedingten Sprungbefehl und korrekter Spekulation nicht erfolgt wäre.

Wie man sehen kann, ermöglicht es die Prädikation, bedingte Sprungbefehle aus dem Maschinencode zu eliminieren. Eine Fehlspekulation ist nicht mehr möglich, die Kosten für das Rückrollen und Neuaufsetzen der Befehlsausführung werden eingespart. Diese sind meist höher als die unnötige Ausführung einer der beiden Sprungpfade. Weiterhin wird die Grundblocklänge (die Anzahl der Befehle zwischen zwei Sprungbefehlen) vergrößert und damit dem Compiler mehr Spielraum für eine Optimierung der Befehlsanordnung gegeben.

Prädikation lässt sich besonders gut für if-then-Anweisungen mit kleinem then-Teil einsetzen, wie es in der obigen Codefolge der Fall ist. Sie ist besonders effizient, wenn der Sprungausgang irregulär wechselt und damit nicht vorhersagbar ist. Bei sehr gut vorhersagbaren Sprüngen führt die Prädikation ständig zu unnötigen Befehlsausführungen und belastet die Prozessorressourcen.

Prädikation kann außerdem nicht immer sinnvoll angewandt werden. Falls sehr lange Befehlsfolgen von einer Verzweigung abhängen, so müssten eventuell sehr viele Befehle prädikativ ausgeführt werden. Eine Sprungspekulation ist dann meist effizienter.

Prädikation beeinflusst nicht nur die Architektur durch zusätzliche Befehle im Befehlssatz und zusätzlich nötige Bits im Befehlsformat, sondern führt auch zu einer komplexeren Mikroarchitektur des Prozessors.

Prädikative Befehle durchlaufen wie die nicht-prädikativen Befehle die Befehlsbereitstellung und -decodierung und werden wie diese im Befehlsfenster zwischengespeichert. Nun hängt es vom Prozessor ab, wie weit die prädikativen Befehle in der Prozessor-Pipeline voranschreiten können, bevor die Prädikation entschieden ist.

Falls das Prädikationsregister als zusätzliches Operandenregister des prädikativen Befehls betrachtet wird, so verbleibt der Befehl im Befehlsfenster, bis alle Operanden (also auch der Wert des Prädikatsregisters) vorhanden sind. In diesem Fall erreicht ein prädikativer Befehl erst dann die Ausführungsstufe, wenn die Prädikation entschieden ist, also der vorangehende Vergleichsbefehl das Prädikatsregister gesetzt hat. Anschließend wird der Befehl nur dann weiter ausgeführt, wenn das Prädikat zu *true* ausgewertet wurde.

Eine alternative Implementierung führt den prädikativen Befehl spekulativ aus und macht das Resultat nur dann gültig, wenn das Prädikat zu *true* ausgewertet wurde (z.B. bei der Intel IA-64-Architektur).

Bei heutigen superskalaren Prozessoren wird die Prädikation noch selten eingesetzt, jedoch findet sich die Prädikation bei den Digitalen Signalprozessoren und bei leistungsstarken Mikroprozessoren und Mikrocontrollern, die im Bereich der eingebetteten Systeme eingesetzt werden. Der Befehlssatz der ARM-Prozessoren ist voll prädikativ. Weiterhin sind im IA-64-Befehlssatz alle Befehle prädikativ, wobei der Itanium-Prozessor Prädikation und Sprungspekulation implementiert. Die MIPS-, PowerPC- und SPARC-Befehlssätze enthalten nur prädikative Ladebefehle.

2.2.2.6 Mehrpfadausführung

Bei der Mehrpfadausführung (*Multipath*) oder *Eager Execution* werden nach einem bedingten Sprung beide möglichen Ausführungspfade in die Pipeline ge-

laden und ausgeführt. Eventuelle Datenabhängigkeiten zwischen den beiden Ausführungspfaden (insbesondere das Schreiben auf das gleiche Register) können durch die Technik der Registerumbenennung (*Register Renaming*, s.u.) gelöst werden. Sobald die Sprungrichtung entschieden ist, werden alle Befehlsresultate und noch in der Pipeline befindlichen Befehle des nicht genommenen Ausführungspfads wieder gelöscht. Die Mehrpfadausführung vermeidet Fehlspekulationen und erhöht die Ausführungsgeschwindigkeit, da der Fehlspekulationsaufwand entfällt. Die Mehrpfadausführung ist eine Mikroarchitekturtechnik, die ohne Einfluss auf die Architektur implementiert werden kann.

Das Problem beim Einsatz einer Mehrpfadausführung ist der dafür nötige Ressourcenverbrauch. Wenn man eine Mehrpfadausführung mit unbegrenzten Prozessorressourcen annimmt, so erhält man denselben (theoretischen) Geschwindigkeitsgewinn für die Programmausführung wie die Annahme einer perfekten Sprungvorhersage, bei der kein einziger Sprung falsch vorhergesagt würde. Bei den in der Realität begrenzten Ressourcen muss bei der Mehrpfadausführung jedoch sehr sorgfältig ausgewählt werden, wann diese eingesetzt wird. Es werden deshalb Mechanismen benötigt, um auszuwählen, wann die Mehrpfadausführung und wann eine Sprungspekulation angewandt werden sollte.

Ein Entscheidungsmechanismus könnte die Anwendung einer Zuverlässigkeitsschätzung der Sprungvorhersage sein, die immer dann, wenn für die Vorhersage eines Sprunges eine hohe Zuverlässigkeit ermittelt wird, eine Sprungspekulation durchführt und bei niedriger Zuverlässigkeit eine Mehrpfadausführung vorschlägt.

Bisher ist die Mehrpfadausführung noch selten implementiert worden und auch dann immer nur in einem sehr begrenzten Umfang. Beispielsweise können beim SuperSPARC-Prozessor sowie beim IBM 360/91 und nachfolgenden IBM-Großrechnern einige Befehle beider Ausführungspfade in die jeweiligen Befehlspuffer geladen werden.

2.2.3 Decodierung und Registerumbenennung

In einem Superskalar- oder VLIW-Prozessor werden in der Decodierstufe mehrere Befehle pro Takt decodiert und die in den Befehlen angegebenen Architekturregister[7] „umbenannt“. Das heißt, in Abhängigkeit von der Mikroarchitektur werden die Architekturregister auf die physikalisch vorhandenen Register abgebildet bzw. den Architekturregistern werden Umbenennungspufferregister zugeordnet. Die Befehlsdecodierung kann sich über eine oder mehrere Pipeline-Stufen erstrecken. Die Registerumbenennung kann in einer eigenen Pipeline-Stufe erfolgen oder in die Decodierstufe integriert sein.

2.2.3.1 Decodierung

Um eine hohe Verarbeitungsleistung zu erzielen, muss der Prozessor mindestens so viele Befehle bereitstellen und decodieren, wie die Zuordnungsbandbreite beträgt. Falls das Befehlsfenster immer voll gehalten werden kann, ist eine tiefere Befehlsvorschau (*Instruction Lookahead*) möglich, die es wiederum erlaubt,

[7] das sind die in einem Befehl stehenden Operanden- und Resultatregister

mehr Befehle zu finden, die pro Takt den Ausführungseinheiten zugeordnet werden können. Außerdem holt und decodiert ein Superskalarprozessor bereits heute etwa doppelt so viele Befehle, wie letztendlich gültig gemacht werden, da viele spekulativ ausgeführte Befehle nach einer Fehlspekulation wieder gelöscht werden müssen.

Üblicherweise ist die Decodierbandbreite dieselbe wie die Bandbreite der Befehlsbereitstellung. Das Bereitstellen und Decodieren mehrerer Befehle pro Takt wird durch ein Befehlsformat fester Länge erleichtert.

Falls die Befehlslängen variabel sind, was bei CISC-Architekturen wie der Intel IA-32-Architektur der Fall ist, wird eine mehrstufige Decodierung angewandt. Die erste Decodierstufe bestimmt die Grenzen der Befehle im Befehlsstrom und liefert eine Anzahl von Befehlen an die zweite Stufe. Diese decodiert die Befehle und erzeugt aus jedem Befehl einen oder mehrere „Mikrobefehle“ (μOps). Das ermöglicht es, komplexe CISC-Befehle in einfachere Befehle aufzuspalten, die den RISC-Befehlen vergleichbar sind.

Code Morphing-Technik

Man bezeichnet solche Prozessoren häufig als CISC-Prozessoren mit einem RISC-Kern (in der Mikroarchitektur) und sieht darin die Lösung des Kompatibilitätsproblems. Die Befehle älterer, oft verwendeter, aber nicht pipelinegerechter Befehlssätze werden während der Decodierung durch mehrere einfache, in der Pipeline leicht zu verarbeitende RISC-artige Mikrobefehle ersetzt. Das Verfahren wird bei den PC-Prozessoren der Intel Pentium-Familie und der AMD Athlon-Familie für den IA-32-Befehlssatz angewandt, um die heute den Markt dominierenden PCs mit ihren Vorläufern kompatibel zu halten.

Der Vorteil der CISC-Befehle gegenüber den RISC-Befehlen ist ihre höhere Codedichte. Dieser Vorteil wird durch den höheren Decodieraufwand erkauft.

Falls eine Teildecodierung bereits beim Übertrag vom Sekundär- in den Code-Cache-Speicher oder während der Befehlsbereitstellung gemacht wird, so kann die Decodierstufe vereinfacht werden. Der MIPS R10000-Prozessor erweitert z.B. jeden Befehl beim Übertrag in den Code-Cache-Speicher von 32 auf 36 Bits. Die vier Extrabits bezeichnen, welcher Art von Ausführungseinheit der Befehl zugeordnet werden muss. Die Vorabdecodierung ordnet auch bereits Operanden- und Resultatbezeichner um, damit diese Felder bei jedem Befehl in derselben Position des Befehlsformats sind, und modifiziert die Opcodes, um die Decodierung zu vereinfachen.

2.2.3.2 Registerumbenennung

Die Registerumbenennung beseitigt scheinbare Datenabhängigkeiten (*Name Dependencies*) zwischen Registeroperanden. Scheinbare Datenabhängigkeiten sind gemäß Abschnitt 1.4.4 die Gegenabhängigkeiten (*Anti Dependencies*), d.h. das Lesen eines Registerwertes, bevor dieser von einem in Programmordnung nachfolgenden Befehl überschrieben wird, und die Ausgabeabhängigkeiten (*Output Dependencies*), also das Rückschreiben eines Registerwertes, bevor dieser von einem in Programmordnung nachfolgenden Befehl erneut überschrieben wird. Gegenabhängigkeiten könnten in einem Superskalarprozessor mit Befehlszuweisung außerhalb der Programmordnung zu falschen Ergebnissen führen, wenn ein nachfolgender Befehl außerhalb der Programmordnung

ausgeführt und beendet würde, bevor ein vorheriger Befehl seine Operanden gelesen hat. Ohne Registerumbenennung könnten Ausgabeabhängigkeiten bei jedem Prozessor, der eine Beendigung der Befehlsausführung außerhalb der Programmordnung zulässt, das Ergebnis verfälschen.

Die Registerumbenennung kann auf statische Weise (durch den Compiler) oder auf dynamische Weise (per Hardware) erfolgen. Bei der dynamischen Registerumbenennung wird jedem im Befehl spezifizierten Zielregister ein noch nicht belegtes physikalisches Register zugeordnet. Falls das Zielregister bereits in einem vorhergehenden Befehl als Zielregister vorliegt, besteht eine Ausgabeabhängigkeit, falls es dort als Quellregister auftritt, hingegen eine Gegenabhängigkeit. In diesen Fällen wird das Register auf ein anderes physikalisches Register abgebildet, wodurch Ausgabe- und Gegenabhängigkeiten zwischen Registeroperanden automatisch beseitigt werden. Nachfolgende Befehle, die auf dasselbe Architekturregister als Operandenregister zugreifen, erhalten bei der Registerumbenennung das zuletzt zugeordnete physikalische Register als Eingabeoperand.

Jedes physikalische Register wird nach seiner Zuordnung nur einmal beschrieben, da nachfolgende Schreibzugriffe auf dasselbe Architekturregister auf andere physikalische Register erfolgen. Falls ein anderer Befehl den Registerwert benötigt, also eine echte Datenabhängigkeit vorliegt, so muss dieser datenabhängige Befehl warten, bis der Registerwert vorhanden ist. Datenabhängigkeiten zwischen Registeroperanden können nach der Registerumbenennung einfach durch Vergleich der Registerbezeichner der physikalischen Register ermittelt werden, ohne dass die ursprüngliche Befehlsanordnung berücksichtigt werden muss. Dies ist eine Voraussetzung für die bei heutigen Superskalarprozessoren übliche Zuordnung auch außerhalb der Programmordnung, die die Befehle aus dem Befehlsfenster zuweist und dabei nur das Vorhandensein der Operanden, also echte Datenabhängigkeiten, aber keine Namensabhängigkeiten berücksichtigt.

Implementierungsalternativen der dynamischen Registerumbenennung

Eine dynamische Registerumbenennung kann in der Mikroarchitektur auf zwei Arten implementiert werden:

- Im ersten Fall sind für jede Registerart (allgemeine Register, Gleitkomma- und Multimediaregister) zwei verschiedene Registersätze physikalisch auf dem Chip vorhanden: die Architekturregister, die genau dem Registermodell der Prozessorarchitektur entsprechen, und zusätzliche Umbenennungspufferregister. Bei der Registerumbenennung werden den Architekturregistern Umbenennungspufferregister zugeordnet. Die Umbenennungspufferregister nehmen nur temporäre Resultatwerte auf, also solche, die am Ende einer Ausführungsstufe erzeugt und weiteren nachfolgenden Befehlen als Operanden wieder zur Verfügung gestellt werden, jedoch noch nicht rückgeordnet sind. Die Architekturregister speichern die „gültigen“ Registerwerte. Nach der Rückordnung müssen die gültigen Resultatwerte aus den Umbenennungspufferregistern in die Architekturregister übertragen und anschließend die Umbenennungspufferregister freigegeben werden. Beim PowerPC 604-Prozessor sind solche getrennten Architektur- und Umbenennungspufferregister sowie in der Pipeline eine eigene Rückspeicherstufe vorhanden.

- Im zweiten Fall existiert für jede Registerart jeweils nur ein Satz von so genannten physikalischen Registern, die in den Maschinenbefehlen nicht ansprechbar sind. Die Architekturregister sind als solche physikalisch nicht vorhanden. Die physikalischen Register speichern die temporären, noch nicht gültigen Resultate wie auch die bereits gültigen Werte. Die Bezeichnungen der Architekturregister werden dynamisch auf die physikalisch vorhandenen Register abgebildet. Es gibt *nur* eine Abbildungstabelle pro Registerart. Bei der Rückordnung werden die Registerwerte gültig gemacht, ein Umspeichern ist nicht nötig. Ein bereits gültiges physikalisches Register kann freigegeben werden, sobald ein nachfolgender Befehl das entsprechende Architekturregister als Resultatregister bezeichnet.

Das letztere Verfahren wird beim MIPS R10000 und bei den Intel Pentium-Prozessoren ab PentiumPro angewandt. Üblicherweise gibt es mehr physikalische Register als Architekturregister. Zum Beispiel definiert das IA-32-Registermodell der Pentium-Prozessoren acht allgemein verwendbare Architekturregister, wohingegen die Pentium-Prozessoren ab PentiumPro über 40 physikalische Register zur Umbenennung der allgemeinen Register verfügen. Die MIPS R10000-Architektur definiert 33 Architekturregister als allgemeine Register, verfügt aber über je 64 physikalische Register für die allgemeinen Register und die Gleitkommaregister.

Eine Alternative zur dynamischen Registerumbenennung ist, einen großen Registersatz (wie z.B. die 128 Ganzzahl- und die 128 Gleitkommaregister bei der Intel IA-64-Architektur) zu verwenden und eine statische Registerzuweisung (per Compiler) durchzuführen. Statische Registerumbenennung ist eine Compilertechnik, während die dynamische Registerumbenennung eine Mikroarchitekturtechnik ist. Auch werden für einen großen Registersatz mehr Bits für die Registerbezeichner im Befehl und damit längere Befehlsformate (als das übliche 32-Bit-Befehlsformat) benötigt. Weiterhin kann die Registerzugriffszeit die erreichbare Pipeline-Taktrate negativ beeinflussen. Allerdings wird die für eine dynamische Registerumbenennung notwendige Hardware gespart.

Die Decodier- und Umbenennungsbandbreite ist üblicherweise genauso groß wie die maximale Zuordnungsbandbreite. Nach der Registerumbenennung, die häufig keine eigene Pipeline-Stufe ist, sondern mit der Decodierstufe kombiniert wird, werden die Befehle in den Befehlspuffer, das sog. Befehlsfenster, geschrieben.

2.2.4 Befehlszuordnung

Befehlsfenster

Der Begriff des **Befehlsfensters** (*Instruction Window*) umfasst konzeptuell alle Befehlspufferplätze zwischen den Befehlsdecodier-/Registerumbenennungs- und den Ausführungsstufen. Das Befehlsfenster entkoppelt den Befehlsbereitstellungs- und Decodierteil vom Ausführungsteil des Prozessors. Der Befehlsbereitstellungs- und Decodierteil des Prozessors kann weiterarbeiten, ohne dass die vorherigen Befehle taktsynchron dazu ausgeführt werden müssen. Nach dem Decodieren und Umbenennen der Registerbezeichner können die Befehle im Befehlsfenster gespeichert werden, solange dort noch Plätze frei sind.

Die Befehle im Befehlsfenster sind durch die Sprungvorhersage frei von Steuerflussabhängigkeiten und durch die Registerumbenennung frei von Namensabhängigkeiten. Es müssen für die Zuordnung zu den Ausführungseinheiten nur noch die echten Datenabhängigkeiten geprüft und mögliche Ressourcenkonflikte beachtet werden. Die **Befehlszuordnung** (*Instruction Issue*) prüft, welche Befehle aus dem Befehlsfenster zugeordnet werden können, und weist diese Befehle den Ausführungseinheiten zu. Die Überprüfung der wartenden Befehle im Befehlsfenster und die Zuweisung von Befehlen bis zur maximalen Zuordnungsbandbreite geschieht in einem Takt. Die Programmreihenfolge der zugewiesenen Befehle wird im Rückordnungspuffer (*Reorder Buffer*) vermerkt.

Zuordnung

Wir benutzen den Begriff **Zuordnung** (*Issue*) für die Übertragung der Befehle in die Ausführungeinheiten oder, falls vorhanden, in die Umordnungspuffer (*Reservation Stations*) vor einer Ausführungseinheit oder einer Gruppe von Ausführungseinheiten. Falls solche Umordnungspuffer vorhanden sind, heißt die zweite Zuordnungsstufe der Befehle aus den Puffern an die Ausführungseinheiten *Dispatch*. Diese ist immer so organisiert, dass die Befehle auch außerhalb der Programmordnung zugeordnet werden können.

Die **Zuordnungsstrategie** (*Instruction-issue Policy*) beschreibt das Protokoll, mit dem Befehle für die Zuordnung ausgewählt werden. Je nach Prozessor können die Befehle nur in der sequenziellen Programmreihenfolge *(in-Order*) oder auch außerhalb der Reihenfolge (*out-of-Order*) zugewiesen werden. Die **Vorausschaufähigkeit** (*Lookahead Capability*) ist dabei die Fähigkeit, wie viele Befehle im Befehlsfenster untersucht werden, um die als nächstes zuordnungsbereiten Befehle zu untersuchen. Meist entspricht die Vorausschaufähigkeit der Größe des Befehlsfensters.

Man unterscheidet dynamische und statische Zuordnung sowie dynamisches und statisches Scheduling. Superskalarprozessoren sind in dieser Terminologie durch eine dynamische Zuordnung (*Dynamic Issue*) charakterisiert, d.h., es wird (dynamisch) von der Hardware entschieden, welche und wie viele Befehle pro Takt zugeordnet werden. Im Gegensatz dazu benutzen die VLIW-Prozessoren eine statische Zuordnung, d.h., pro Takt wird eine feste Anzahl von Befehlen den Ausführungseinheiten zugewiesen und diese Befehle sind vom Compiler festgelegt.

Die dynamische Zuordnung der Superskalarprozessoren kann mit einem statischen Scheduling (*Statical Scheduling*), d.h., die Zuordnungsreihenfolge entspricht der vom Compiler festgelegten Programmordnung, oder mit einem dynamischen Scheduling (*Dynamical Scheduling*) verknüpft werden. Bei letzterem kann die Zuordnungs-Hardware selbst entscheiden, in welcher Reihenfolge die Befehle zugeordnet werden. Die Befehlszuordnungslogik, die feststellt, welche Befehle ausführbereit sind, wird oft auch als **Scheduler** bezeichnet.

Man unterscheidet also die dynamische, d.h. superskalare, Zuordnung von der statischen, also VLIW-Zuordnung, und das dynamische (*out-of-Order*) vom statischen (*in-Order*) Scheduling. Statisches Scheduling war bei den superskalaren Prozessoren bis Mitte der 90er Jahre die Regel. Ein dynamisches Scheduling erhöht die Ausführungsgeschwindigkeit, da mehr Befehle für eine mögliche Zuordnung einbezogen werden, und wird deshalb bei allen heutigen Superskalarprozessoren angewandt.

Bevor die Befehle den In-order-Teil der Befehls-Pipeline verlassen, muss die ursprüngliche Befehlsanordnung im Rückordnungspuffer eingetragen werden. Dies kann im Fall des dynamischen Schedulings bereits beim Eintragen in das Befehlsfenster geschehen. Bei Anwendung des statischen Schedulings mit einer Zuordnung in Programmordnung genügt es, die Befehle bei ihrer Zuordnung zu den Ausführungseinheiten bzw. den Umordnungspuffern in den Rückordnungspuffer einzutragen.

In jedem Takt müssen im Befehlsfenster die Verfügbarkeit aller Eingabeoperanden eines Befehls geprüft, ausführbereite Befehle selektiert, die Verfügbarkeit der Ressourcen geprüft[8] und die Befehle zugeordnet werden. Sofern WAR- oder WAW-Konflikte (vgl. Unterabschnitt 1.5.3 in Kapitel 1) zwischen Befehlen vorliegen, kann durch eine Registerumbenennung die Zahl der zuordnungsbereiten Befehle erhöht werden.

Heutige superskalare Mikroprozessoren können vier bis sechs Anweisungen pro Takt aus einem 16 bis 56 Einträge fassenden Befehlsfenster zuordnen. Ein großes Befehlsfenster und eine ausgezeichnete Sprungvorhersage sind notwendig, um einen IPC-Wert (*Instructions Per Cycle*) zu erreichen, der nahe an der maximalen Zuordnungsbandbreite liegt.

Organisationsvarianten für das Befehlsfenster eines Superskalarprozessors

Die Prüfung der Datenabhängigkeiten und der Strukturkonflikte kann bei Superskalarprozessoren gleichzeitig in einer Stufe geschehen oder jeweils aufgeteilt auf eine eigene Stufe. Zur Organisation des Befehlsfensters gibt es die folgenden Alternativen:

- *Einstufige Zuweisung und zentrales Befehlsfenster* (Abbildung 2.10): Alle Befehle werden nach dem Decodieren und Registerumbenennen in einem zentralen Befehlsfenster gepuffert und in einer Pipeline-Stufe aus diesem zugeordnet. Bei der Zuordnung werden Daten- und Strukturkonflikte im gleichen Takt geprüft. Eine solche einstufige Zuordnung aus einem einzigen Befehlsfenster erfolgte bei den Pentium-II- und -III-Prozessoren. Die für die Zuordnung benötigte Hardware wird bei einer Vergrößerung des Befehlsfensters rasch so komplex, dass große Befehlsfenster die Taktrate des Prozessors beschränken.

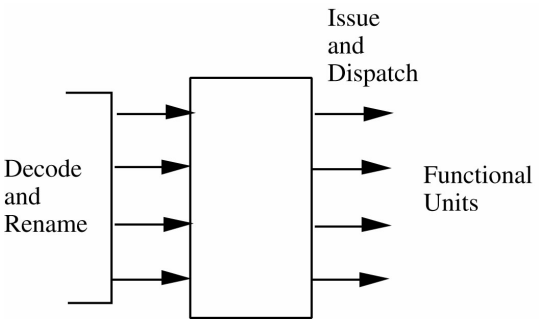


Abbildung 2.10: Einstufige Zuordnung und zentrales Befehlsfenster.

- *Einstufige Zuordnung und Entkopplung in verschiedenen Befehlsfenster* (Abbildung 2.11): Jedes Befehlsfenster führt zu einer Gruppe von Ausfüh-

[8] d.h., es muss eine geeignete Ausführungseinheit gefunden werden

rungseinheiten, meist für gleichartige Befehle. Beim HP-PA-8000-Prozessor werden separate Befehlsfenster für Gleitkomma- und Ganzzahleinheiten verwendet, beim MIPS-R10000-Prozessor sind separate Befehlsfenster für Gleitkomma-, Ganzzahl- und Adresseinheiten vorhanden und beim Pentium-4-Prozessor gibt es jeweils ein Befehlsfenster für die Gleitkomma- und SSE-Einheiten sowie für die Integer- und Adressgenerierungseinheiten. Die Prüfung der Datenabhängigkeiten wird vereinfacht, da die Datenabhängigkeiten auf die Befehle eines Fensters beschränkt werden können. Die Prüfung der Daten- und Strukturkonflikte und die Zuordnung geschieht in einer einzigen Pipeline-Stufe.

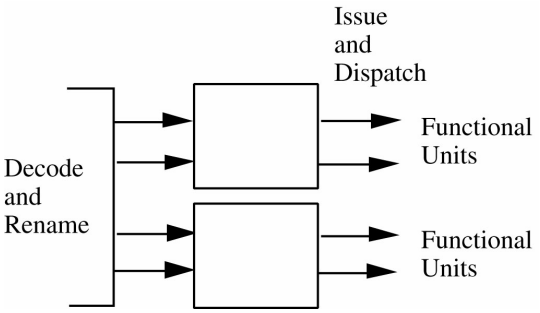


Abbildung 2.11: Einstufige Zuordnung mit entkoppelten Befehlsfenstern.

- *Mehrstufige Zuordnung mit einem Befehlsfenster aus mehreren hintereinander gelagerten Befehlspuffern*: Die Prüfung der Operanden- und Ressourcenverfügbarkeit wird in zwei (oder mehr) getrennten Pipeline-Stufen durchgeführt. Dabei kann zuerst eine ressourcenabhängige Zuweisung zu Umordnungspuffern (*Reservation Stations*) geschehen, die jeweils einen einzelnen Befehl aufnehmen. In der zweiten Zuordnungsstufe wird die Befehlsausführung in der Ausführungseinheit angestoßen, sobald die benötigten Operanden vorhanden sind. Abhängig vom Prozessor gehören die Umordnungspuffer zu einer Gruppe von Ausführungseinheiten (Pentium-Prozessoren) oder jede Ausführungseinheit hat einen eigenen Satz von Umordnungspuffern (PowerPC-Prozessoren). Ein Befehl wartet in einem Umordnungspuffer, bis alle Operanden verfügbar sind. Sollten bei der Zuweisung an den Umordnungspuffer schon alle Operanden verfügbar und die Ausführungseinheit nicht beschäftigt sein, so kann die Ausführung des Befehls schon direkt im folgenden Takt beginnen. Im Prinzip können die zwei Stufen auch in umgekehrter Reihenfolge angeordnet werden, also erst die Prüfung der Operandenverfügbarkeit und Zuweisung der ausführbereiten Befehle an die Umordnungspuffer vor den Ausführungseinheiten und in einer zweiten Stufe der Start der Ausführung, sobald eine Ausführungseinheit frei ist.

- *Kombination einer mehrstufigen Zuordnung und entkoppelter Befehlsfenster* (Abbildung 2.12): Die Entkopplung kann sogar bis zu einer vollständigen Verteilung auf die Ausführungseinheiten wie bei den PowerPC-Prozessoren erweitert werden. Die Befehle werden dann in der ersten Zuordnungstufe auf die entsprechenden Ausführungseinheiten verteilt. Die

Operandenverfügbarkeit, die in der zweiten Stufe geprüft wird, muss allerdings wieder Resultaterzeugungen von allen Ausführungseinheiten der Gruppe beachten. Die Zuordnung aus den Befehlsfenstern kann nun wieder in Programmordnung (statisches Scheduling) oder auch außerhalb der Programmordnung (dynamisches Scheduling) erlaubt sein. Beim zweistufigen Schema mit einer ressourcenabhängigen Zuweisung vor der Operandenverfügbarkeitsprüfung wird die erste Stufe in Programmreihenfolge durchgeführt, während die zweite Zuordnung auch außerhalb der Programmreihenfolge möglich ist. Falls nach der ersten Zuordnung die Operanden bereits vorhanden sind und die Ausführungseinheit frei ist, so wird die Ausführung aus den Umordnungspuffern sofort gestartet, ohne einen weiteren Takt zu benötigen. Abbildung 2.12 zeigt eine zweistufige Zuordnung mit einem zentralen Befehlsfenster auf der ersten Stufe und separate, auf die Ausführungseinheiten aufgeteilte Umordnungspuffer für die zweite Stufe (Beispiel: PowerPC 604).

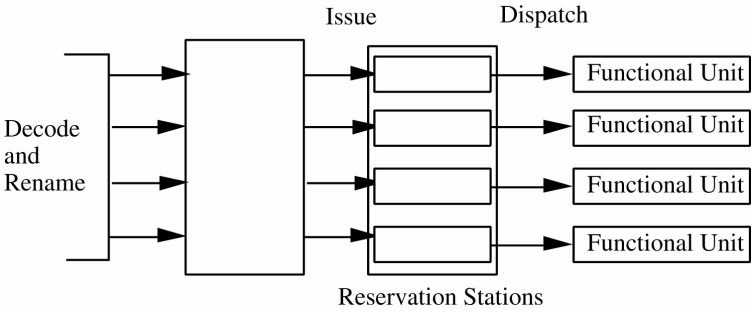


Abbildung 2.12: Zweistufige Zuweisung mit verteilten Umordnungspuffern.

2.2.5 Ausführungsstufen

In der oder den Ausführungsstufen werden die in den Opcodes der Befehle spezifizierten Operationen ausgeführt und die Resultate in Umbenennungspufferregistern oder physikalischen Registern gespeichert. Meist gibt es eine ganze Anzahl von spezialisierten Ausführungseinheiten auf dem Prozessor-Chip, die alle parallel zueinander arbeiten können. Die Ausführungseinheiten selbst können für die Ausführung einer Operation einen oder mehrere Taktzyklen benötigen. Je nach Operation kann die Ausführung ohne Pipelining geschehen oder eine Ausführungseinheit kann intern selbst wieder als Pipeline (arithmetische Pipeline oder Ausführungs-Pipeline genannt) organisiert sein.

Einzykleneinheiten

Die einfachste Art einer Ausführungseinheit ist die der Einzykleneinheiten, die eine Latenz von eins haben, da sie bereits im gleichen Takt, in dem die Befehlsausführung begonnen hat, das Resultat liefern. Da sie in der Regel auch pro Takt einen neuen Befehl ausführen können, weisen sie einen Durchsatz von 1 auf. Beispiele dafür sind einfache Ganzzahl- und Multimediaeinheiten.

Mehrtakteinheiten

Mehrtakteinheiten führen komplexe Operationen durch, die nicht in einer einzigen Pipeline-Stufe des Prozessors implementiert werden können. Sie haben also eine Latenz größer 1.

- Mehrtakteinheiten können ganz ohne Pipelining arbeiten, also einen Durchsatz haben, der reziprok der Latenz entspricht, d.h. der Anzahl der für die Ausführung einer Operation benötigten Takte. Mehrtakteinheiten ohne Pipeline-Verarbeitung und mit einer Latenz von mehr als 1 sind die Divisions-, die Quadratwurzel- und komplexe Multimediaeinheiten.

- Mehrtakteinheiten können selbst wieder als mehrstufige Pipeline mit einer festen Zahl an Stufen arbeiten und in jedem Takt oder jedem zweiten Takt eine neue Befehlsausführung starten, also einen Durchsatz von 1 oder 1/2 bieten. Beispiele für Mehrtakteinheiten mit Pipeline-Verarbeitung und Durchsatz 1 sind komplexe Ganzzahl- und gleitkommaorientierte Multimediaeinheiten.

- Mehrtakteinheiten können andererseits eine Pipeline mit variabler, von der Operation abhängiger Latenz haben.

Nach Beendigung der Befehlsausführung in der Ausführungseinheit (*Completion Unit* genannt) wird das Resultat in einem temporären Register (Umbenennungspufferregister oder physikalisches Register) gepuffert und ein Zeichen für das Vorliegen des Resultats, das sog. *Tag*, an die Befehle im Befehlsfenster weitergeleitet. Das Resultat steht, obwohl noch nicht „gültig“ gemacht, damit bereits als Operand für die Ausführung datenabhängiger Befehle auch in anderen Ausführungseinheiten zur Verfügung.

Es folgt nun eine Bechreibung der wichtigsten Ausführungseinheiten.

einfache Ganzzahleinheit

Eine **einfache Ganzzahleinheit** (*Simple Integer Unit*) enthält eine ALU, die alle 32-Bit- oder 64-Bit-Festpunktadditionen, die Schiebe-, Rotations- und die logischen Operationen ausführt.

komplexe Ganzzahleinheit

Eine **komplexe Ganzzahleinheit** (*Complex Integer Unit*) führt die komplexeren Ganzzahloperationen aus. Dazu gehören die 32- und 64-Bit-Ganzzahlmultiplikationen. Für solche Ganzzahlmultiplikationen gibt es üblicherweise Mehrtakteinheiten mit Pipeline-Verarbeitung mit einem Durchsatz von 1. (Der Multiplizierer benutzt dazu Standardalgorithmen der Rechnerarithmetik, die in einschlägigen Büchern oder auch im Anhangskapitel von [9] beschrieben werden.)

Divisionseinheit

Für die ganzzahlige Division kann eine eigene **Divisionseinheit** vorhanden sein oder eine komplexe Ganzzahleinheit führt die Division aus. Die Latenz hängt vom Operandentyp und von der benötigten Genauigkeit ab und bewegt sich in der Größenordnung von 13 bis 17 für eine 32-Bit-Ganzzahldivision. Die Division wird ohne Pipelining implementiert. Die Divisionseinheit wird meist auch für die Quadratwurzelbildung benutzt, falls ein solcher Befehl wie beispielsweise beim MIPS R10000 im Befehlssatz vorhanden ist.

Gleitkommaeinheiten

Gleitkommaeinheiten (*Floating-point Units*) sind als Pipeline implementiert und können eine Gleitkommaoperation nach einfach- oder doppelt-genauem IEEE 754-Format ausführen. Typischerweise ist der Durchsatz gleich 1 bei einer

Latenz von 3 Takten. Die Rundung und Normalisierung der Gleitkommazahlen kann beim IEEE-Standard besonders komplex ausfallen. Deshalb wird häufig nicht der volle IEEE-Standard in Hardware implementiert.

Lade-/Speichereinheiten

Lade-/Speichereinheiten (*Load/Store Units*) sind komplexe Einheiten, die hier nur kurz beschrieben werden. Sie benötigen für einen Zugriff auf den Primär-Daten-Cache-Speicher meist zwei oder drei Takte, haben – im Falle eines Treffers im Primär-Cache-Speicher – also eine Ladelatenz von 2 oder 3. Für Lade- und für Speicherbefehle existieren meist zwei verschiedene Wartepuffer innerhalb der Lade-/Speichereinheit.

Ein **Ladebefehl** gilt als von der Lade-/Speichereinheit beendet (*completed*), wenn der zu ladende Wert in einem Umbenennungspufferregister steht. Bei einem **Speicherbefehl** ist dies komplizierter. Er benötigt zusätzlich zur Adressrechnung auch noch den zu speichernden Wert, der häufig von vorangehenden arithmetischen Operationen erst geliefert werden muss. Eine Speicheroperation kann außerdem nicht mehr rückgängig gemacht werden. Der Speicherbefehl kann damit erst „beendet“ werden, also der Wert wirklich in den (Cache-)Speicher geschrieben werden, wenn er in der Rückordnungsstufe als „gültig“ markiert (*committed*) wird. Bei manchen Prozessoren können deshalb die Ladebefehle, die nur die Adressrechnung benötigen, *vor* den Speicheroperationen ausgeführt werden, sofern nicht dieselbe Adresse betroffen ist. Als Beispiel betrachte man die Implementierung der Lade-/Speichereinheit in Abbildung 2.13. Ladebefehle werden sofort ausgeführt. Speicherbefehle werden zunächst in einem internen, als FIFO organisierten Schreibpuffer (*Write Buffer*) der Lade-/Speichereinheit untergebracht.

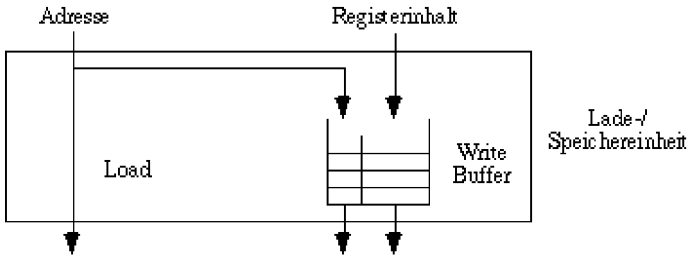


Abbildung 2.13: Lade-/Speichereinheit.

Während ein Speicherbefehl auf seinen Datenwert wartet, kann ein nachkommender Ladebefehl ihn überholen und vor ihm auf den Daten-Cache-Speicher zugreifen. Zuerst wird aber überprüft, dass der Lade- und der Speicherbefehl nicht dieselbe Zieladresse haben. Damit wird verhindert, dass statt eines abzuspeichernden Datenwerts, der im Schreibpuffer hängt und noch nicht geschrieben wurde, von einem in der Programmordnung nachfolgenden Lesebefehl fälschlicherweise ein veralteter Datenwert aus dem Cache-Speicher gelesen wird. Das Prinzip, dass Ladezugriffe vor Speicherzugriffe gezogen werden, sofern nicht dieselbe Adresse betroffen ist und kein Spezialbefehl (Synchronisationsbefehl, *swap*-Befehl oder markierter Befehl) dazwischen liegt, ist in vielen Mikroprozessoren verwirklicht.

Damit wird die Verarbeitungsgeschwindigkeit des Prozessors erheblich verbessert, allerdings führt dies zu einer abgeschwächten Speicherkonsistenz (vgl. Unterabschnitt 3.1.3.6)[9], da nun die nach außen sichtbare Befehlsausführungsreihenfolge nicht mehr derjenigen der Programmordnung entspricht.

Wenn beim Vorziehen eines Ladebefehls vor einen Speicherbefehl dieselbe Adresse betroffen ist, so gibt es zwei Möglichkeiten: Der zu ladende Wert wird aus dem Speicherpuffer entnommen oder der geladene Wert wird nachträglich gelöscht und die Ladeoperation wiederholt. Natürlich kann ein Ladebefehl nur dann vor einen Speicherbefehl gezogen werden, wenn die Adresse des zu ladenden Werts und die Adresse des zu speichernden Werts bereits berechnet sind.

Meist ist nur eine Lade-/Speichereinheit in einem Prozessor vorhanden, da ein gleichzeitiger Zugriff von mehreren Lade-/Speichereinheiten auf den Daten-Cache die Cache-Speicherverwaltung noch mehr erschwert. Bei vielfach superskalaren Prozessoren sind jedoch mehrere Lade-/Speichereinheiten wichtig, da sonst der Prozessor nicht mit genügend Daten versorgt werden kann. Eine Lösung, um zwei Speichereinheiten zu ermöglichen, besteht darin, den Daten-Cache-Speicher mit doppelter Taktfrequenz wie die Pipeline zu betreiben. Damit wird jedoch die Taktfrequenz der Pipeline selbst beschränkt.

Multimediaeinheiten

Multimediaeinheiten führen mehrere gleichartige Operationen auf Teilen von Registersätzen gleichzeitig aus. Damit kann eine sehr feinkörnige Parallelität genutzt werden, die man als *Subword Parallelism* oder als SIMD-Parallelität (*Single Instruction Multiple Data*) bezeichnet. Dieselbe Operation, wie sie durch den Opcode gegeben ist, wird auf mehreren Dateneinheiten gleichzeitig ausgeführt. Diese SIMD-Parallelität war vor der Einführung der Multimediaeinheiten bereits von den Feldrechnern her bekannt.

Multimediaoperationen sind arithmetische oder logische Befehle auf gepackten Datentypen wie z.B. acht 8-Bit-, vier 16-Bit- oder zwei 32-Bit-Teilwörtern, die jeweils in einem 64-Bit-Doppelwort untergebracht sind. Operationen sind das Packen und Entpacken in bzw. von Teilwörtern sowie Maskier-, Selektions-, Umordnungs-, Konversions-, Vergleichs-, arithmetische und logische Operationen auf diesen Teilwörtern. Natürlich sind die Teilwörter häufig zu klein, um die Resultate der arithmetischen Operationen aufnehmen zu können. Deshalb wird bei den arithmetischen Operationen eine Saturationsarithmetik angewandt, die keinen Über- oder Unterlauf des Zahlbereichs kennt, sondern solche Ergebnisse auf die größte oder die kleinste Zahl des darstellbaren Zahlenbereichs abbildet.

Heute haben alle allgemein verwendeten Mikroprozessoren Multimediaerweiterungen für die Video-, die Audio- und die Sprachverarbeitung. Solche bitfeldorientierten Multimediaerweiterungen bei heutigen Mikroprozessoren sind:

- der *Visual Instruction Set* (VIS) bei den Sun UltraSPARC-Prozessoren als eine der ersten Multimediaerweiterungen,
- die MMX und MMX2 (*Multi Media eXtensions*) genannten Erweiterungen für die Intel IA-32-Prozessoren,

[9] Vereinfachend versteht man unter Speicherkonsistenz, dass alle Kopien eines Datums in den verschiedenen Speichereinheiten stets identisch sind.

- die AltiVec-Erweiterung für die Motorola/IBM PowerPC-Prozessoren.

Beim Intel P55C und beim Pentium-II wurden die acht 64 Bit breiten Multimediaregister mit den Gleitkommaregistern überlappt, sodass Multimediabefehle und Gleitkommabefehle nicht gleichzeitig ausgeführt werden konnten. VIS, MVI und MDMX erweitern 64-Bit-RISC-Prozessoren, die bereits 64 Bit breite allgemeine Register besitzen, die gleichzeitig auch als Multimediaregister benutzt werden können.

Die Unterstützung der 2D- und 3D-Graphikverarbeitungen benötigt schnelle Gleitkommaoperationen sowie reziproke Operationen mit geringer Genauigkeit. Bei den graphikorientierten Multimediaerweiterungen werden zwei (oder vier) 32-Bit-Gleitkommaoperationen gleichzeitig auf zwei (oder vier) 32-Bit-Gleitkommazahlenpaaren, die in 64- oder 128-Bit-Multimediaregistern untergebracht sind, ausgeführt.

Solche graphikorientierten Multimediaerweiterungen wurden erstmals durch die 3DNow!-Erweiterung von AMD und anschließend durch Intels Befehlssatz-Erweiterung ISSE (*Internet Streaming SIMD Extension*) zu MMX-2 implementiert. 3DNow! definiert 21 neue Multimediabefehle, hauptsächlich als gepaarte 32-Bit-Gleitkommaoperationen. Der ISSE-Befehlssatz wurde erstmals 1999 im Intel Pentium-III-Prozessor implementiert. MMX-2 definiert 72 neue Befehle, die auf einem Satz von acht 128 Bit breiten Multimediaregistern arbeiten oder als „stromorientierte“ Ladebefehle die Multimediadaten bereitstellen. Damit können vier 32-Bit-Gleitkommaoperationen parallel ausgeführt werden.

Zukünftige Ausführungseinheiten werden wohl noch komplexer als die heutigen sein. Angedacht sind beispielsweise Gleitkommavektoreinheiten, hochgenaue Skalarprodukteinheiten oder ganze spezialisierte Multimediaeinheiten wie beispielsweise eine MPEG-Einheit (*Motion Picture Experts Group*) oder eine 3D-Graphikeinheit.

2.2.6 Gewährleistung der sequenziellen Programmsemantik

2.2.6.1 Rückordnungsstufe

In der **Rückordnungsstufe** werden die Resultate der Befehlsausführungen gültig gemacht oder verworfen, das Rückrollen von falsch spekulierten Ausführungspfaden nach einem Sprung überwacht und präzise Unterbrechungen durchgeführt.

Man muss dabei die folgenden Begriffe unterscheiden [10]:

Wichtige Begriffsdefinitionen der Rückordnungsstufe!

- Die Beendigung eines Befehls (*Completion*), d.h., die Ausführungseinheit hat die Ausführung des Befehls abgeschlossen, das Resultat steht in einem Pufferregister und wird datenabhängigen Befehlen als Operand zur Verfügung gestellt. Dies geschieht unabhängig von der Programmordnung.

- Nach der Beendigung werden die Befehle in Programmordnung gültig gemacht (*Commitment*), d.h. die Resultate können nicht mehr rückgängig

[10] Leider werden die englischen Begriffe *Completion*, *Retirement* und *Commitment* in der Literatur häufig in vertauschten Bedeutungen verwendet.

gemacht werden und der Befehl wird aus dem Rückordnungspuffer entfernt.

- Das Löschen eines Befehls (*Removement*) bedeutet, dass der Befehl aus dem Rückordnungspuffer entfernt wird, ohne dass der Resultatwert weiter verwendbar ist.

- Die Rückordnung (*Retirement*) bedeutet das Entfernen des Befehls aus dem Rückordnungspuffer mit oder ohne das Gültigmachen des Resultats.

Ein Resultat wird dadurch gültig gemacht, dass entweder die Abbildung des Architekturregisters auf das physikalische Register „gültig“ gemacht wird (falls keine von den Architekturregistern verschiedenen Umbenennungspufferregister existieren), oder durch Kopieren des Resultatwerts von seinem Umbenennungspufferregister in sein Architekturregister. Das Kopieren geschieht bei den PowerPC-Prozessoren in einer separaten Rückschreibstufe der Pipeline nach der Rückordnungsstufe und das Umbenennungspufferregister wird nach dem Kopieren wieder freigegeben.

2.2.6.2 Präzise Unterbrechungen

Wie bereits gesagt, wird eine Unterbrechung (*Interrupt* oder *Exception*) als präzise bezeichnet, wenn der bei Ausführung der Unterbrechungsroutine gesicherte Prozessorzustand mit dem sequenziellen Ausführungsmodell der von-Neumann-Architektur konform geht, bei dem eine Befehlsausführung vollständig beendet ist, bevor mit der nächsten Befehlsausführung begonnen wird.

Unterbrechungen gehören in die folgenden Klassen:

- Programmunterbrechungen (*Traps*) werden durch Ausnahmebedingungen während der Befehlsausführung in der Pipeline hervorgerufen. Diese Ausnahmen können durch nicht statthaften Code, Privilegienverletzungen oder durch numerische Fehler wie Überlauf, Unterlauf, Division durch Null hervorgerufen werden. Diese fatalen Ausnahmen führen meist zu einem kontrollierten Programmabbruch durch die aktivierte *Trap*-Routine. Wesentlich ist, dass beim Programmabbruch der Fehler und die auslösende Programmstelle angegeben werden kann. Ob die Unterbrechung präzise oder nicht präzise erfolgt, ist hier oft nicht so wesentlich und hängt vom Prozessor ab.

- Programmunterbrechungen können aber auch beispielsweise durch Seitenfehler (*Page Fault*), d.h. durch den Zugriff auf ein Datum, das noch nicht im Hauptspeicher liegt, oder TLB-Fehlzugriffe[11] verursacht und damit Teil der normalen Ausführung sein. In diesen Fällen darf der Befehl nicht ausgeführt werden, sondern seine Ausführung muss nach der Ausführung der Behandlungsroutine wiederholt werden. Die Gewährleistung einer präzisen Unterbrechung ist für eine korrekte Programmausführung notwendig.

[11] Der TLB (*Translation Lookaside Buffer*) ist ein kleiner Cache-Speicher, der die letzten Adressumrechnungen der virtuellen Speicherverwaltung enthält, vgl. Kapitel 3.

- Externe Unterbrechungen (*Interrupts*) werden von Quellen außerhalb des Prozessors ausgelöst. Das sind beispielsweise Ein-/Ausgabe- oder Zeitgeber-Unterbrechungen. Bei diesen Unterbrechungen muss ein Weiterführen der Programmausführung durch Gewährleistung einer präzisen Unterbrechung vorhanden sein.

durch Befehle verursachte Unterbrechungen

Der gesicherte Zustand, der bei einer Unterbrechung der beiden ersten Klassen auftritt, muss für die Erzeugung einer präsisen Unterbrechnung die folgenden Bedingungen erfüllen:

- Alle Befehle, die in der Programmordnung *vor* dem Befehl stehen, der die Unterbrechung ausgelöst hat, sind vollständig ausgeführt worden und haben den Prozessorzustand entsprechend modifiziert.
- Alle Befehle, die in der Programmordnung *nach* dem Befehl stehen, der die Unterbrechung ausgelöst hat, sind nicht ausgeführt worden und haben den Prozessorzustand nicht beeinflusst.
- Falls die Unterbrechung von einem Ausnahmezustand bei der Befehlsausführung ausgelöst wurde, zeigt der Befehlszähler auf den Befehl, der die Unterbrechung ausgelöst hat. Je nach Art des Befehls sollte der auslösende Befehl noch vollständig ausgeführt oder vollständig aus der Pipeline gelöscht werden.

externe Unterbrechungen

Externe Unterbrechungen und Unterbrechungen durch illegale Befehle oder Privilegienverletzungen, die in der Decodiereinheit erkannt werden, können in präziser Weise wie folgt implementiert werden: In Superskalarprozessoren bleiben die Befehle bis zum Befehlsfenster in Programmordnung. Die Programmunterbrechung führt nun zum Anhalten der Befehlszuordnung. Weiterhin wartet der Prozessor, bis alle bereits zugeordneten Befehle aus dem Rückordnungspuffer entfernt worden sind. Dann befindet sich der Prozessor in einem klar definierten Zustand, wobei der Befehlszähler des obersten Befehls im Befehlsfenster den Aufsetzpunkt für die Programmweiterführung nach der Ausnahmebehandlung angibt. Natürlich müssen vor dem Starten der Ausnahmebehandlungs-Routine alle Befehle aus dem Befehlsfenster und den Puffern vor dem Befehlsfenster ebenfalls gelöscht werden.

Falls der gesicherte Prozessorzustand nicht mit dem sequenziellen Ausführungsmodell übereinstimmt und die obigen Bedingungen nicht erfüllt, so wird die Unterbrechung als eine nicht präzise (*imprecise*) Unterbrechung bezeichnet.

2.2.6.3 Rückordnungspuffer

Um die Rückordnung in Programmreihenfolge und präzise Unterbrechungen zu implementieren, wird bei heutigen Superskalarprozessoren meist ein **Rückordnungspuffer** (*Reorder Buffer*) eingesetzt. Dieser speichert die Programmordnung der Befehle nach ihrer Zuordnung und ermöglicht die Serialisierung der Resultate (*Result Serialization*) während der Rückordnungsstufe. Wenn ein Befehl seine Ausführung in einer Ausführungseinheit beendet hat, wird dieser

Zustand im Rückordnungspuffer notiert. Das Gleiche gilt, wenn der Befehl auf eine Ausnahme gestoßen ist und eine Unterbrechung auslösen soll. Der Rückordnungspuffer wird als zyklischer FIFO-Puffer implementiert. Die Befehlseinträge im Rückordnungspuffer werden in der ersten Zuordnungsstufe belegt und durch die Rückordnung wieder freigegeben.

Während der Rückordnung wird entsprechend der Rückordnungsbandbreite eine Anzahl von Befehlseinträgen am Kopf des FIFO-Puffers untersucht. Meist ist die Bandbreite der Rückordnungseinheit dieselbe wie die Zuordnungsbandbreite. Ein Befehlsresultat wird gültig gemacht, wenn alle vorherigen Befehle gültig gemacht wurden oder im gleichen Takt gültig gemacht werden. Schon ausgeführte Befehle auf fehlspekulierten Ausführungspfaden werden aus dem Rückordnungspuffer gelöscht und die Umbenennungspufferregister oder physikalischen Register werden freigegeben. Das gleiche passiert für alle nach der Fehlspekulation stehenden Befehle im Rückordnungspuffer. Die Befehlsholeeinheit wird angewiesen, die Befehle auf dem korrekten Ausführungspfad bereitzustellen.

Es gibt verschiedene Organisationsvarianten für Rückordnungspuffer, die sich von dem in diesem Kapitel beschriebenen Rückordnungspuffer unterscheiden. Auf sie kann in diesem Kurs aus Platzgründen nicht eingegangen werden.

2.2.7 Rückordnung ohne Sequenzialisierung

Die Rückordnung geschieht immer strikt in Programmordnung, um die vom von-Neumann-Prinzip geforderte Resultatserialisierung zu gewährleisten. Die einzige Ausnahme davon ist das Vorziehen der Lade- vor die Speicherbefehle, das einige Prozessoren erlauben. Deshalb muss auch ein intern vielfach parallel arbeitender Superskalarprozessor nach außen hin wie ein einfacher von-Neumann-Rechner aus den 50er Jahren wirken.

Um das zu hinterfragen, sollte man sich überlegen, dass ein Algorithmus eigentlich eine halbgeordnete Menge von Aktionen darstellt: Manche dieser Aktionen sind voneinander abhängig, andere können unabhängig zueinander in beliebiger Reihenfolge ausgeführt werden. Die heute gängigen Programmiersprachen sind praktisch ausschließlich Sprachen, die auf einer Abstraktion des sequenziellen von-Neumann-Operationsprinzip beruhen und den Programmierer zwingen, die algorithmischen Schritte sequenziell darzulegen. Das sequenzielle Programm wird einem Compiler übergeben, der für seine Optimierungen versucht, möglichst viel Parallelität aufzufinden und in seiner Zwischensprache darzustellen. Die Codegenerierung für einen Superskalarprozessor durch den Compiler bedeutet erneut das Sequenzialisieren der Anweisungen zu einem Maschinenprogramm, denn auch Maschinenprogramme gehorchen dem durch das von-Neumann-Prinzip gegebenen Zwang zur Sequenzialisierung. Der Superskalarprozessor versucht nun erneut die Programmparallelität aus der Maschinenbefehlsfolge wiederzugewinnen, um die Ausführungsgeschwindigkeit zu erhöhen. Allerdings erzwingt die Rückordnung wiederum die Resultatserialisierung.

Man sieht, dass es genügend Stellen gibt, um die vom von-Neumann-Prinzip geforderte Sequenzialisierung abzustreifen. VLIW-/EPIC-Prozessoren ermögli-

chen es, parallel ausführbare Befehle in einem Befehlstupel unterzubringen. Sie durchbrechen damit die Restriktion des von-Neumann-Prinzips. Allerdings müssen die Befehlstupel in streng sequenzieller Weise hintereinander ausgeführt werden. Die Mehrkernprozessoren und mehrfädigen Prozessoren ermöglichen es, mehrere parallele Kontrollfäden (*Threads*) gleichzeitig auf dem Prozessor-Chip auszuführen. In Verbindung mit einem Thread-Konzept in Programmiersprachen wie Java oder in mehrfädigen Betriebssystemen kann damit grobkörnige Parallelität vom Prozessor genutzt werden.

Eine Rückordnung außerhalb der Programmordnung ist bei heutigen Superskalarprozessoren nicht zugelassen. Trotzdem wäre sie möglich. Man nehme an, eine Befehlsfolge A endet mit einem Sprungbefehl, der die Vorhersage der Befehlsfolge B auslöst. B sei von einer Befehlsfolge C gefolgt und C sei unabhängig von B. Damit kann C unabhängig von der Sprungrichtung des Sprungbefehls ausgeführt werden. Die Befehle in C könnten bereits rückgeordnet werden, bevor B ausgeführt ist. Falls zur Implementierung von B die Prädikation angewandt wird, so kann der Sprungbefehl entfernt werden. Auch dann könnten die Befehle von C vor den prädikativen Befehlen aus B rückgeordnet werden.

Es treten allerdings zwei Komplikationen auf: Eine Unterbrechung, die von einem Befehl aus B ausgelöst wird, kann nur schwer in präziser Weise behandelt werden, da nachfolgende Befehle auf C bereits rückgeordnet sein können. (Der *History Buffer* wäre dabei eventuell hilfreich.) Die zweite Schwierigkeit ist, dass die sequenzielle Programmordnung geopfert wird, die zwar ein Hindernis für die Ausnutzung der vorhandenen Parallelität ist, doch Korrektheitsbeweise des Programms erleichtert.

2.3 Lösungen zu den Selbsttestaufgaben

Selbsttestaufgabe 2.1 von Seite 89

a) Superpipeline-Prozessoren nutzen zeitliche Parallelität durch eine Verdopplung der Taktfrequenz in Verbindung mit einer längeren Pipeline; superskalare Prozessoren machen sich räumliche Parallelität durch mehr Ausführungseinheiten zu Nutze.

b) VLIWs benötigen eine statische, also compilerbasierte Planung der parallel ausführbaren Befehle, während Superskalarprozessoren die Befehlsebenenparallelität dynamisch, also zur Laufzeit entdecken und nutzen.

Selbsttestaufgabe 2.2 von Seite 98

Man betrachte das obige Programmbeispiel mit dem in Tabelle 2.1 gegebenen Programmablauf und verfolge dabei 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	NT	T	NT
s2	T	NT	T	NT

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

Sprung	Initialzustand	d=0	d=2	d=0
s1	WT	WNT	WT	WNT
s2	WT	WNT	WT	WNT

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

Sprung	Initialzustand	d=0	d=2	d=0
s1	WT	SNT	WNT	SNT
s2	WT	SNT	WNT	SNT

Selbsttestaufgabe 2.3 von Seite 99

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 ; (1) R3 <= R0 ? Status in R10
        BNEZ R10, Ende1 ; (2) wenn ja, dann gehe zu Ende1
Loop2:  SLE R11, R4, R0 ; (3) R4 <= R0 ? Status in R11
        BNEZ R11,Ende2  ; (4) wenn ja, dann gehe zu Ende2
    .
    .
    .
        SUB R4, R4, R1 ; (19) R4 = R4 -- R1
        J Loop2        ; (20) Goto Loop2
Ende2:  SUB R3, R3, R1 ; (21) R3 = R3 -- R1
        J Loop1        ; (22) Goto Loop1
Ende1:  ...            ; (23)
```

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.

Lösung:

Die folgende Tabelle zeigt die bedingten Sprünge und die Zustände der Sättigungszähler nach dem jeweiligen Sprung. Die Abkürzungen und Zeichen sind unter der Tabelle erläutert.

Sprung		Ein-Bit-Prädiktor		Zwei-Bit_Prädiktor		
äußere Schleife	innere Schleife	äußere Schleife	innere Schleife	äußere Schleife	innere Schleife	
Initialzustand		T	T		WT	
ng		x NT		x WNT		
	ng		x NT		x WNT	
	ng		√ NT		√ SNT	
	.		.		.	m
	.		.		.	
	.		.		.	
	ng		√ NT		√ SNT	
	g		x T		x WNT	
ng		√ NT		√ SNT		
	g		√ T		x WT	*1*
ng		√ NT		√ SNT		
	g		√ T		√ ST	*1*
ng		√ NT		√ SNT		
.	.	.	.	.	.	
.	.	.	.	.	.	
.	.	.	.	.	.	
ng		√ NT		√ SNT		
	g		√ T		√ ST	*1*
ng		√ NT		√ SNT		
	g		√ T		√ ST	*1*
g		x T		x WNT		

n

Erläuterung der Abkürzungen:

√	Sprung richtig vorhergesagt
×	Sprung falsch vorhergesagt
g	Sprung wurde genommen
ng	Sprung wurde nicht genommen
T	predict taken
NT	predict not taken
ST	predict strongly taken
WT	predict weakly taken
WNT	predict weakly not taken
SNT	predict strongly not taken

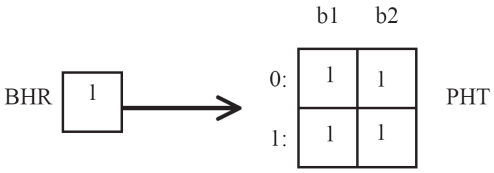
Für die äußere Schleife ergeben sich ein genommener und n nicht genommene Sprünge und für die innere Schleife n genommene und m nicht genommene Sprünge.

	äußere Schleife		innere Schleife	
	richtig	falsch	richtig	falsch
a)	1	n	n	m
b)	$n-1$	2	$n + m$ - *2*	2
c)	$n-1$	2	$n + m$ - *3*	3

Selbsttestaufgabe 2.4 von Seite 101

Man betrachte erneut das Programmbeispiel aus Selbsttestaufgabe 2.2 mit dem in Tabelle 2.1 gegebenen Programmablauf. Im Text heißt es: „Ein (1,1)-Korrelationsprädiktor kann den Zusammenhang zwischen beiden Sprüngen erkennen und nutzen. Er trifft nur in der ersten Iteration eine falsche Vorhersage."

Vervollständigen Sie die Zustände der Prädiktortabelleneinträge für die beiden Sprünge bei Verwendung eines (1,1)-Korrelationsprädiktors (Initialzustand: „predict taken") und begründen Sie damit die obige Aussage.



Lösung:

Die Prädiktortabelle ergibt sich wie folgt:

Sprung	Initial-zustand	$d = 0$	$d = 2$
$s1$	T	NT	T
$s2$	T	NT	T

Beim ersten Durchlauf mit der Startbedingung $d = 0$ ergibt sich nach dem ersten Sprung:

$$\text{PHT(nach s1)} = \begin{array}{|c|c|}\hline 1 & 1 \\ \hline 0 & 1 \\ \hline \end{array}$$

Das BHR ist 0 und nach dem zweiten Sprung des ersten Durchlaufs wird bereits der stationäre Zustand erreicht, d.h. danach ändern sich die Einträge in der Sprungverlaufstabelle nicht mehr. Die resultierende PHT hat ab diesem Zeitpunkt folgende Belegung:

$$\text{PHT(nach s2)} = \begin{array}{|c|c|}\hline 1 & 0 \\ \hline 0 & 1 \\ \hline \end{array}$$

Korrelation bedeutet, dass die Sprungvorhersagen sich wechselseitig beeinflussen. Im vorliegenden Fall hat der Korrelationsprädiktor die direkte Abhängigkeit der beiden Sprünge „gelernt". Er trifft nach Erreichen des stationären Zustands stets die richtige Entscheidung. In der PHT haben die Sprünge komplementäre Sprungvorhersagen, weil jeweils nach dem ersten Sprung das BHR komplementiert wird.

Kapitel 3

Speicherverwaltung und innovative Mikroprozessortechniken

Kapitelinhalt

Zusammenfassung

Dieses Kapitel beschäftigt sich zunächst mit der Speicherhierarchie von Rechnern, wie sie insbesondere bei heutigen PC-Systemen zu finden ist. Dazu gehören nach einer Einführung in die Speicherhierarchie insbesondere die Hardwaretechniken zur Unterstützung einer virtuellen Speicherverwaltung und die Organisation von Cache-Speichern. Weiterhin beschäftigt es sich mit neuen Techniken für Mikroprozessoren. Nach einer Analyse der Grenzen heutiger Prozessortechniken werden innovative Techniken aus der Forschung und Entwicklung von Mikroprozessoren behandelt, die teilweise gerade ihren Einzug in kommerzielle Mikroprozessoren halten.

Lernziele

Die Lernziele dieses Kapitels sind das Erreichen des Verständnisses:

- der Stufen einer Speicherhierarchie
- der durch Caches genutzten Lokalität
- der Vorteile einer virtuellen Speicherverwaltung
- der Grenzen gegenwärtiger Prozessortechniken
- der Forschungsrichtungen zur Weiterentwicklung von Mikroprozessoren

3.1 Speicherverwaltung

3.1.1 Speicherhierarchie

Heutige Mikroprozessoren müssen wegen ihrer ausgeprägten Fähigkeit, Befehle intern simultan (parallel) zu verarbeiten, und ihrer hohen Taktraten ihrem Befehlsbereitstellungs- und Decodierteil genügend Befehle als auch ihrem Ausführungsteil genügend viele Daten zuführen. Da die Daten nicht immer nur den Registern entnommen werden können, wachsen mit der Erhöhung der Verarbeitungskapazität des Prozessors auch die Anforderungen an die Bandbreite, mit der dem Prozessor Daten aus dem Speicher zugeführt werden müssen.

Eine weitere Beobachtung ist, dass der Speicherbedarf eines Programms mit seinen während der Ausführung erzeugten Zwischendaten häufig sehr groß werden kann und u.U. die Kapazität des Hauptspeichers sprengt. Ideal wäre ein einstufiges Speicherkonzept, bei dem mit jedem Prozessortakt auf jedes Speicherwort zugegriffen werden kann. Das ist technologisch für Prozessoren hoher Leistung nicht möglich, denn große Speicher existieren nur mit relativ langsamem Zugriff, während Speicherbausteine mit hoher Zugriffsgeschwindigkeit in ihrer Speicherkapazität beschränkt und teuer sind.

Technologisch klafft eine immer größer werdende Lücke zwischen der Verarbeitungsgeschwindigkeit des Prozessors und der Zugriffsgeschwindigkeit der heute üblichen DRAM-Speicherchips (*Dynamic Random Access Memory*)[1] des Hauptspeichers. Die hohen Prozessortaktraten und die Fähigkeit superskalarer Mikroprozessoren, mehrere Operationen pro Takt auszuführen, erzeugen von Seiten des Prozessors einen immer größeren „Hunger" nach Code und Daten aus dem Speicher. Die Geschwindigkeit dynamischer Speicherbausteine hat hingegen über die Jahre hinweg aus technologischen Gründen deutlich weniger zugenommen. Eine Ausführung der Programme direkt aus dem Hauptspeicher hätte zur Folge, dass der Prozessor nur mit einem Bruchteil seiner maximalen Leistung arbeiten könnte. Deshalb muss der Prozessor seine Befehle vorwiegend aus dem auf dem Chip befindlichen Code-Cache-Speicher und seine Daten aus den Registern und dem Daten-Cache-Speicher erhalten.

Unter einem **Cache** ist dabei ein sehr schneller, aber im Vergleich zum Arbeitsspeicher auch sehr kleiner Zwischenspeicher zu verstehen, der Teile des Inhalts des Hauptspeichers als Kopie enthält. Die Folge aus Registern, Caches, Hauptspeicher und Plattenspeicher nennt man **Speicherhierarchie**.

Cache

Speicherhierarchie

Das einer Speicherhierarchie zu Grunde liegende Prinzip ist das **Lokalitätsprinzip**, dem die Befehle und die Daten eines typischen Programms weitgehend gehorchen. Man unterscheidet die zeitliche Lokalität (*Temporal Locality*) von der räumlichen Lokalität (*Spatial Locality*). Die Erste bedeutet, dass auf dasselbe Code- oder Datenwort während einer Programmausführung mehrfach zugegriffen wird. Räumliche Lokalität bedeutet, dass im Verlaufe einer Programmausführung auch die benachbarten Code- oder Datenwörter benötigt werden.

zeitliche und räumliche Lokalität

Bezogen auf eine Codefolge heißt dies, dass auf einen Befehl meist der durch den Befehlszähler adressierte nächste Befehl oder ein befehlszählerrelativ adres-

Code-Lokalität

[1] Das sind Speicherzellen, die die Information als Ladungsträger in winzigen Kondensatoren speichern.

sierter „kurzer“ Sprung folgt (räumliche Lokalität) und darüber hinaus die Anwendung von Schleifen zur vielfachen Ausführung derselben Codefolge führt (zeitliche Lokalität). Die gesamte Sprungvorhersage würde ohne zeitliche Lokalität in der Programmausführung genauso sinnlos sein wie die Verwendung von Code-Cache-Speichern.

Daten-Lokalität

Auf Daten bezogen, bedeutet zeitliche Lokalität, dass auf ein Datenwort mehrfach zugegriffen wird, und räumliche Lokalität, dass darüber hinaus auch im Speicher benachbarte Datenwörter verwendet werden. Daten, auf die in wenigen Takten wieder zugegriffen wird, werden vom Compiler in Registern bereitgehalten (zeitliche Lokalität). Der Daten-Cache-Speicher nutzt sowohl zeitliche Lokalität – er verdrängt einmal geholte Daten erst wieder, wenn sie durch neuere Zugriffe ersetzt werden müssen – als auch räumliche Lokalität, da nach einem Cache-Fehlzugriff nicht nur das 32- oder 64-Bit-Datenwort, sondern der gesamte, meist 32 Byte große Block, der das Datenwort enthält, im Cache-Speicher bereitgestellt wird.

Es macht deshalb Sinn, räumliche und zeitliche Lokalität zu nutzen, um die Befehle und Daten, auf die wahrscheinlich als nächstes zugegriffen werden muss, „nahe“ am Prozessor zu platzieren und solche Befehle und Daten, die wahrscheinlich in nächster Zeit nicht benötigt werden, auf „entfernteren“ Speichermedien abzulegen – nahe beim Prozessor bedeutet dabei, auf kleinen, schnellen und häufig teuren Speichermedien, entfernt vom Prozessor heißt hingegen, auf großen, billigen Massenspeichern (wie z.B. einer Festplatte).

Die bei heutigen Hochleistungsrechnern realisierte *Speicherhierarchie* setzt sich häufig – nach absteigenden Zugriffsgeschwindigkeiten und aufsteigenden Speicherkapazitäten geordnet – aus Registern, Primär-Cache-Speicher, Sekundär-Cache-Speicher, Hauptspeicher und Sekundärspeicher zusammen.

Abbildung 3.1 gibt diese Speicherhierarchie mit einigen beispielhaften Zugriffszeiten und Speicherkapazitäten wieder. Speichermedien in höheren Ebenen der Speicherhierarchie sind kleiner, schneller, aber auch pro Byte teurer als solche in den tiefer gelegenen Ebenen. Von einer tieferen zu einer höheren und damit prozessornäheren Speicherebene werden Speicherbereiche *auf Anforderung* übertragen, z.B. nach einem Ladebefehl, der einen Cache-Fehlzugriff auslöst. Meist sind die Speicherwörter in den höheren Ebenen Kopien der Speicherwörter in den unteren Ebenen, sodass sich eine hohe Redundanz ergibt. Eine grundlegende Frage der Speicherorganisation ist, ob und wann bei einem Schreibzugriff auf einen Speicher höherer Ebene auch eine Modifikation des Speicherworts in allen Speichern der tieferen Ebenen stattfindet. Ein solches sofortiges „Durchschreiben“ erhält zwar die Konsistenz in allen Kopien, d.h., ein Datum besitzt in allen Stufen der Speicherhierarchie denselben Wert. Das Durchschreiben ist aber nicht immer effizient implementierbar.

Eine Technik, um die Zugriffsgeschwindigkeit auf den Hauptspeicher stärker an die Verarbeitungsgeschwindigkeit des Prozessors anzupassen, ist, den Hauptspeicher in n so genannte **Speicherbänke** $M_0,...,M_{n-1}$ zu unterteilen und jede Speicherbank mit einer eigenen Adressierlogik zu versehen: die **Speicherverschränkung**(*Memory Interleaving*). Falls nun k ($k > n$) sequenziell hintereinander ablaufende Befehle k fortlaufende, physikalische Speicherplätze mit den Adressen $A_0,...,A_{k-1}$ benötigen (typische Beispiele sind Vektor- und Matrixope-

Speicherverschränkung

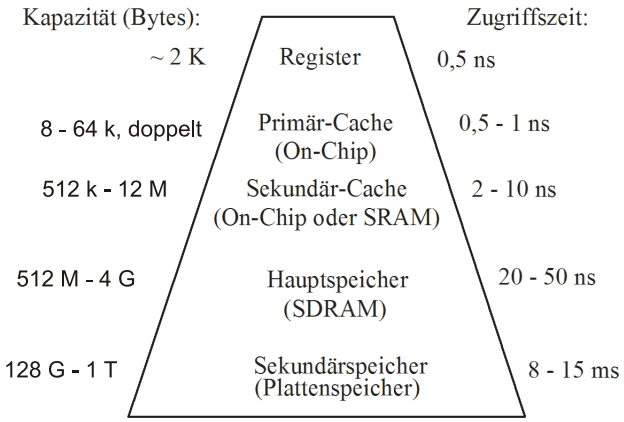


Abbildung 3.1: Die Speicherhierarchie modernen PCs.

rationen), werden die einzelnen Speicherplätze nach der folgenden **Verschränkungsregel** (*Interleaving Rule*) auf die einzelnen Speicherbänke verteilt:

A_i wird genau dann in Speicherbank M_j abgelegt, wenn $j = i \bmod n$ gilt.

Auf diese Weise werden die Adressen A_0, A_n, A_{2n}, ... der Speicherbank M_0, die Adressen A_1, A_{n+1}, A_{2n+1}, ... der Speicherbank M_1 etc. zugeteilt.

Wie gesagt, heißt diese Technik Speicherverschränkung, und die Verteilung auf n Speicherbänke nennt man ***n*-fache Verschränkung**. Der Zugriff auf die Speicherplätze kann nun ebenfalls verschränkt, das heißt zeitlich überlappt, geschehen, sodass bei der n-fachen Verschränkung in ähnlicher Weise wie bei einer Verarbeitungs-Pipeline nach einer gewissen Anlaufzeit in jedem Speicherzyklus n Speicherwörter geliefert werden können.

Eine Speicherverschränkung kann auf jeder Speicherhierarchieebene angewandt werden, insbesondere auch beim Hauptspeicher und bei Cache-Speichern.

3.1.2 Register und Registerfenster

Register stellen die oberste Ebene der Speicherhierarchie dar. Ihre Inhalte können in einem einzigen Prozessortakt in die Pipeline-Register (*Latches*) der Ausführungseinheiten geladen werden. Da die Register im Befehl direkt selektiert bzw. adressiert und der Registersatz auf dem Prozessor-Chip untergebracht und mit vielen Ein- und Ausgabekanälen versehen sein müssen, ist die Anzahl der Register stark beschränkt. Beim Intel IA-32-Registermodell sind es z.B. acht, bei den meisten RISC-Prozessoren 32 oder beim Intel IA-64-Registermodell 128 allgemeine Register. Dazu kommen nochmals ebenso viele Gleitkommaregister sowie zusätzliche Multimediaregister.

Je mehr Register vorhanden sind, desto größer wird der Zeitaufwand für das Sichern der Register beim Unterprogrammaufruf, bei Unterbrechungen und bei einem Betriebssystem-Kontextwechsel bzw. für das Wiederherstellen des Registerzustandes bei Rückkehr aus einem Unterprogramm, einer Unterbrechung oder einem Kontextwechsel. Jeder Aufruf einer Prozedur oder Rücksprung aus einer Prozedur – Aktionen, die sehr häufig auftreten – ändert die lokale Umge-

bung eines Programmablaufes. Bei jedem Unterprogrammaufruf muss der gesamte Registersatz bzw. ein Teil davon gesichert werden, sodass die Daten im Anschluss an die Unterprogrammbearbeitung vom aufrufenden Unterprogramm wieder verwendet werden können. Hinzu kommt die Übergabe von Parametern.

Die im Folgenden beschrieben Lösung dieses Problems basiert auf zwei Feststellungen: Ein Unterprogramm besitzt typischerweise nur wenige Eingangsparameter und lokale Variablen. Ferner ist die Schachtelungstiefe der Unterprogrammaufrufe in der Regel relativ klein. Um diese Eigenschaften zu nutzen,

Registerfenster

werden mehrere kleine Registergruppen, die **Registerfenster** genannt werden, benötigt, wobei jede dieser Gruppen einem Unterprogramm in der momentanen Schachtelungshierarchie zugeordnet ist. Bei einem Unterprogrammaufruf wird auf ein neues Registerfenster umgeschaltet, anstatt die Register im Speicher (Cache oder Hauptspeicher) zu sichern. Die Fenster von aufrufendem und aufgerufenem Unterprogramm überlappen sich, um die Übergabe von Parametern zu ermöglichen (s. Abbildung 3.2). Zu jedem Zeitpunkt ist nur ein Registerfenster sichtbar und adressierbar, so als ob nur ein kleiner Registersatz vorhanden wäre.

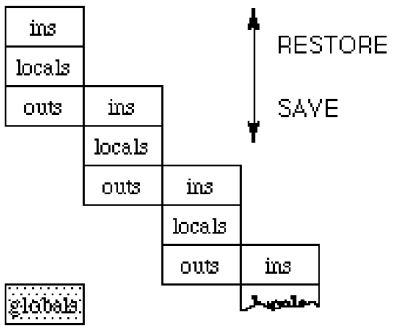


Abbildung 3.2: Überlappende Registerfenster.

Durch eine Registerorganisation mit Registerfenstern kann die Anzahl der Speicherzugriffe und der verbundene Zeitaufwand für das Sichern des Registersatzes bei Prozeduraufrufen verringert werden. Dieses zunächst beim Berkeley RISC eingeführte Verfahren wird im Folgenden am Beispiel der Registerorganisation beim SPARC-Prozessor der Firma SUN gezeigt. Es findet sich auch bei den superskalaren SPARC-Nachfolgern SuperSPARC und UltraSPARC in der Organisation der allgemeinen Register sowie in modifizierter Form beim IA-64-Registersatz des Itanium wieder.

Ein Registerfenster der SPARC-Architektur ist in drei Bereiche aufgeteilt und besteht aus den folgenden 24 Registern: aus 8 *In*-Registern (*Ins*), aus 8 lokalen Registern (*Locals*) und aus 8 *Out*-Registern (*Outs*)[2]. Die lokalen Register sind nur jeweils einer Unterprogrammaktivierung zugänglich. Die In-Register beinhalten sowohl die Parameter, die von dem aufrufenden (Unter-)Programm übergeben wurden, als auch die Parameter, die als Ergebnis des aufgerufenen

[2] Bei anderen RISC-Architekturen können die drei Bereiche auch von variabler Größe sein.

Unterprogramms zurückgegeben werden. Die Out-Register sind mit den In-Registern der nächsthöheren Schachtelungsebene identisch, sodass diese Überlappung eine Parameterübergabe erlaubt, ohne Daten physisch verschieben zu müssen.

Um jede mögliche Schachtelungstiefe handhaben zu können, müsste die Anzahl der Registerfenster unbegrenzt sein. Untersuchungen zeigen jedoch, dass etwa 8 bis 10 Registerfenster ausreichen. Sollte die Schachtelungstiefe die Anzahl der Registerfenster dennoch überschreiten, werden die ältesten Daten zwischenzeitlich im Hauptspeicher abgelegt und die frei werdenden Register erneut verwendet.

Daraus ergibt sich eine Umlaufspeicher-Organisation, wie sie in Abbildung 3.3 am Beispiel eines SPARC-Prozessors mit acht Registerfenstern gezeigt ist. Die Registerfenster sind hier mit w0 bis w7 bezeichnet. Das augenblicklich aktive Registerfenster wird über den CWP (*Current Window Pointer*) referenziert, der in einem Feld des Prozessorstatusworts untergebracht ist.

Der typische Ablauf bei einem Unterprogrammaufruf sieht folgendermaßen aus: Zunächst schreibt das aufrufende Programm die Parameter für das Unterprogramm in seine Out-Register. Dann wird mit einem CALL-Befehl in das Unterprogramm gesprungen. Am Anfang des Unterprogramms wird ein SAVE-Befehl ausgeführt, mit dem auf das nächste Fenster weitergeschaltet wird. Dabei wird der CWP dekrementiert. Im neuen Fenster können jetzt die Parameter aus den In-Registern gelesen und bearbeitet werden. Das Ergebnis bzw. die Ergebnisse der Berechnungen werden am Ende des Unterprogramms wieder in die In-Register geschrieben. Dann wird mit einem RESTORE-Befehl wieder auf das vorherige Fenster zurückgeschaltet (d.h. der CWP wird inkrementiert) und mit dem RET-Befehl in das aufrufende Programm zurückgesprungen. Dort können die Ergebnisse aus den Out-Registern abgeholt werden.[3]

Da der Umlaufspeicher für Registerfenster nur eine begrenzte Größe hat, wird ein Mechanismus benötigt, der einen **Fensterüberlauf** (*Window Overflow*) erkennen und bearbeiten kann. Ein Fensterüberlauf tritt auf, wenn mit einem SAVE-Befehl auf ein Registerfenster umgeschaltet wird, das mit einem bereits verwendeten Registerfenster identisch ist. In diesem Fall muss das zu überschreibende Fenster im Hauptspeicher gesichert werden.

Ein **Fensterunterlauf** (*Window Underflow*) tritt auf, wenn genau ein Registerfenster belegt ist und mit einem RESTORE-Befehl auf das vorher verwendete Fenster zurückgeschaltet wird. In diesem Fall muss das vorherige Fenster aus dem Hauptspeicher nachgeladen werden.

Für die Erkennung eines Fensterüberlaufs bzw. eines Fensterunterlaufs wird beim SPARC das WIM-Register (*Window-Invalid Mask*) verwendet, in dem jedes Registerfenster durch Setzen des entsprechenden Bits als ungültig (*invalid*) markiert werden kann. Wenn der CWP durch SAVE- bzw. RESTORE-Befehle ein Fenster erreicht, das als ungültig markiert ist, wird ein *Trap* ausgelöst. Die zugehörigen Trap-Routinen sind Teil des Betriebssystems und haben die Aufgabe, bei einem Fensterüberlauf ein Fenster auszulagern und bei einem Fen-

[3] Man beachte, dass es beim SPARC für den Unterprogrammaufruf und das Weiterschalten des Fensters zwei verschiedene Befehle CALL und SAVE gibt. Beim Berkeley RISC-Prozessor dagegen sind diese beiden Befehle zu einem einzigen Befehl zusammengefasst.

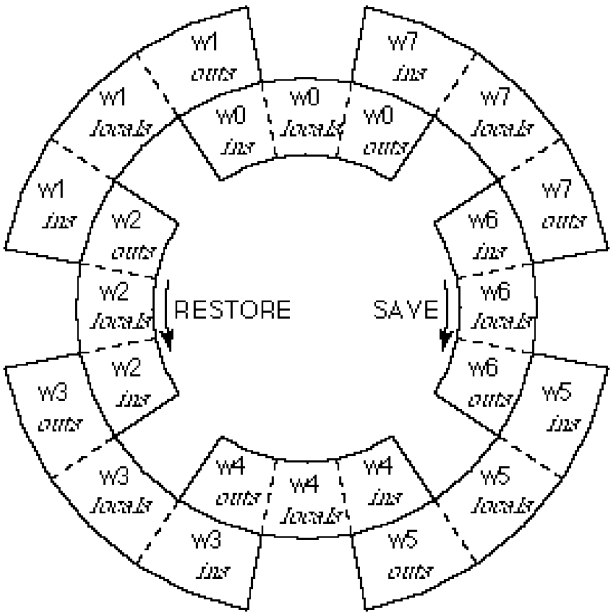


Abbildung 3.3: Umlaufspeicher für überlappende Registerfenster.

sterunterlauf das vorherige Fenster wieder zurückzuholen. Es genügt dabei, nur die beiden Registerbereiche *Ins* und *Locals* zwischenzuspeichern. In den Trap-Routinen wird außerdem das WIM-Register jeweils an die veränderte Situation angepasst.

Man erkennt, dass ein Umlaufspeicher mit n Registerfenstern nur $n-1$ verschachtelte Unterprogrammaufrufe behandeln kann. Es hat sich jedoch gezeigt, dass bereits eine relativ kleine Anzahl von Registerfenstern für eine effiziente Prozedurbehandlung ausreicht. Der Berkeley-RISC-Prozessor verwendet acht Fenster mit jeweils 16 Registern. Untersuchungen haben gezeigt, dass damit in nur einem Prozent aller Unterprogrammaufrufe ein Hauptspeicherzugriff zur Rettung eines Fensters erforderlich ist.

Ein zu großes Registerfenster kann sich sogar negativ auswirken. Die Adressdecodierung erfordert für eine größere Registermenge einen höheren Hardware-Aufwand und kann eventuell mehr Zeit in Anspruch nehmen. In einer Multitasking-Umgebung, bei der der Programmablauf oft zwischen einzelnen Prozessen wechselt, wird mehr Zeit benötigt, um ein großes Registerfenster im Hauptspeicher zu sichern. Die Frage der optimalen Anzahl an Registern ist also keineswegs trivial und hängt sicherlich auch vom Anwendungsgebiet ab.

Die Verwendung von Registerfenstern ermöglicht zwar eine effiziente Behandlung von lokalen Variablen, es werden dennoch auch globale Variablen benötigt, auf die mehrere Unterprogramme Zugriff haben müssen. Dafür sind bei der SPARC-Architektur die Register R0 bis R7 als globaler Registerbereich vorgesehen, auf den alle Unterprogramme jederzeit Zugriff besitzen. Die Register des aktuellen Registerfensters werden mit R8 bis R31 bezeichnet. Der Compiler muss entscheiden, welche Variablen den einzelnen Registerbereichen zugewiesen werden sollen.

Selbsttestaufgabe 3.1 *Eine vereinfachte RISC-Maschine verfüge über acht globale Register R0–R7 und über achtzig weitere Register als Umlaufspeicher für die überlappenden Registerfenster. Ein Fenster umfasse acht In-Register R8–R15, acht lokale Register R16–R23 und acht Out-Register R24–R31. Die Maschine besitze außerdem ein Statusregister SR und ein Register für den Zeiger auf das aktuelle Fenster CWP.*

a) Schreiben Sie für die Funktion $f : N_0 \to N_0$, die die Fibonacci-Zahlen berechnet, wobei

$$f(x) = \begin{cases} x & \text{falls } x \leq 1 \\ f(x-1) + f(x-2) & \text{sonst,} \end{cases}$$

ein Programm für den gegebenen RISC-Prozessor, wobei die globalen Register R0 mit dem Wert 0 und R1 mit dem Wert 1 vorbelegt seien. Der Parameter x soll vom aufrufenden Assemblerprogramm im Register R25 übergeben werden und das Ergebnis soll nach Abarbeitung der zu implementierenden Rechenvorschrift im Register R26 zu lesen sein. Das Register R24 diene zur Aufnahme der Rückkehradresse.

b) Testen Sie Ihr Programm für $f(3)$ und zeichnen Sie sich die Fensterwechsel für diesen Testlauf auf. Trennen Sie die einzelnen Registertypen voneinander und machen Sie sich die Überlappung der einzelnen Registerfenster klar, indem Sie die Fenster in zeitlicher Reihenfolge nebeneinander anordnen und dieselben Register auf dieselbe Höhe ausrichten. Die Inhalte der verwendeten Register sollen in die Fenster eingetragen werden.

c) Für welche Argumentwerte reicht die Gesamtanzahl der Register aus? Was passiert bei größeren Argumentwerten? Welche Techniken könnten Sie sich vorstellen, um auch größere Rekursionstiefen zu ermöglichen?

3.1.3 Cache-Speicher

3.1.3.1 Grundlegende Definitionen

„Cache“ bedeutet in wörtlicher Übersetzung „Versteck“. Unter einem **Cache** oder **Cache-Speicher** versteht man einen kleinen, schnellen Pufferspeicher, in dem Kopien derjenigen Teile des Hauptspeichers gepuffert werden, auf die aller Wahrscheinlichkeit nach vom Prozessor als nächstes zugegriffen wird. Cache-Speicher

Auf den Cache-Speicher soll der Prozessor fast so schnell wie auf seine Register zugreifen können. Er ist deshalb bei heutigen Mikroprozessoren als so genannter **Primär-Cache** (*Primary Cache*, *First-level Cache*, *L1-Cache*) direkt auf dem Prozessor-Chip (*On-chip Cache*) angelegt oder als **Sekundär-Cache** (*Secondary Cache*, *Second-level Cache*, *L2-Cache*) entweder ebenfalls auf dem Prozessor-Chip oder durch die schnellen und teuren statischen Speicherbausteine (SRAM-Technologie) realisiert. Meist werden heute als Primär-Caches getrennte **Code-** und **Daten-Cache-Speicher** mit jeweils eigenen Speicherverwaltungseinheiten eingesetzt, um die Zugriffe durch die Befehlsholeeinheit und die Lade-/Speichereinheit zu entkoppeln. Die Cache-Speicher besitzen nur einen Bruchteil der Kapazität des mit dynamischen Speicherbausteinen (DRAM-Technologie) aufgebauten Hauptspeichers. Durch die Verwendung von Cache-

Speichern wird die Lücke zwischen der hohen Zugriffsgeschwindigkeit auf Register und dem langsameren Zugriff auf den Hauptspeicher teilweise überbrückt. Hauptzweck eines Cache-Speichers ist es, die mittlere Zugriffszeit auf den Hauptspeicher zu verringern.

Cache-Speicherverwaltung

Die Cache-Speicherverwaltung sorgt dafür, dass der Inhalt des Cache-Speichers in der Regel das Speicherwort enthält, auf das der Prozessor als nächstes zugreift. Die Cache-Speicherverwaltung muss sehr schnell sein und ist deshalb vollständig in Hardware realisiert. Durch eine Hardwaresteuerung (*Cache Controller*) werden automatisch die Speicherwörter in den Cache kopiert, auf die der Prozessor zugreift. Die Cache-Speicherverwaltung muss von der für die virtuelle Speicherverwaltung zuständigen Speicherverwaltungseinheit unterschieden werden, wenn auch beide meist ineinander greifen (vgl. Unterabschnitt 3.1.4).

Man spricht von einem **Cache-Treffer** (*Cache Hit*), falls das angeforderte Speicherwort[4] im Cache-Speicher vorhanden ist, und von einem **Cache-Fehlzugriff** (*Cache Miss*), falls das angeforderte Speicherwort nur im Hauptspeicher steht. Im Falle eines Cache-Fehlzugriffs wird ein bestimmter Speicherblock, der das angeforderte Speicherwort umfasst, aus dem Hauptspeicher in den Cache-Speicher geladen. Dies geschieht – unabhängig vom Prozessor – durch die Cache-Speicherverwaltung. Für den Prozessor erscheint es, als ob er immer auf den Hauptspeicher zugreift. Er merkt den Unterschied nur daran, dass der Zugriff im Falle eines Cache-Fehlzugriffs länger dauert. Für das Anwenderprogramm bleibt die Verwendung eines Cache-Speichers transparent.

Die **Cache-Zugriffszeit** t_{hit} bei einem Treffer ist die Anzahl der Takte, die benötigt wird, um ein Speicherwort im Cache zu identifizieren, die Verfügbarkeit und Gültigkeit zu prüfen und das Speicherwort der nachfolgenden Pipeline-Stufe zur Verfügung zu stellen.

Die **Trefferrate** (*Hit Rate*) ist der Prozentsatz der Treffer beim Cache-Zugriff bezogen auf alle Cache-Zugriffe. Die **Fehlzugriffsrate** (*Miss Rate*) ist der Prozentsatz der Cache-Fehlzugriffe bezogen auf alle Cache-Zugriffe.

Der **Fehlzugriffsaufwand** (*Miss Penalty*) t_{miss} ist die Zeit, die benötigt wird, um einen Speicherblock von einer tiefer gelegenen Hierarchiestufe in den Cache zu laden und das Speicherwort dem Prozessor zur Verfügung zu stellen.

Die **Zugriffszeit** (*Access Time*) t_{access} zur unteren Hierarchiestufe ist abhängig von der Latenz des Zugriffs auf der unteren Hierarchiestufe.

Die **Übertragungszeit** (*Transfer Time*) $t_{transfer}$ ist die Zeit, um den Block zu übertragen, und hängt von der Übertragungsbandbreite und der Blocklänge ab.

Die **mittlere Zugriffszeit** (*Average Memory Access Time*) $t_{average-access}$ ist definiert als:

$t_{average-access} = HitRate * t_{hit} + MissRate * t_{miss}$

= Trefferrate ∗ Cache-Zugriffszeit + Fehlzugriffsrate ∗ Fehlzugriffsaufwand

Diese Zeit wird wahlweise in ns oder Prozessortakten angegeben.

[4] Im Folgenden wird der Einfachheit halber von „Speicherwort“ gesprochen, wenn allgemein ein Speicherzugriff des Prozessors gemeint ist. Der Zugriff kann in gleicher Weise ein in ein Register zu ladendes oder aus einem Register zu speicherndes Datenwort als auch einen bzw. im Falle der Superskalarprozessoren sogar gleich mehrere aufeinander folgende Befehle betreffen.

Je höher die Trefferquote, desto mehr verringert sich die mittlere Zugriffszeit. Dadurch beschleunigt sich sowohl der Zugriff auf den Programmcode als auch der Zugriff auf die Daten. Dies gilt insbesondere für Programmschleifen, die häufig vollständig im Cache stehen.

3.1.3.2 Grundlegende Techniken

Bei einem Cache-Speicher versteht man unter einem **Blockrahmen** (*Block Frame*) eine Anzahl von Speicherplätzen, dazu ein Adressetikett und Statusbits. Ein **Cache-Block** (*Cache Block*, *Cache Line*) ist der Speicherbereich, der in einen Blockrahmen passt. Dies ist gleichzeitig auch der Speicherbereich, der auf einmal zwischen Cache- und Hauptspeicher übertragen wird. Unter **Blocklänge** (*Block Size*, *Line Size*) versteht man den verfügbaren Speicherplatz innerhalb eines Blockrahmens. Typische Blocklängen sind 16 oder 32 Byte.

Das **Adressetikett** (*Address Tag*, *Cache Tag*), das auch einfach ***Tag*** genannt wird, enthält einen Teil der Blockadresse, also einen Teil der Speicheradresse des ersten Speicherworts der aktuell im Blockrahmen gespeicherten Speicherwörter. Zu den Statusbits gehört einerseits das *Valid Bit*, das aussagt, ob der Cache-Block „gültig“ bzw. „ungültig“ ist, und das *Dirty Bit*, das anzeigt, ob (wenigstens) ein Wort des Cache-Blocks geändert wurde.

Die Menge der Blockrahmen eines Cache-Speichers ist in **Sätze** (*Sets*) unterteilt. Die Anzahl der Blockrahmen in einem Satz wird als **Assoziativität** (*Associativity, Degree of Associativity*, *Set Size*) bezeichnet. Jeder Block kann nur in einem bestimmten Satz, aber dort in einem beliebigen Blockrahmen gespeichert werden. Die Gesamtzahl c der Blockrahmen in einem Cache-Speicher entspricht immer dem Produkt aus der Anzahl s der Sätze und der Assoziativität n, also $c = s \cdot n$.

Platzierung eines Blocks

Die erste der drei wesentlichen Entscheidungen, die beim Entwurf einer (Cache-)Speicherhierarchie getroffen werden müssen, betrifft die Platzierung eines Blocks: Hier besteht die Wahl zwischen vollassoziativ, satzassoziativ oder direkt-abgebildet. Ein Cache-Speicher mit c Blockrahmen heißt

vollassoziativ, satzassoziativ und direkt-abgebildet

- **vollassoziativ** (*fully-associative*), falls er nur aus einem einzigen Satz besteht ($s = 1$, $n = c$),
- **direkt-abgebildet** (*direct-mapped*), falls jeder Satz nur einen einzigen Blockrahmen enthält ($n = 1$, $s = c$) und
- ***n*-fach satzassoziativ** (*n-way set-associative*), falls ($s = c/n$).

Abbildung 3.4 demonstriert diese Unterscheidung am Beispiel eines Cache-Speichers mit einer Kapazität von acht Cache-Blöcken ($c = 8$). Unter Annahme einer Abbildungsfunktion „*Blockadresse modulo Anzahl der Sätze*“ kann ein Block mit der Adresse 12 in einem der grau markierten Blockrahmen gespeichert werden. In Abbildung 3.4 sind an den linken Seiten jeweils die Satznummern angegeben. Im Falle des vollassoziativen Cache-Speichers gibt es nur einen Satz und der Block kann an einer beliebigen Stelle abgelegt werden. Im Falle der zweifach satzassoziativen Verwaltung gibt es vier Sätze mit je zwei

Blockrahmen. Der Cache-Block mit Adresse 12 muss also in einem der beiden Blockrahmen des Satzes 0 ($0 = 12 \bmod 4$) abgelegt werden. Im Falle der direkt-abgebildeten Organisation gibt es 8 Sätze mit je einem Blockrahmen. Der Block muss also im Satz 4 ($4 = 12 \bmod 8$) gespeichert werden.

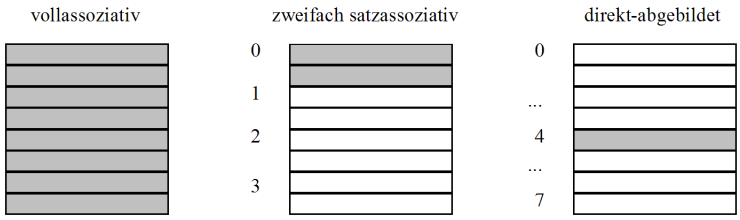


Abbildung 3.4: Vollassoziative, satzassoziative und direkt-abgebildete Cache-Speicher.

Der prinzipielle Ablauf einer direkt-abgebildeten Cache-Speicherverwaltung ist in Abbildung 3.5 dargestellt und wird im Folgenden ausführlich beschrieben.

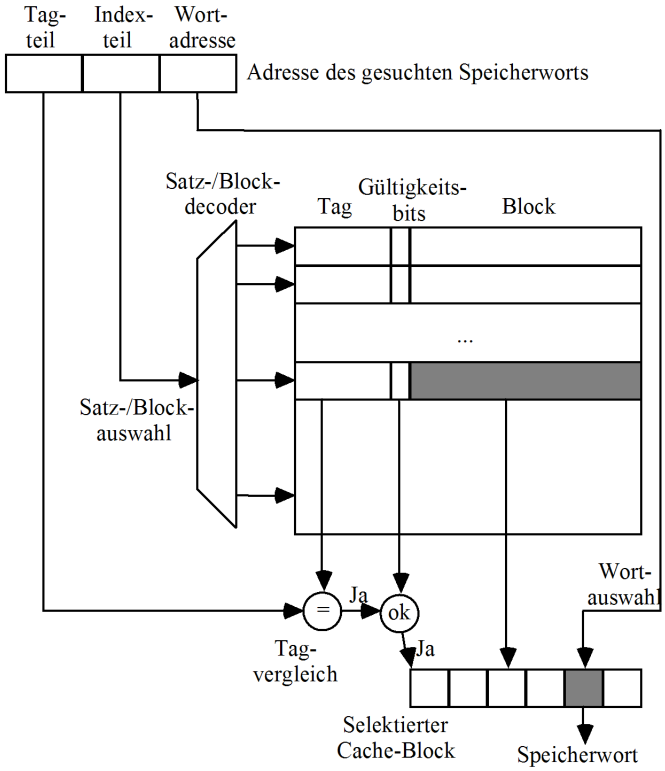


Abbildung 3.5: Direkt-abgebildeter Cache-Speicher

Die Adresse gliedert sich in einen *Tag*-Teil, einen Indexteil und eine Wortadresse. Tag- und Indexteil der Adresse ergeben die Blockadresse, während die Wortadresse das Speicherwort innerhalb des Blocks identifiziert.

Jeder Cache-Block wird durch einen Tag identifiziert. Die Wortadresse bezeichnet das Speicherwort innerhalb eines Cache-Blocks, ist also von der Block-

länge und der Speicheradressierbarkeit abhängig. Im Falle eines byteadressierbaren Speichers und einer Blockgröße von 32 Bytes benötigt die Wortadresse fünf Bits. Der Indexteil wird benötigt, um in dem direkt-abgebildeten Cache-Speicher[5] den Satz ausfindig zu machen. Seine Länge ist also von der Anzahl der Sätze im Cache-Speicher abhängig. So benötigt z.B. unter Annahme von 128 Cache-Sätzen der Indexteil sieben Bits der Speicherwortadresse. Der Rest der Adresse wird zum Tag-Teil, d.h., die Tag-Länge berechnet sich beim hier betrachteten direkt-abgebildeten Cache als Differenz der Adresslänge minus der Index- und Wortadresslänge.

Allgemein hängt die Tag-Länge im Cache-Speicher entscheidend von der Blocklänge, der Assoziativität und der Größe des Cache-Speichers ab. Bei gleich bleibender Cache-Größe verringert eine höhere Assoziativität die Indexlänge und vergrößert die Tag-Länge. Genauer führt eine Verdopplung der Assoziativität zu einer Verringerung der Indexlänge und eine Verlängerung der Tag-Länge um jeweils ein Bit.

Jeder Blockrahmen besteht aus dem Tag, d.h. dem Adressteil des enthaltenen Blocks, einigen Statusbits und dem gespeicherten Cache-Block. Der Indexteil x der angelegten Adresse wird einer Abbildungsfunktion f unterworfen und adressiert mit Hilfe des Satzdecoders einen Satz. Die meist verwendete Abbildungsfunktion ist $f(x) = x \bmod s$, wobei s eine Zweierpotenz ist. Ein Vergleicher überprüft die Gleichheit des Tag-Teils der angelegten Adresse und des Tags, der im selektierten Blockrahmen gespeichert ist. Bei Gleichheit wird noch der Status überprüft. Ist auch dieser in Ordnung, so haben wir einen Cache-Treffer und es wird anhand der Wortadresse das gesuchte Speicherwort aus dem Block ausgewählt. Im Falle eines Cache-Fehlzugriffs wird der Cache-Block verdrängt, der in dem ausgewählten Satz vorhanden ist, und durch den entsprechenden Block aus der nächsten Speicherhierarchie-Stufe ersetzt.

Der direkt-abgebildete Cache-Speicher weist folgende Eigenschaften auf:

- Die Hardwarerealisierung ist einfach. Es wird für alle Sätze nur ein gemeinsamer Vergleicher und pro Satz nur ein Tag-Feld benötigt.
- Der Zugriff erfolgt schnell, weil auf das Tag-Feld parallel mit dem Block zugegriffen werden kann.
- Auch wenn an anderer Stelle im Cache noch Platz ist, erfolgt wegen der direkten Zuordnung eine Ersetzung, sodass die Speicherkapazität oft schlecht ausgenutzt wird.
- Das Ersetzungsverfahren ist einfacher als bei einem n-fach satzassoziativen Cache-Speicher, da bei einem direkt-abgebildeten Cache-Speicher jeder Satz nur *einen* Cache-Block enthält.
- Bei einem abwechselnden Zugriff auf Cache-Blöcke, deren Adressen den gleichen Indexteil haben, erfolgt laufendes Überschreiben des gerade geladenen Blocks. Es kommt zum **Cache-Flattern** (*Thrashing*).[6] Cache-Flattern

[5] Weiter unten wird gezeigt, dass seine Funktion in einem n-fach satzassoziativen Cache ähnlich ist.

Der prinzipielle Ablauf einer n-fach satzassoziativen Cache-Speicherverwaltung ist vereinfacht in Abbildung 3.6 dargestellt und wird im Folgenden ausführlich beschrieben.

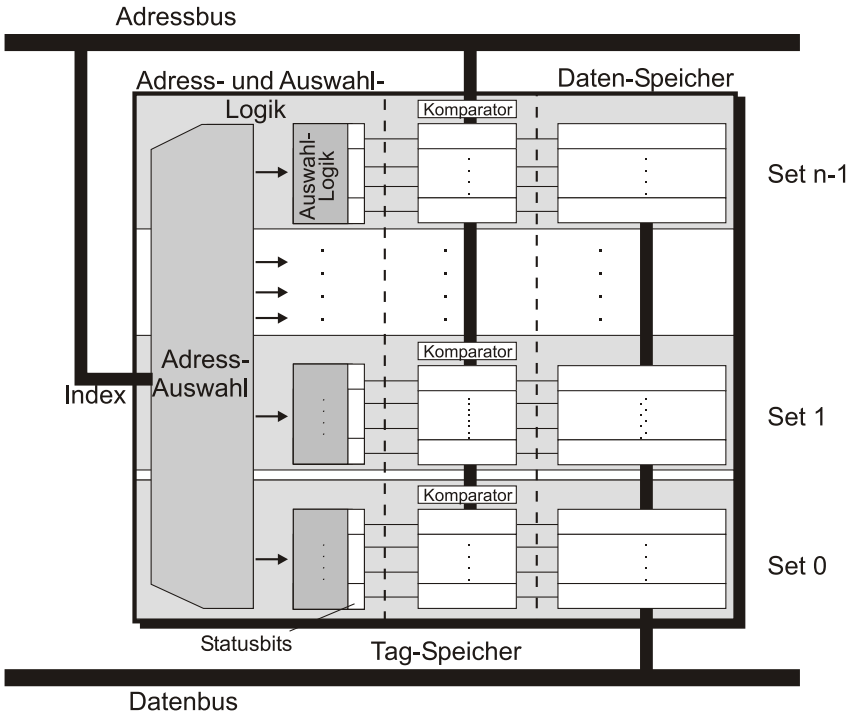


Abbildung 3.6: n-fach satzassoziative Cache-Speicherverwaltung

Die Adresse des gesuchten Speicherworts ist durch eine Blockadresse, bestehend aus Tag- und Indexteil, sowie die Offset-Adresse innerhalb des Blocks gegeben (vgl. Abbildung 3.5). Die letztere adressiert das Speicherwort innerhalb des Cache-Blocks. Der Indexteil der Blockadresse x wird der Abbildungsfunktion f unterworfen und adressiert mit Hilfe des Satzdecoders (Adress-Auswahl) einen Satz. Die Blockrahmen innerhalb des selektierten Satzes werden nun nach dem gewünschten Block durchsucht. Dazu wird der Tag-Teil der Blockadresse mit den Cache-Tags aller Blöcke des selektierten Satzes durch einen Komparartor simultan verglichen. Wird der gesuchte Block gefunden, haben wir einen Cache-Treffer. In diesem Fall wird das verlangte Speicherwort gemäß der Block-Offset-Adresse vom Speicherwortselektor aus dem Block herausgesucht. (Der Vorgang der Wortauswahl wird in Abbildung 3.5 gezeigt, in Abbildung 3.6 zur Vereinfachung jedoch weggelassen.) Falls festgestellt wird, dass der gesuchte Block in dem Satz nicht vorhanden ist, ergibt sich ein Cache-Fehlzugriff. Im Falle eines Cache-Fehlzugriffs wird automatisch ein Cache-Block nach der Ersetzungsstrategie durch den durch f(x) selektierten Block ersetzt. Bei Cache-Speichern mit geringer Assoziativität müssen nur wenige Blockrahmen pro Satz beim Zugriff durchsucht werden, was die Implementierung vereinfacht.

[6]Davon ist zwar u.U. auch ein n-fach satzassoziativer Cache-Speicher betroffen, aber nicht so stark wie der direkt-abgebildete Cache.

Eine weitere Entscheidung betrifft die Auswahl des Blocks, der nach einem Fehlzugriff verdrängt wird, falls kein freier Blockrahmen im Cache mehr zur Verfügung steht. Dieser Block, der nach einem Cache-Fehlzugriff evtl. überschrieben wird, wird durch die **Verdrängungsstrategie** bestimmt. Sie ist in Hardware realisiert und muss sehr schnell reagieren. Ideal wäre die **LRU-Strategie** (*Least Recently Used*), die den Cache-Block verdrängt, auf den am längsten nicht mehr zugegriffen wurde. Meist wird jedoch die einfachere **Pseudo-LRU-Strategie** implementiert, die im Falle eines vierfach satzassoziativen Cache-Speichers mit drei Bits auskommt, jedoch gelegentlich auch den zweitletzten Cache-Block überschreibt. Bei Primär-Cache-Speichern wird heute meist eine zwei- bis vierfach satzassoziative Cache-Speicherorganisation angewandt.

Verdrängungsstrategie

Für vollassoziative Cache-Speicher bietet sich eine **Zufallsstrategie** an, da diese den kleinsten Hardware-Aufwand verursacht. Da vollassoziative Cache-Speicher technisch ab einer gewissen Größe nicht mehr realisierbar sind, ist diese Organisationsform meist nur bei den sehr kleinen Adressübersetzungspuffern (TLB) der virtuellen Speicherverwaltung mit 32 bis 128 Einträgen anzutreffen, die wir im Unterabschnitt 3.1.4 beschreiben werden.

Selbsttestaufgabe 3.2 *Bei einem byteadressierten Rechner seien ein direktabgebildeter Cache, ein 2-fach satzassoziativer Cache und ein vollassoziativer Cache gegeben. Die drei Cachespeicher haben jeweils eine Speicherkapazität von 32 Bytes und werden in Blöcken von je vier Bytes geladen. Die Hauptspeicheradresse umfasst 32 Bits.*

a) Geben Sie für die drei Cache-Speicher an, wie viele Bits zur Verwaltung (Tag- und Zustandsbits) eines Cache-Blocks benötigt werden. Dabei sollen für den Zustand des Cache-Blocks zwei Statusbits verwendet werden (Valid Bit und Dirty Bit).

b) Betrachten Sie die Folge der Lesezugriffe auf die folgenden, in hexadezimaler Schreibweise angegebenen Hauptspeicheradressen:

$46, $09, $44, $B9, $11, $FB, $55, $F9, $5C, $06

Nehmen Sie an, die Caches seien zu Beginn leer. Ermitteln Sie, ob es sich beim Lesezugriff auf die jeweiligen Adressen um einen Treffer (Cache Hit) oder einen Fehlzugriff (Cache Miss) handelt. Falls notwendig, wird die LRU-Ersetzungsstrategie (Least Recently Used) verwendet. Geben Sie die Tag-Zustände des Cache-Speichers nach dem letzten Zugriff an.

Die dritte Entscheidung betrifft die Organisation der Schreibzugriffe des Prozessors auf den Cache-Speicher und die Methode, die Inhalte von Cache- und Hauptspeicher konsistent zu halten, d.h. dafür zu sorgen, dass alle Kopien eines Speicherworts in den Caches und dem Hauptspeicher denselben Wert haben. Hier besteht die Wahl zwischen der Rückschreibe- und der Durchschreibestrategie mit Schreibpuffer.

Organisation der Schreibzugriffe

Bei der **Durchschreibestrategie**(*Write-through Cache*, *Store-through Cache*) wird ein Speicherwort vom Prozessor immer gleichzeitig in den Cache bzw. die Caches und in den Hauptspeicher geschrieben. Der Vorteil ist eine garantierte Konsistenz der Inhalte von Cache- und Hauptspeicher, da der Hauptspeicher immer die zuletzt geschriebene, also gültige Kopie enthält. Das ist besonders dann wichtig, wenn mehrere Verarbeitungselemente (beispielsweise

Durchschreibestrategie

mehrere Prozessoren) auf ein Speicherwort zugreifen wollen, das sich in einem Cache-Speicher befindet.

Der Nachteil ist, dass der sehr schnelle Prozessor bei Schreibzugriffen mit dem langsamen Hauptspeicher synchronisiert werden muss und somit an Verarbeitungsgeschwindigkeit verliert. Um diesen Nachteil zu mildern, wird in Verbindung mit einer Durchschreibestrategie ein kleiner **Schreibpuffer** (*Write Buffer*) verwendet, der das zu schreibenden Speicherwort temporär aufnimmt und sukzessive in den Hauptspeicher überträgt, während der Prozessor parallel dazu mit weiteren Operationen fortfährt. Diese Technik wird als **gepufferte Durchschreibestrategie** (*buffered Write-through*) bezeichnet. Selbst in diesem Fall wird aber der Bus zwischen Prozessor und Speicher belegt, was zum Engpass führen kann.

gepufferte Durchschreibestrategie

Bei der **Rückschreibestrategie** (*Store-in Cache*, *Write-back Cache*, *Copy-back Cache*) wird ein Speicherwort nur in den Cache-Speicher geschrieben und ein Zustandsbit, das so genannte *Dirty Bit*, auf „verändert“ gesetzt. Der Hauptspeicher wird nur dann geändert, wenn das *Dirty Bit* gesetzt ist und der gesamte Block durch die Verdrängungsstrategie ersetzt wird. Diese Strategie, die einen höheren Verwaltungsaufwand bei einem Lesezugriff verlangt, benötigt beim Schreibzugriff durch den Prozessor keine Synchronisation mit dem Hauptspeicher. Die Strategie ist komplexer, aber vorteilhaft. Sie wird bei heutigen Rechnern vorwiegend angewandt.

Rückschreibestrategie

Da die mittlere Zugriffszeit von der Fehlzugriffsrate, dem Fehlzugriffsaufwand und der Cache-Zugriffszeit bei einem Treffer abhängt, kann die Verarbeitungsleistung eines Cache-Speichers durch die Verringerung einer oder mehrerer der genannten Parameter verbessert werden, diese Möglichkeiten werden in den folgenden Unterabschnitten beschrieben.

3.1.3.3 Verringern der Fehlzugriffsrate

Die Fehlzugriffsrate hängt entscheidend von einer Analyse der Ursachen für die Fehlzugriffe ab. Diese lassen sich in drei Klassen einteilen:

Ursachen für Fehlzugriffe

- **Erstzugriff** (*compulsory* – obligatorisch): Beim ersten Zugriff auf einen Cache-Block befindet sich dieser noch nicht im Cache-Speicher und muss erstmals geladen werden. Diese Art von Fehlzugriffen lässt sich auch als Kaltstartfehlzugriffe (*Cold Start Misses*) oder Erstbelegungsfehlzugriffe (*First Reference Misses*) bezeichnen. Sie würden sogar in einem unendlich großen Cache-Speicher auftreten und sind unvermeidbar.

- **Kapazität** (*Capacity*): Falls der Cache-Speicher nicht alle benötigten Cache-Blöcke aufnehmen kann, müssen Cache-Blöcke verdrängt und eventuell später wieder geladen werden. Fehlzugriffe wegen mangelnder Cache-Kapazität sind von der Cache-Speicherorganisation unabhängig und würden auch in einem vollassoziativen Cache mit entsprechend beschränkter Größe auftreten.

- **Konflikt** (*Conflict*): Bei einer satzassoziativen oder direkt-abgebildeten Cache-Speicherorganisation können zusätzlich zu den ersten beiden Fehlzugriffsarten Konfliktfehlzugriffe auftreten, da ein Cache-Block potenziell

verdrängt und später wieder geladen wird, falls zu viele Cache-Blöcke auf denselben Satz abgebildet werden. Diese Fehlzugriffe werden auch Kollisionsfehlzugriffe (*Collision Misses*) oder Interferenzfehlzugriffe (*Interference Misses*) genannt. Kollisionsfehlzugriffe treten nur bei direkt-abgebildeten oder satzassoziativen Cache-Speichern beschränkter Größe auf.

Die Gesamtzahl der Fehlzugriffe hängt von den *3 Cs – Compulsory*, *Capacity*, *Conflict* – ab. Die Zahl der Fehlzugriffe, die durch einen Erstzugriff bedingt sind, kann nur durch die Wahl eines größeren Cache-Blocks und durch eine bessere Verteilung der Daten im Speicher durch den Compiler verringert werden. In beiden Fällen geht es darum, räumliche Lokalität auszunutzen und mit einem Speicherzugriff mehr erforderliche Daten bereitzustellen. Fehlzugriffe wegen mangelnder Cache-Speicherkapazität lassen sich nur durch größere Caches verringern. Unter der Annahme eines festen Budgets an Cache-Speicherplatz auf dem Chip kann jedoch die Anzahl der durch Konflikte ausgelösten Fehlzugriffe verringert werden. Die dafür möglichen Maßnahmen werden im Folgenden dargestellt:

- *Größerer Cache-Block*: Größere Cache-Blöcke erhöhen die räumliche Lokalität und können somit die Zahl der Erstzugriffs- als auch der Konfliktfehlzugriffe verringern. Allerdings bedeutet das Laden größerer Cache-Blöcke auch einen größeren Fehlzugriffsaufwand und potenziell wiederum eher Konflikte, da der Cache weniger Cache-Blöcke umfasst. Simulationen zeigten, dass die optimale Blockgröße bei 32 – 128 Bytes liegt.

- *Höhere Assoziativität*: Die Zahl der Konflikte lässt sich durch eine größere Assoziativität entscheidend verringern. Allerdings lässt sich die Assoziativität nicht beliebig steigern, da der Hardware-Aufwand und die Zugriffszeit ansteigen und im Falle eines On-Chip-Cache-Speichers die Taktrate des Prozessors beeinträchtigt wird. Üblich sind heute Assoziativitäten von vier und acht.

- *Pseudo-Assoziativität*: Diese dient dazu, bei einem direkt-abgebildeten Cache-Speicher die Anzahl der Konflikte zu verringern. Der Cache wird in zwei Bereiche geteilt, auf die mit verschiedenen Geschwindigkeiten zugegriffen wird. Im Fall eines Fehlzugriffs im ersten Cache-Bereich wird in zweiten Cache-Bereich gesucht. Falls das Speicherwort dort gefunden wird, so spricht man von einem Pseudo-Treffer.

  Das Verfahren eignet sich nicht gut für Primär-Cache-Speicher, da ein Cache-Zugriff, der eine unterschiedliche Anzahl von Takten (ein oder zwei) benötigt, nur schwer in die Befehls-Pipeline eingebaut werden kann. Die Technik eignet sich jedoch für Sekundär-Cache-Speicher, da dann schon ein Cache-Fehlzugriff in der Prozessor-Pipeline ausgelöst wurde und sich nur die Zahl der Wartetakte erhöht. Die Technik wird bei den Sekundär-Cache-Speichern des MIPS R10000 und in ähnlicher Weise beim UltraSPARC angewandt.

- *Victim Cache*: Im Falle eines direkt-abgebildeten Cache-Speichers tritt häufig das oben beschriebene Cache-Flattern auf, das durch mehrfache Speicherzugriffe auf zwei Cache-Blöcke, die auf denselben Cache-Satz abgebildet werden, bedingt ist. Die Folge ist, dass die Cache-Blöcke ständig zwischen dem Cache-Speicher und der nächsten Speicherhierarchieebene ausgetauscht werden. Die Fehlzugriffsrate kann in diesem Fall durch einen so genannten *Victim Cache* verringert werden. Das ist ein kleiner Pufferspeicher, auf den genauso schnell wie auf den Cache zugegriffen werden kann und der die zuletzt aus dem Cache geworfenen Cache-Blöcke aufnimmt.

- *Vorabladen per Hardware* (*Hardware Prefetching*): Die Hardware lädt nach einem bestimmten Schema Speicherwörter spekulativ in den Cache-Speicher, ohne dass diese durch einen Fehlzugriff angefordert wurden. Spekulatives Laden benötigt, wie alle Arten von Spekulation, freie Ressourcen als Voraussetzung. Vorabladen macht nur dann Sinn, wenn zusätzliche Speicherbandbreite zur Verfügung steht, die durch eine normale Programmausführung nicht ausgenutzt wird. Die einfachste Form des spekulativen Vorabladens besteht darin, bei einem Cache-Fehlzugriff nicht einen, sondern gleich noch den darauf folgenden Cache-Block in den Cache-Speicher zu laden.[7] Das Vorabladen per Hardware wird von vielen heutigen Prozessoren unterstützt. Beispielsweise erkennt der Pentium-4-Prozessor ständig wiederkehrende Zugriffsmuster und führt ein Vorabladen per Hardware durch seine *Hardware-Data-Prefetch*-Einheit durch.

- *Vorabladen per Software* (*Software Data Prefetching*): Die Software lädt durch spezielle Vorabladebefehle (*Cache Prefetch*, *Cache Touch*) Daten spekulativ in den Daten-Cache-Speicher. Das Verfahren wird von vielen Prozessorarchitekturen unterstützt, wie beispielsweise den PowerPC-, SPARC-V9-, MIPS IV- und IA-64-Architekturen. Die Vorabladebefehle stellen eine Form der Software-Spekulation dar und erzeugen keine Ausnahmen. Zu beachten ist weiterhin, dass die Vorabladebefehle Befehle sind, die zusätzlich zum zugehörigen Ladebefehl in der Pipeline ausgeführt werden. Voraussetzung ist deshalb wiederum, dass ansonsten ungenutzte Ressourcen zur Verfügung stehen. Das betrifft neben der Speicherbandbreite auch die Befehlslade-, Zuordnungs- und Ausführungsbandbreite des Prozessors. Für heutige Superskalarprozessoren stellt die Verwendung von Vorabladebefehlen durch den Compiler neben der Sprungspekulation eine weitere Methode dar, um die Prozessorressourcen besser auszulasten.

- *Compileroptimierungen*: Die Befehle und Daten werden vom Compiler so im Speicher angeordnet, dass Konflikte möglichst vermieden werden und die Kapazität besser ausgenutzt wird. Ziel ist es, dass möglichst nur Speicherwörter, die auch benötigt werden, in den Cache-Speicher geladen werden. Durch Hintereinanderanordnung von Daten, auf die nacheinander

[7] Das geschah z.B. bei einem Code-Cache-Fehlzugriff des Alpha-21064-Prozessors, der den zweiten Cache-Block allerdings in einen *Stream Buffer* genannten Pufferspeicher schrieb. Beim nächsten Cache-Fehlzugriff wurde dann auf den Stream Buffer zugegriffen.

zugegriffen wird, lässt sich auch die Anzahl der Fehlzugriffe beim Erstzugriff verringern. Mögliche Techniken dafür sind das Zusammenfassen von Feld-Datenstrukturen (*Arrays*) und die Manipulation der Schleifenanordnung: Beispiele dafür sind das Vertauschen geschachtelter Schleifen oder die Verschmelzung von Schleifen, um auf die Daten in der gleichen Ordnung zuzugreifen, in der sie im Speicher stehen.

3.1.3.4 Verringern des Fehlzugriffsaufwandes

Es gibt verschiedene Techniken, um den Fehlzugriffsaufwand zu verringern. Die Grundidee ist meist, dem Prozessor das zu ladende Speicherwort so schnell wie möglich zur Verfügung zu stellen.

- *Ladezugriffe vor Speicherzugriffen beim Cache-Fehlzugriff*: Im Falle einer Durchschreibestrategie werden die in den Speicher zu schreibenden Cache-Blöcke in einem Schreibpuffer zwischengespeichert. Dabei kann das Laden neuer Cache-Blöcke vor dem Speichern von Cache-Blöcken ausgeführt werden, um die Wartezeit des Prozessors auf die zu ladenden Speicherwörter zu verringern. Um sicherzugehen, dass bei einem vorgezogenen Laden keine Datenabhängigkeit zu einem der passierten Speicherwörter verletzt wird, müssen die Ladeadressen mit den Adressen der zu speichernden Daten verglichen werden. Im Falle einer Übereinstimmung muss die Ladeoperation ihren Wert aus dem Schreibpuffer und nicht aus der nächst tieferen Speicherhierarchieebene erhalten.

  Auch bei Rückschreib-Cache-Speichern wird bei einem Cache-Fehlzugriff der verdrängte (*dirty*) Block zurückgespeichert. Normalerweise würde das Rückschreiben vor dem Holen des neuen Cache-Blocks erfolgen. Um den zu ladenden Wert dem Prozessor so schnell wie möglich zur Verfügung stellen zu können, wird der verdrängte Cache-Block ebenfalls im Schreibpuffer zwischengespeichert und erst der Ladezugriff vor dem anschließenden Rückschreibzugriff auf den Speicher durchgeführt.

- *Zeitkritisches Speicherwort zuerst* (*Critical Word first*): Das Laden eines Cache-Blocks in den Cache kann mehrere Takte dauern. Der Prozessor benötigt jedoch nur das Speicherwort dringend, dessen Ladezugriff den Fehlzugriff ausgelöst hat. Sobald dieses Speicherwort im Cache ankommt, wird es sofort dem Prozessor zur Verfügung gestellt und dieser kann weiterarbeiten (*early Restart*). Darüber hinaus kann das angeforderte Speicherwort auch als erstes vom Speicher zum Cache übertragen werden (*Critical Word first, wrapped Fetch*) und anschließend werden die nicht zeitkritischen, restlichen Speicherwörter des Cache-Blocks übertragen. Beide Verfahren sind nur bei einem großen Cache-Block effizient und bringen keinen Vorteil, wenn auch sofort auf die benachbarten Speicherwörter zugegriffen wird.

- *Nichtblockierender Cache-Speicher* (*non-blocking Cache*, *lockup-free Cache*): Im Falle eines Cache-Fehlzugriffs können nachfolgende Ladeoperationen, die nicht denselben Cache-Block benötigen, auf dem Cache-

Speicher durchgeführt werden, ohne dass auf die Ausführung der den Fehlzugriff auslösenden Lade- oder Speicheroperation gewartet werden muss. Ein Fehlzugriff blockiert damit keine nachfolgenden Ladebefehle. Konsequenterweise kann das Verfahren auch auf mehrere Cache-Fehlzugriffe ausgedehnt werden, also Ladezugriffe bei mehreren ausstehenden Cache-Fehlzugriffen erlauben. Die Komplexität der Cache-Speicherverwaltung wird dadurch erheblich erhöht, da mehrere ausstehende Speicherzugriffe verwaltet werden müssen. Dennoch wird das Verfahren heute allgemein angewandt. Bereits der Pentium-Pro-Prozessor erlaubte beispielsweise vier ausstehende Speicherzugriffe. Probleme ergeben sich jedoch für die Speicherkonsistenz, falls Prozessoren mit nicht-blockierenden Cache-Speichern in Multiprozessorsystemen eingesetzt werden (s. Unterabschnitt 3.1.3.6).

- *Einsatz eines Sekundär-Cache-Speichers* (*L2-Cache*): Wie schon erwähnt, befinden sich auf dem Prozessor-Chip getrennte Code- und Daten-Cache-Speicher als Primär-Caches. Auf diese kann mit der Taktrate des Prozessors zugegriffen werden. Sie sind deshalb mit 8 bis 64 kB verhältnismäßig klein. Durch die Trennung erreicht man einen parallelen Zugriff auf Programm und Daten, wodurch die hohen Anforderungen der heutigen Superskalar-Prozessoren an die Speicherbandbreiten erfüllt werden können. Dabei wird auf dem Chip eine Harvard-Architektur realisiert, bei der Programm und Daten in getrennten Speichern abgelegt werden (s. Kapitel 1). Falls nur ein einziger Cache-Speicher für Code und Daten vorhanden ist, spricht man dagegen von einem einheitlichen Cache-Speicher (*unified Cache*).
  Neben den Primär-Caches befindet sich häufig ein weiterer, heute etwa 256 kB bis 12 MB[8] großer Sekundär-Cache auf dem Prozessor-Chip. Dieser sorgt dafür, dass bei einem Fehlzugriff auf den Primär-Cache die Daten schnell nachgeladen werden können. Beim Austausch von Daten zwischen Hauptspeicher und Sekundär-Cache bzw. Sekundär- und Primär-Cache werden komplette Blöcke oder Teilblöcke in einer Aktion (*Burst*) übertragen.
  Beim Vorhandensein eines Sekundär-Cache-Speichers wird der Primär-Cache meist als Durchschreibespeicher betrieben, sodass alle Daten im Primär-Cache auch im Sekundär-Cache stehen (Inklusionsprinzip). Der Sekundär-Cache arbeitet dagegen in der Regel mit dem Rückschreibeverfahren.
  Der Sekundär-Cache kann als so genannter *Look-aside Cache* parallel zum Hauptspeicher an den Bus angeschlossen werden. Der Prozessor kann so direkt auf Cache und Hauptspeicher zugreifen. Allerdings ist dies nur bei Ein-Prozessorsystemen sinnvoll, da es bei jedem Speicherzugriff zu einem Buszugriff und damit zu einer erheblichen Zusatzbelastung des Speicherbusses kommt.

[8] als gemeinsamer L2-Cache für die Kerne eines Mehrkernprozessors.

3.1.3.5 Verringern der Cache-Zugriffszeit bei einem Treffer

Als weitere Cache-Optimierung kann die Zugriffszeit beim Cache-Treffer verringert werden.

- *Primär-Cache-Speicher auf dem Prozessor-Chip*: Wie schon erwähnt, sollte der Primär-Cache-Speicher auf dem Prozessor-Chip untergebracht und klein sowie einfach genug strukturiert sein, um mit der Geschwindigkeit der Prozessortaktrate zugreifen zu können. Bereits der Alpha-21164-Prozessor hatte beispielsweise je 8 kB Primär-Caches für Befehle und Daten und einen Sekundär-Cache-Speicher von 96 kB auf dem Prozessor-Chip. Ein direkt-abgebildeter Cache ist im Zugriff schneller als ein mehrfach satzassoziativer Cache. Jedoch finden sich heute bis zu achtfach satzassoziative Daten-Caches als Primär-Caches auf dem Prozessor-Chip. Die Primär-Cache-Größen heutiger Prozessoren variieren von 32 kB bei den Intel-Prozessoren bis zu 64 kB bei den AMD-Prozessoren.

- *Virtueller Cache*: Die Zugriffszeit kann auch durch ein Vermeiden von Adressübersetzungen erhöht werden. Der Cache kann mit der virtuellen Adresse oder mit der physikalischen Adresse angesprochen werden. Je nachdem, ob die Adressumsetzung durch eine Speicherverwaltungseinheit mit Adressübersetzungspuffer (TLB, vgl. Unterabschnitt 3.1.4) vor oder hinter dem Cache vorgenommen wird (s. Abbildung 3.7), spricht man von einem virtuell bzw. physikalisch adressierten Cache-Speicher oder kürzer von einem **virtuellen Cache** bzw. einem **physikalischen Cache**.

  Der virtuell adressierte Cache hat den Vorteil, dass auf die Daten schneller zugegriffen werden kann, da bei einem Cache-Treffer keine Adressumsetzung notwendig ist. Jedoch muss bei einem virtuellen Cache bei jedem Prozesswechsel ein Löschen der Cache-Inhalte erfolgen, da ansonsten falsche Treffer geschehen. Die zusätzlichen Kosten sind dann die Zeit für das Leeren des Caches plus die Zeit für die Erstbelegungsfehlzugriffe, da nach einem Prozesswechsel immer mit einem leeren Cache gestartet wird. Eine Lösung dafür ist das Einführen von Prozessidentifikatoren, die zusätzlich zu den Cache-Tags für die Blockselektion verwendet werden und die dafür sorgen, dass nur die zu einem Prozess gehörenden Einträge im TLB gezielt gelöscht werden können.

  Weiterhin verursacht der virtuelle Cache ein Problem beim „Bus-Schnüffeln“ (vgl. Unterabschnitt 3.1.3.7), da die Cache-Tags virtuelle (bzw. logische) und nicht physikalische Adressen sind. Wenn eine Speicherzelle durch einen Ein-/Ausgabevorgang eines anderen Prozesses oder durch einen anderen Prozessor in einem Multiprozessorsystem verändert wird, so merkt der virtuell adressierte Cache diese Änderung nicht ohne Weiteres, da gleiche Daten i.d.R. ungleiche Namen haben. Um zu verhindern, dass der Prozessor mit veralteten Daten arbeitet, wird zusätzlich zu den virtuellen Cache-Tags ein weiterer Satz von physikalischen Cache-Tags für das Bus-Schnüffeln benötigt.

  Eine effiziente Lösung bietet eine Cache-Organisation, bei der der Index aus der logischen, der Tag jedoch aus der physikalischen Adresse gewon-

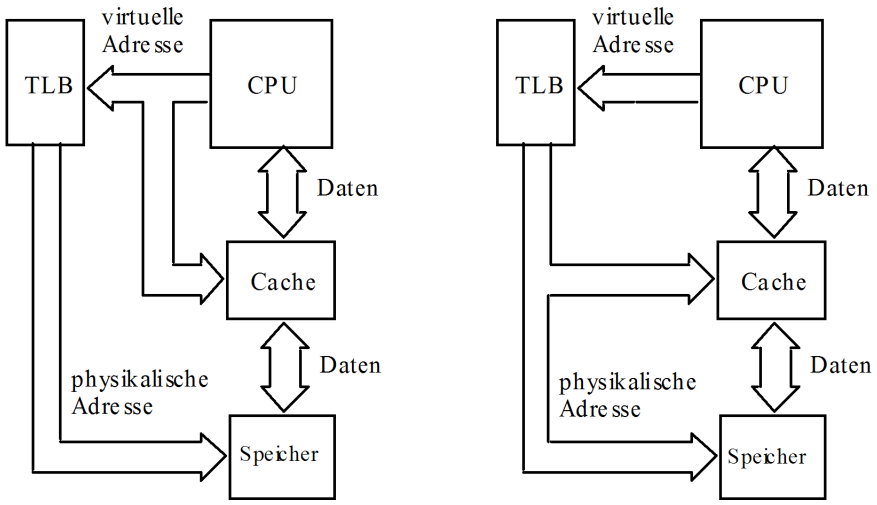


Abbildung 3.7: Zwei Möglichkeiten der Cache-Adressierung: (a) virtueller Cache, (b) physikalischer Cache.

logically indexed physically tagged Cache

nen wird. Diese Lösung wird als *logically indexed and physically tagged Cache* bezeichnet. Sie wurde bereits beim Motorola 88110-Prozessor angewandt. Ihr Vorteil besteht darin, dass TLB- und Cache-Zugriff parallel zueinander durchgeführt werden können. Damit ist dieses Verfahren besonders effizient. Es wird nur ein Takt mehr als für einen Registerzugriff benötigt.

Der Cache-Tag ist eine physikalische Adresse, die durch Übersetzung der höchstwertigen Bits der logischen Adresse mit Hilfe eines TLBs erzielt wird (deshalb: *physically tagged*). Gleichzeitig selektiert der Indexteil der logischen Adresse den Cache-Satz (deshalb: *logically indexed*) und die niederstwertigen Bits der logischen Adresse selektieren das Speicherwort im Satz. Satzauswahl, Speicherwortselektion und TLB-Zugriff geschehen gleichzeitig. Die durch den TLB-Zugriff erzielte physikalische Tag-Adresse wird mit dem Cache-Tag des selektierten Cache-Blocks verglichen, um festzustellen, ob der Zugriff ein Treffer oder Fehlschlag ist.

- *Pipeline-Verarbeitung für Schreibzugriffe auf den Cache*: Die Tag-Prüfung eines Cache-Zugriffs und die Cache-Modifikation durch den vorherigen Schreibzugriff können als separate Pipeline-Stufen implementiert und überlappt zueinander ausgeführt werden. Durch eine solche zusätzliche Pipeline-Verarbeitung wird die Zugriffsgeschwindigkeit bei hohen Prozessortaktraten erhöht.

Selbsttestaufgabe 3.3 *Gegeben seien ein direkt-abgebildeter Cache, ein 4-fach satzassoziativer Cache und ein vollassoziativer Cache. Alle drei Cache-Speicher werden durch 32 Bits adressiert und haben eine Kapazität von 16 Blöcken, wobei ein Block jeweils vier Datenwörter umfasst. Bei den assoziativen Caches wird die LRU-Ersetzungsstrategie verwendet.*

Die folgende Sequenz zeigt die hexadezimalen Adressen der ersten zwanzig Lesezugriffe eines Programms zur Multiplikation zweier Matrizen mit je 10000 Zeilen/Spalten:

1F296FFA, 378CF121, 1F296FFB, 378D1831, 1F296FFC, 378D3F41, 1F296FFD, 378D6651, 1F296FFE, 378D8D61, 1F296FFF, 378DB471, 1F297000, 378DDB81, 1F297001, 378E0291, 1F297002, 378E29A1, 1F297003, 378E50B1

a) Geben Sie unter Berücksichtigung der jeweiligen Cache-Organisationsform an, wie viele Adressbits für den Tag- bzw. den Index-Teil benötigt werden. Wie breit ist die Wortadresse?

b) Bestimmen Sie für alle drei Cache-Typen, bei welchen Zugriffen auf die oben genannten Adressen es sich um Cache-Treffer handelt. Führen Sie dazu Buch über die Belegungen der einzelnen Cache-Blöcke. Nehmen Sie an, dass die Caches zu Beginn leer sind. Wie hoch sind die Trefferraten?

3.1.3.6 Kohärenz und Konsistenz

Falls in einem Multiprozessorsystem mehrere Prozessoren mit jeweils eigenen Cache-Speichern unabhängig voneinander auf Speicherwörter des Hauptspeichers zugreifen, können Gültigkeitsprobleme entstehen. Mehrere Kopien des gleichen Speicherworts können sich in den Cache-Speichern der verschiedenen Prozessoren befinden und müssen miteinander in Einklang gebracht werden.

Eine Cache-Speicherverwaltung heißt **Cache-kohärent,** falls ein Lesezugriff immer den Wert des zeitlich letzten Schreibzugriffs auf das entsprechende Speicherwort liefert – unabhängig davon, ob dieser Schreibzugriff auf den Hauptspeicher oder einen der Caches stattgefunden hat. **Kohärenz** bedeutet allgemeiner das korrekte Voranschreiten des Systemzustands durch ein abgestimmtes Zusammenwirken der Einzelzustände. Im Zusammenhang mit einem Cache-Speicher muss das System dafür sorgen, dass immer die aktuellen und nicht die veralteten Daten gelesen werden.

Ein System ist **konsistent**, wenn alle Kopien eines Speicherworts im Hauptspeicher und den verschiedenen Cache-Speichern stets *identisch* sind. Dadurch ist insbesondere auch die Kohärenz sichergestellt. Eine Inkonsistenz zwischen Cache-Speicher und Hauptspeicher entsteht dann, wenn ein Speicherwort nur im Cache-Speicher und nicht gleichzeitig im Hauptspeicher verändert wird. Solche Inkonsistenzen entstehen bei Anwendung des Rückschreibeverfahren – im Gegensatz zur Anwendung des Durchschreibeverfahrens durch alle Cache-Stufen bis zum Hauptspeicher.

Um alle Kopien eines Speicherworts immer konsistent zu halten, müsste ein hoher Aufwand getrieben werden, der zu einer Leistungseinbuße führen würde. Es kann nun im begrenzten Umfang die Inkonsistenz der Daten zugelassen werden, wenn ein geeignetes **Cache-Kohärenzprotokoll** dafür sorgt, dass die Cache-Kohärenz gewährleistet ist. Wie bereits gasgt, muss dazu das Protokoll sicherstellen, dass immer die aktuellen und nicht die veralteten Daten gelesen werden. Dabei gibt es zwei prinzipielle Ansätze für Cache-Kohärenzprotokoll:

- *Write-update-Protokoll*: Beim Verändern einer Kopie in einem Cache-Speicher müssen alle Kopien in anderen Cache-Speichern ebenfalls ver-

ändert werden, wobei die Aktualisierung auch verzögert, spätestens aber beim Zugriff erfolgen kann.

- *Write-invalidate-Protokoll*: Vor dem Verändern einer Kopie in einem Cache-Speicher müssen alle Kopien in anderen Cache-Speichern für „ungültig“ erklärt werden.

Üblicherweise wird bei symmetrischen Multiprozessoren ein *Write-invalidate-Cache*-Kohärenzprotokoll mit Rückschreibeverfahren angewandt.

3.1.3.7 Busschnüffeln und MESI-Protokoll

Das so genannte **Bus-Schnüffeln** (*Bus Snooping*) wird meist bei symmetrischen Multiprozessoren angewandt, bei denen mehrere Prozessoren mit lokalen Cache-Speichern über einen Systembus an einen gemeinsamen Hauptspeicher angeschlossen sind. Die Schnüffel-Logik jedes Prozessors „hört“ am Speicherbus die Adressen mit, die von den anderen Prozessoren auf den Bus gelegt werden. Die Adressen auf dem Bus werden mit den Adressen der Cache-Blöcke im lokalen Cache-Speicher verglichen. Bei Übereinstimmung der auf dem Systembus erschnüffelten Adresse mit einer der Adressen der Cache-Blöcke im Cache-Speicher geschieht Folgendes:

- Im Fall eines „erschnüffelten” Schreibzugriffs wird der im Cache gespeicherte Cache-Block für „ungültig“ erklärt, sofern auf den Cache-Block nur lesend zugegriffen wurde.
- Wenn ein Lese- oder Schreibzugriff erschnüffelt wird und die Datenkopie im Cache-Speicher verändert wurde, unterbricht die Schnüffel-Logik die Bustransaktion. Sie übernimmt den Bus und schreibt den betroffenen Cache-Block in den Hauptspeicher. Dann wird die ursprüngliche Bustransaktion erneut durchgeführt. Alternativ könnte auch ein direkter Cache-zu-Cache-Transfer durchgeführt werden, der mit *Snarfing* bezeichnet wird.

MESI-Protokoll

Als Cache-Kohärenzprotokoll in Zusammenarbeit mit dem Bus-Schnüffeln hat sich das so genannte **MESI-Protokoll** durchgesetzt (s. Abbildung 3.8). Dieses ist als *Write-invalidate*-Kohärenzprotokoll einzuordnen. Das MESI-Protokoll ordnet jedem Cache-Block einen der folgenden vier Zustände zu:

- *Exclusive modified*: Der Cache-Block wurde durch einen Schreibzugriff geändert und befindet sich ausschließlich in diesem Cache.
- *Exclusive unmodified*: Der Cache-Block wurde für einen Lesezugriff übertragen und befindet sich unverändert nur in diesem Cache.
- *Shared unmodified*: Unveränderte Kopien des Cache-Blocks befinden sich für Lesezugriffe in mehr als einem Cache.
- *Invalid*: Der Cache-Block ist ungültig.

Die Bezeichnung „MESI" wurde aus den Anfangsbuchstaben der Zustände *(exclusive)* ***M****odified*, ***E****xclusive unmodified*, ***S****hared unmodified* und ***I****nvalid* gebildet. Bei der Erläuterung der Funktionsweise des MESI-Protokolls beginnen wir mit einem Lesezugriff auf ein Datenwort in einem Cache-Block, der sich im Zustand *Invalid* befindet. Bei einem solchen Lesezugriff wird ein Cache-Fehlzugriff ausgelöst, und der Cache-Block, der das Datenwort enthält, wird aus dem Hauptspeicher in den Cache-Speicher des lesenden Prozessors übertragen.

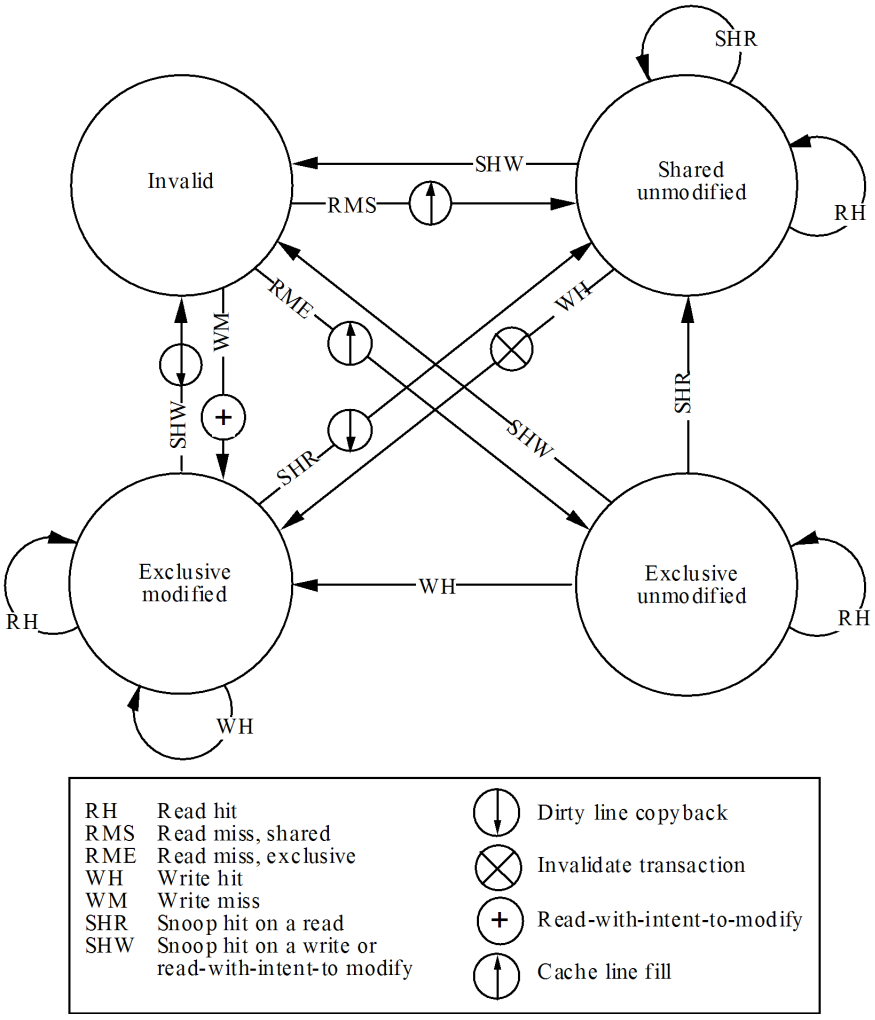


Abbildung 3.8: MESI-Cache-Kohärenzprotokoll.

Je nachdem, ob der Cache-Block bereits in einem anderen Cache-Speicher steht, muss man folgende Fälle unterscheiden:

- Der Cache-Block stand in keinem Cache-Speicher eines anderen Prozessors: Dann wird ein RME (*Read Miss exclusive*) durchgeführt und der übertragene Cache-Block erhält das Attribut *Exclusive unmodified*. Dieses Attribut bleibt so lange bestehen, wie sich der Cache-Block in keinem anderen Cache-Speicher befindet und nur Lesezugriffe des einen Prozessors stattfinden (*Read Hit* – RH).

- Falls beim Lesezugriff erkannt wird, dass der Cache-Block bereits in einem (oder mehreren) anderen Cache-Speichern steht und dort den Zustand *Exclusive unmodified* (bzw. *Shared unmodified*) besitzt, wird das Attribut auf *Shared unmodified* gesetzt (*Read Miss shared* – RMS) und der andere Cache-Speicher ändert sein Attribut ebenfalls auf *Shared unmodified*. Das geschieht durch die Schnüffelaktion SHR (*Snoop Hit on a Read*).

- Falls beim Lesezugriff ein anderer Cache-Speicher den Cache-Block als *Exclusive modified* besitzt, so erschnüffelt dieser die Adresse auf dem Bus (*Snoop Hit on a Read* – SHR), er unterbricht die Bustransaktion, schreibt den Cache-Block in den Hauptspeicher zurück (*Dirty Line Copyback*) und setzt in seinem Cache-Speicher den Zustand auf *Shared unmodified*. Danach wird die Leseaktion auf dem Bus wiederholt.

Falls ein Prozessor ein Datenwort in seinen Cache-Speicher schreibt, so kann dies nur im Zustand *Exclusive modified* des betreffenden Cache-Blocks ohne Zusatzaktion geschehen (*Write Hit* – WH). Der geänderte Cache-Block wird wegen des Rückschreibeverfahrens nicht sofort in den Hauptspeicher zurückgeschrieben.

Ist der vom Schreibzugriff betroffene Cache-Block jedoch nicht im Zustand *Exclusive modified*, so wird die Adresse des Code-Blocks auf den Bus gegeben. Dabei muss man folgende Fälle unterscheiden:

- Der Cache-Block war noch nicht im Cache vorhanden (Cache-Fehlzugriff, Zustand *Invalid*): Es wird ein *Read-with-Intent-to-modify* auf den Bus gegeben. Alle anderen Cache-Speicher erschnüffeln die Transaktion (*Snoop Hit on a Write* – SHW) und setzen ihren Cache-Blockzustand auf *Invalid*, falls dieser vorher *Shared unmodified* oder *Exclusive unmodified* war. In diesen Cache-Speichern kann nun nicht mehr lesend auf den Cache-Block zugegriffen werden, ohne dass ebenfalls ein Cache-Fehlzugriff ausgelöst wird. Der Cache-Block wird aus dem Hauptspeicher übertragen und erhält das Attribut *Exclusive modified*. War der Cache-Block jedoch in einem anderen Cache-Speicher mit dem Attribut *Exclusive modified*, so geschehen (wie im obigen Fall eines Lesezugriffs) eine Unterbrechung der Bustransaktion und ein Rückschreiben des Cache-Blocks in den Hauptspeicher.

- Der Cache-Block ist im Cache-Speicher vorhanden, aber im Zustand *Exclusive unmodified*: Hier genügt eine Änderung des Attributs auf *Exclusive modified*.

- Der Cache-Block ist im Cache-Speicher vorhanden, aber im Zustand *Shared unmodified*: Hier müssen zunächst wieder über eine *Invalidate*-Transaktion der oder die anderen betroffenen Cache-Speicher benachrichtigt werden, dass ein Schreibzugriff durchgeführt wird, sodass diese ihre Kopien des Cache-Blocks invalidieren können.

Ein Cache-Block wird erst dann in den Hauptspeicher zurückgeschrieben, wenn er gemäß der Verdrängungsstrategie ersetzt wird oder einer der anderen Prozessoren darauf zugreifen will.

3.1.3.8 Speicherkonsistenz

Die Lade-/Speichereinheit eines Prozessors führt alle Datentransfer-Befehle zwischen den Registern und dem Daten-Cache-Speicher durch. Cache-Kohärenz wird erst dann wirksam, wenn ein Lade- oder Speicherzugriff durch die Lade-/Speichereinheit des Prozessors ausgeführt wird, also ein Zugriff auf den Cache-Speicher geschieht. Cache-Kohärenz sagt nichts über mögliche Umordnungen der Lade- und Speicherbefehle *innerhalb* der Lade-/Speichereinheit aus. Heutige Mikroprozessoren führen jedoch die Lade- und Speicherbefehle nicht mehr unbedingt in der Reihenfolge aus, wie sie vom Programm her vorgeschrieben sind.[9]

Die Prinzipien der vorgezogenen Ladezugriffe und des nichtblockierenden Cache-Speichers (vgl. Abschnitt 3.1.1) steigern die Verarbeitungsgeschwindigkeit des Prozessors, da neue Daten so schnell wie möglich in die Register geladen werden, damit nachfolgende Verarbeitungsbefehle ausgeführt werden können – das Rückspeichern erscheint demgegenüber nicht so zeitkritisch. Wesentlich ist dabei bei beiden Prinzipien, dass aus Speichersicht die Programmordnung nicht mehr eingehalten wird. Unter der angegebenen Randbedingung – dem lokalen Adressabgleich der betroffenen Datenwerte – bleibt jedoch die Ausführung außerhalb der Programmreihenfolge ohne Auswirkung auf das Ergebnis, sofern nur ein einzelner Prozessor beteiligt ist.

Dies gilt jedoch nicht mehr bei einer parallelen Programmausführung auf einem Multiprozessor. Bei speichergekoppelten Multiprozessoren können im Prinzip zu jedem Zeitpunkt mehrere Zugriffe auf denselben Speicher erfolgen. Bei symmetrischen Multiprozessoren können trotz Cache-Kohärenz durch Verwendung von vorgezogenen Ladezugriffen oder durch nichtblockierende Cache-Speicher die Speicherzugriffsoperationen in anderer Reihenfolge als durch das Programm definiert am Speicher wirksam werden.

Aus Sicht des Programmierers ist weniger die implementierungstechnische Seite der Cache-Kohärenz interessant, als das zugrundeliegende Konsistenzmodell, das etwas über die zu erwartende Ordnung der Speicherzugriffe durch parallel arbeitende Prozessoren aussagt. Genauer gesagt gilt die Definition:

*Ein **Konsistenzmodell** spezifiziert die Reihenfolge, in der Speicherzugriffe eines Prozesses von anderen Prozessen gesehen werden.*

sequentielle Konsistenz

Im Normalfall setzt der Programmierer die **sequenzielle Konsistenz** voraus, die jedoch zu starken Einschränkungen bei der Implementierung führt. Ein Multiprozessorsystem ist **sequenziell konsistent**, wenn das Ergebnis einer beliebigen Berechnung dasselbe ist, als wenn die Operationen aller Prozessoren auf einem Einprozessorsystem in einer sequenziellen Ordnung ausgeführt würden. Dabei ist die Ordnung der von den Prozessoren ausgeführten Operationen durch das jeweilige Programm vorgegeben.[10] Insbesondere verbietet die sequenzielle Konsistenz die Prinzipien der vorgezogenen Ladezugriffe und des nichtblockierenden Cache-Speichers.

[9] Dieser Abschnitt stellt einen Vorgriff auf das Kapitel 4 (Multiprozessorsysteme) dar.

[10] Die Originaldefinition ist durch die Formulierung "... *some sequential order* ..." etwas unklar gefasst [12]: "*The result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.*"

Die sequenzielle Konsistenz stellt zwar die korrekte Ordnung der Speicherzugriffe sicher, nicht jedoch die Korrektheit der Zugriffe auf gemeinsam benutzte Datenobjekte durch parallele Kontrollfäden. Letzteres ist durch geeignete Synchronisationen des Datenzugriffs oder des Prozessablaufs sicherzustellen, was nach dem heutigen Stand Sache des Programmierers ist und in Zukunft vielleicht Sache eines parallelisierenden Compilers sein wird.

Wenn sich der Programmierer über die Auswirkungen einer möglichen Verletzung der Zugriffsordnung nicht im Klaren ist, so wird er solche Verletzungen unter allen Umständen zu vermeiden suchen. Das heißt, er wird vom System die Einhaltung der sequenziellen Konsistenz fordern und dafür dann den dadurch entstehenden hohen, systemimmanenten Synchronisationsaufwand in Kauf nehmen müssen. (Für weitere Erläuterungen sei auf das Kapitel 4 verwiesen.)

Weiß der Programmierer, dass in gewissen Phasen der Programmausführung eine Verletzung der Zugriffsordnung toleriert werden kann, so kann er die Forderung nach einer sequenziellen Konsistenz aufgeben und sich statt dessen mit einer **schwachen Konsistenz** (*weak Consistency*) begnügen, um die Effizienz der Programmausführung zu steigern. Schwache Konsistenz heißt, dass die Konsistenz des Speicherzugriffs nicht mehr zu allen Zeiten gewährleistet ist, sondern nur zu bestimmten, vom Programmierer in das Programm eingesetzten Synchronisationspunkten. Praktisch bedeutet dies die Einführung von *kritischen Bereichen*, innerhalb derer Inkonsistenzen zugelassen werden. Die Synchronisationspunkte sind dabei die Ein- oder Austrittspunkte der kritischen Bereiche.

schwache Konsistenz

Selbsttestaufgabe 3.4 *(Kontrollfragen)*

a) Geben Sie die Speicherhierarchie für heutige PCs geordnet nach absteigenden Zugriffsgeschwindigkeiten und aufsteigenden Speicherkapazitäten an.

b) Was ist die Voraussetzung für eine effiziente Speicherhierarchie?

c) Was versteht man unter Speicherverschränkung?

d) Welcher technologische Unterschied besteht zwischen Cache und Hauptspeicher?

e) Wodurch unterscheidet sich die Rückschreibe- von der Durchschreibe-Strategie?

f) Welche Vor- und Nachteile hat das Durchschreib-Verfahren gegenüber dem Rückschreibverfahren?

g) Welche Vor- und Nachteile weisen virtuell bzw. physikalisch adressierte Cache-Speicher auf?

h) Was bedeutet Cache-Kohärenz? Nennen Sie ein Cache-Kohärenzprotokoll?

Selbsttestaufgabe 3.5 *Gegeben sei ein speichergekoppeltes Multiprozessorsystem mit zwei Prozessoren, die über einen Bus mit einem globalen Speicher verbunden sind. Zur Wahrung der Cache-Kohärenz wird das MESI-Protokoll verwendet. Die Caches der beiden Prozessoren haben je eine Größe von drei Cache Lines die genau ein Speicherwort aufnehmen können. Die Caches werden von der niedrigsten Cache Line aufwärts gefüllt, falls noch freie Lines zur Verfügung stehen. Im Falle eines voll besetzten Caches werden sie nach der Strategie LRU (Least Recently Used) überschrieben. Ergänzen Sie die folgende Tabelle, wobei die Buchstaben die Zustände: M = exclusive Modified, E =*

Exclusive unmodified, S = Shared unmodified, I = Invalid repräsentieren. Die Zahlen hinter den Aktionen und Zuständen bezeichnen Speicheradressen.

Proz.	***Aktion***	***Cache1 Line1***	***Cache1 Line2***	***Cache1 Line3***	***Cache2 Line1***	***Cache2 Line2***	***Cache2 Line3***
-	*-*	*E/8*	*E/12*	*I/-*	*E/6*	*I/-*	*I/-*
2	*read 10*						
1	*write 8*						
1	*read 10*						
2	*read 8*						
1	*write 8*						
1	*write 8*						
1	*read 18*						
2	*write 10*						
2	*write 18*						

3.1.4 Virtuelle Speicherverwaltung

Mehrbenutzer- und Multitasking-Betriebssysteme stellen hohe Anforderungen an die Speicherverwaltung. Programme, die in einer Mehrprozessumgebung lauffähig sein sollen, müssen relozierbar, d.h. im Speicher verschiebbar sein. Das bedeutet, dass die geladenen Programme und ihre Daten nicht an festgelegte, physikalische Speicheradressen gebunden sein dürfen. Weiterhin sollen für jeden Prozess ein eigener großer Adressraum und geeignete Schutzmechanismen zwischen den Prozessen bereitgestellt werden. Zur Durchführung der Schutzmechanismen wird zusätzliche Information über Zugriffsrechte und Gültigkeit der Speicherwörter benötigt.

Relozierbarkeit

Die virtuelle Speicherverwaltung unterteilt den physisch zur Verfügung stehenden Speicher in Speicherblöcke (als Seiten oder Segmente organisiert) und bindet diese an Prozesse. Damit ergibt sich ein Schutzmechanismus, der den Zugriffsbereich eines Prozesses auf seine Speicherblöcke beschränkt. Die Vorteile sind:

virtuelle Speicherverwaltung

- ein großer Adressraum mit einer Abbildung von 2^{32} bis 2^{64} virtuellen Adressen auf beispielsweise 2^{28} physikalische Adressen für physisch vorhandene Hauptspeicherplätze,
- eine einfache Relozierbarkeit (*Relocation*), die es erlaubt, das Programm in beliebige Bereiche des Hauptspeichers zu laden oder zu verschieben,
- die Verwendung von Schutzbits, die beim Zugriff geprüft werden und es ermöglichen, unerlaubte Zugriffe abzuwehren,
- ein schneller Programmstart, da nicht alle Code- und Datenbereiche zu Programmbeginn im Hauptspeicher vorhanden sein müssen,
- eine automatische (vom Betriebssystem organisierte) Verwaltung des Sekundär- und Hauptspeichers, die den Programmierer von diesen Aufgaben entlastet.

Tabelle 3.1: Typische Parameterbereiche für Caches und virtuelle Speicher.

Parameter	Primär-Cache	virtueller Speicher
Block-/Seitengröße	16 – 128 Byte	4 – 64 kB
Zugriffszeit	1 – 2 Takte	40 – 100 Takte
Fehlzugriffsaufwand	8 – 100 Takte	70000 – 6000000 Takte
Fehlzugriffsrate	0.5 – 10%	0.00001 – 0.001%
Speichergröße	8 – 64 kB	16 M – 8 GB

Die Organisationsformen der virtuellen Speicherverwaltung und der Cache-Speicherverwaltung haben einiges gemeinsam. Cache-Blöcke korrespondieren mit den Speicherblöcken, d.h. den Seiten (*Pages*) oder Segmenten der virtuellen Speicherverwaltung. Ein Cache-Fehlzugriff entspricht einem Seiten- oder Segmentfehlzugriff.

Allerdings wird die Cache-Speicherverwaltung vollständig in Hardware ausgeführt, während die virtuelle Speicherverwaltung zum Großteil vom Betriebssystem, also in Software, durchgeführt wird. Die virtuelle Speicherverwaltung erhält dazu von der Hardware Unterstützung durch die so genannte **Speicherverwaltungseinheit** (*Memory Management Unit* – MMU) und durch spezielle Maschinenbefehle.

Die Adressbreite des Prozessors bestimmt die maximale Größe des virtuellen Adressraums und damit des virtuellen Speichers. Die Größe des oder der Cache-Speicher ist davon unabhängig. Die Cache-Speicherverwaltung verschiebt Cache-Blöcke zwischen den Cache-Speichern verschiedener Hierarchieebenen und dem Hauptspeicher. Die virtuelle Speicherverwaltung verschiebt Speicherblöcke zwischen Hauptspeicher und Sekundärspeicher, der in der Regel aus einer Festplatte besteht.

Auch die Größe und Zugriffsgeschwindigkeiten sind für die Cache- und die virtuelle Speicherverwaltungen sehr unterschiedlich. Tabelle 3.1 gibt einen Überblick über die unterschiedlichen Größenordnungen verschiedener Parameter, bei denen sich Caches und virtuelle Speicher unterscheiden.

Aus den Speicherreferenzen in den Maschinenbefehlen wird durch die Adressrechnung eine effektive Adresse berechnet. Diese wird als so genannte **logische Adresse** durch die virtuelle Speicherverwaltung zuerst in eine **virtuelle Adresse** und dann in eine **physikalische Speicheradresse** transformiert (siehe auch Kapitel 1). Häufig wird in der Literatur nicht zwischen logischen und virtuellen Adressen unterschieden. Die hier getroffene Unterscheidung wurde aus der Literatur zum PowerPC übernommen.

Die virtuelle Speicherverwaltung stellt während der Programmausführung fest, welche Daten gerade gebraucht werden, transferiert die angeforderten Speicherbereiche zwischen Sekundärspeicher (z.B. Festplatte) und Hauptspeicher und aktualisiert die Referenzen zwischen virtueller und physikalischer Adresse z.B. in einer Übersetzungstabelle (*Translation Lookaside Buffer* – TLB) oder speziellen Registern. Die Speicherverwaltung kann Bereiche fester Länge, so genannte **Seiten**, oder variabler Länge, so genannte **Segmente**, verwenden.

TLB

Falls eine Referenz auf eine Seite nicht im physikalischen Speicher gefunden werden kann, wird dies als **Seitenfehler** (*Page Fault*) bezeichnet. Die Speicherseite muss dann durch eine Betriebssystemroutine nachgeladen werden. Das Benutzerprogramm bemerkt diese Vorgänge nicht. Die Zeit, die für den Umladevorgang benötigt wird, verringert jedoch die für den Benutzer erreichbare Verarbeitungsgeschwindigkeit. Die Lokalität von Code und Daten und eine geschickte Ersetzungsstrategie für jene Seiten, die im Hauptspeicher durch Umladen überschrieben werden, gewährleistet dennoch eine hohe Wahrscheinlichkeit, dass die Daten, die durch den Prozessor angefordert werden, im physikalischen Hauptspeicher zu finden sind.

Bei der Organisation einer virtuellen Speicherverwaltung stellen sich folgende Fragen:

- *Wo kann ein Speicherblock im Hauptspeicher platziert werden?* Wegen des außerordentlich hohen Aufwands nach einem Fehlzugriff (s. Tabelle 3.1), der linearen Adressierung des Hauptspeichers durch physikalische Adressen und der festen Seitenlänge ist der Speicherort einer Seite im Hauptspeicher beliebig wählbar. Hingegen sollte bei Segmenten wegen ihrer unterschiedlichen Längen auf eine möglichst große Verdichtung geachtet werden, um nicht durch Fragmentierung des Freispeichers (so genannte *externe Fragmentierung*) so kleine freie Hauptspeicherbereiche zu erhalten, dass diese nicht mehr belegt werden können.

- *Welcher Speicherblock sollte bei einem Fehlzugriff ersetzt werden?* Die übliche Ersetzungsstrategie ist die LRU-Strategie (*Least Recently Used*), die anhand von Zusatzbits für jeden Speicherblock feststellt, auf welchen Speicherblock am längsten nicht mehr zugegriffen wurde, und diesen ersetzt.

- *Wann muss bei Speicherknappheit ein zu verdrängender Speicherblock in den Sekundärspeicher zurückgeschrieben werden?* Falls der zu verdrängende Speicherblock im Hauptspeicher ungeändert ist, so kann dieser einfach durch einen neuen Block überschrieben werden. Andernfalls muss der zu verdrängende Speicherblock in den Sekundärspeicher zurückgeschrieben werden.

- *Wie wird ein Speicherblock aufgefunden, der sich in einer höheren Hierarchiestufe befindet?* Eine Tabelle zur Verwaltung der Seiten oder Segmente kann sehr groß werden. Um die Tabellengröße zu reduzieren, werden meistens Hash-Verfahren auf den virtuellen Seiten-/Segmentadressen durchgeführt. Dies führt dazu, dass die Tabellengröße deutlich verringert werden kann. Diese ist meist sehr viel kleiner als die Zahl der virtuellen Seiten. Man spricht dann von einer invertierten Seitentabelle (*Inverted Page Table*). Eine andere Möglichkeit, die z.B. bei den Intel-Prozessoren angewandt wird, besteht darin, die Verwaltung mehrstufig über mehrere Tabellen zu organisieren, wie es vergleichbar bei der Verwaltung von Dateien in Verzeichnissen und Unterverzeichnissen geschieht.

- *Welche Seitengröße sollte gewählt werden?* Es gibt gute Gründe für große Seiten: Die Größe der Seitentabelle ist umgekehrt proportional zur gewählten Seitengröße. Größere Seiten sparen Speicherplatz, da eine kleinere Seitentabelle genügt. Weiterhin kann ein Transport großer Seiten zwischen Haupt- und Sekundärspeicher effizienter organisiert werden als ein häufiger Transport kleiner Seiten. Die Zahl der Einträge in der Adressübersetzungstabelle (TLB) der Speicherverwaltungseinheit ist beschränkt. Große Seiten erfassen mehr Speicherplatz und führen damit zu weniger TLB-Fehlzugriffen (s. später in diesem Abschnitt). Für kleine Speicherseiten spricht die geringere Fragmentierung des Speichers (so genannte *interne Fragmentierung*) und ein schnellerer Prozessstart. Die Lösung ist meist eine Hybridtechnik, die es erlaubt, eine kleine Anzahl von unterschiedlichen Seitengrößen zuzulassen oder Segmente und Seiten zu kombinieren.

Wie bereits oben gesagt, geschieht die Übersetzung einer logischen in eine physikalische Adresse meist nicht über *eine*, sondern über *mehrere* Übersetzungstabellen. Oft werden zwei Typen von Übersetzungstabellen – Segmenttabellen und Seitentabellen – oder aber mehrere, hierarchisch angeordnete Seitentabellen benutzt.

Bei der Speichersegmentierung wird der gesamte Speicher in ein oder mehrere Segmente variabler Länge eingeteilt. Für jedes Segment können Zugriffsrechte getrennt vergeben werden, die dann bei der Adressübersetzung nachgeprüft werden. Ebenso wird beim Übersetzen einer logischen Adresse getestet, ob die erzeugte virtuelle Adresse überhaupt innerhalb des vom Betriebssystem reservierten Speicherbereichs liegt. Zusammen mit einer prozessspezifischen Festlegung der Zugriffsrechte kann ein Speicherschutzmechanismus realisiert werden.

Speicherschutz

Alternativ oder zusätzlich zur Speichersegmentierung wird die Seitenübersetzung eingesetzt. Dabei wird der Adressraum in Seiten fester Größe zerlegt (Größenordnung 1 bis 64 kB, häufig 4 kB). Die virtuelle Adresse wird in eine virtuelle Seitenadresse und eine Offset-Adresse aufgeteilt. Bei der Übersetzung einer virtuellen in die physikalische Adresse wird die virtuelle Seitenadresse mit Hilfe von möglicherweise mehrstufigen Übersetzungstabellen in eine physikalische Seitenadresse umgewandelt, während die Offset-Adresse unverändert übernommen wird.

In Abbildung 3.9 ist der Adressübersetzungsmechanismus der Intel-Prozessoren ab dem 80386 dargestellt, bei dem vor der Seitenübersetzung zusätzlich noch eine Segmentierung stattfindet.

Bei den Intel-Prozessoren besteht eine virtuelle Adresse, die auch als *logische* Adresse bezeichnet wird, aus einem Segmentselektor und einer Verschiebung (*Offset*). Der Selektor steht in einem so genannten *Segmentregister* und wird meist implizit durch den Maschinenbefehl ausgewählt, während die Verschiebung explizit im Befehl spezifiziert wird.

In der ersten Phase der Adressübersetzung (*Segmentation*) wird die logische Adresse in eine *lineare* Adresse übersetzt. Zu diesem Zweck wird der Segmentselektor als Zeiger in eine Deskriptortabelle verwendet, um einen Segmentdeskriptor auszuwählen. Dieser Deskriptor enthält die Informationen über die

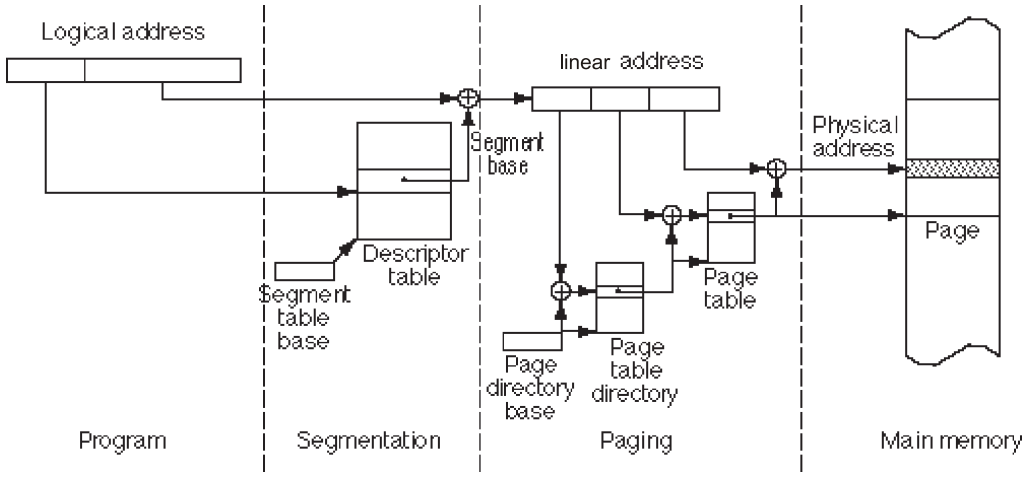


Abbildung 3.9: Automatische Adressübersetzung bei den Intel-Prozessoren ab 80386

Basisadresse und die Länge des Segments und über die erforderlichen Zugriffsrechte. Die lineare 32-Bit-Adresse berechnet sich durch Addition von Segment-Basisadresse und Verschiebung. Falls die Verschiebung größer oder gleich der Segmentlänge ist oder wenn der Zugriff nicht erlaubt ist, wird eine Unterbrechung ausgelöst.

In der zweiten Phase (*Paging*) wird aus der linearen Adresse in einem zweistufigen Verfahren die physikalische Adresse erzeugt. Dazu wird die lineare Adresse in drei Felder zerlegt. Mit den höchstwertigen 10 Bits wird ein Verzeichniseintrag aus dem Seitenverzeichnis (*Page Table Directory*) ausgewählt. Dieser Eintrag enthält die Adresse der zugehörigen Seitentabelle (*Page Table*), aus der mit den nächsten 10 Bits der Tabelleneintrag mit der endgültigen physikalischen Seitennummer ausgewählt wird. Die restlichen 12 Bits der virtuellen Adresse werden unverändert an die physikalische Seitennummer angefügt (Konkatenation) und bilden zusammen mit dieser die physikalische Adresse. Jeder Eintrag im Seitenverzeichnis und in den Seitentabelle besteht aus einer 20 Bit breiten Seitennummer und einigen Verwaltungsbits. Insbesondere gibt es ein *Present*-Bit, das anzeigt, ob die Seite im Hauptspeicher vorhanden ist. Ist dieses Bit nicht gesetzt, wird eine Seitenfehlzugriff-Unterbrechung (*Page Fault*) ausgelöst, welcher es dem Betriebssystem gestattet, die fehlende Seite nachzuladen. Bei den Intel-Prozessoren kann die beschriebene Seitenübersetzung über ein Bit in einem Steuerregister ausgeschaltet werden.

Das softwaremäßige Durchlaufen der verschiedenen Übersetzungstabellen der virtuellen speicherverwaltung dauert verhältnismäßig lange, sodass eine Hardwareunterstützung unabdingbar ist.

Für die segmentierte Speicherverwaltung (*Segmentation*), bei der die relativ großen Segmente seltener ausgetauscht werden müssen, reicht es, die wichtigsten Daten der aktuellen Segmente aus den Segmentdeskriptortabellen in die speziellen **Segmentregister** zu übertragen, auf die erheblich schneller zugegriffen werden kann.

Eine schnelle Adressumsetzung der Seitenspeicherverwaltung (*Paging*) wird durch die oben bereits erwähnte Hardwaretabelle erreicht, die die zuletzt ausgeführten Adressumsetzungen enthält und als **Adressübersetzungspuffer** (*Translation Lookaside Buffer* – TLB) bezeichnet wird. Dieser ist ein meist nur 32 bis 128 Einträge großer Cache und wird vollassoziativ verwaltet, d.h. der Vergleich der zu übersetzenden Adresse geschieht simultan in einem Takt mit allen im TLB gespeicherten Einträgen. Meist ist der TLB um zusätzliche Logik zur Seitenverwaltung und für den Zugriffsschutz erweitert. Für diese Einrichtung hatten wir weiter oben bereits den Ausdruck **Speicherverwaltungseinheit** (*Memory Management Unit* – MMU) eingeführt.

Ist ein gewünschter Eintrag im Adressübersetzungspuffer nicht vorhanden – liegt also ein TLB-Fehlzugriff (*TLB Miss*) vor –, so wird eine Unterbrechung ausgelöst, welche die Übersetzungstabellen der Seitenspeicherverwaltung softwaremäßig durchläuft, um die physikalische Adresse herauszufinden, und dann das Adresspaar aus linearer und physikalischer Adresse in den TLB einträgt.

Für den Befehls- und den Daten-Cache gibt es bei heutigen Mikroprozessoren auf dem Prozessor-Chip getrennte Speicherverwaltungseinheiten mit eigenen TLBs. Damit kann die Adressübersetzung für Code und Daten parallel zueinander durchgeführt werden. Die TLB-Zugriffe geschehen meist im gleichen Takt, in dem auch auf die Primär-Cache-Speicher zugegriffen wird. Ein TLB-Eintrag besteht aus dem *Tag*, der einen Teil der virtuellen Adresse enthält, einem Datenteil mit einer physikalischen Seitennummer (*Physical Page Frame Number*) sowie weiteren Verwaltungs- und Schutzbits für die folgenden Informationen:

- die Seite befindet sich im Hauptspeicher (*resident*),
- der Zugriff ist nur dem Betriebssystem erlaubt (*Supervisor*),
- ein Schreibzugriff ist verboten (*read-only*),
- die Seite wurde verändert (*dirty*) oder
- ein Zugriff auf die Seite ist erfolgt (*referenced*).

Eine Erweiterung stellt der *Tagged TLB* dar, bei dem jeder TLB-Eintrag zusätzlich einen Prozess-*Tag* enthält, mit dem die Adressräume der verschiedenen Prozesse unterschieden werden können. Im Falle eines Prozesswechsels müssen die TLB-Einträge nicht gelöscht werden (*TLB Flush*), sondern werden nach Bedarf verdrängt. Tagged TLBs finden sich beim MIPS R4000, während die TLBs bei den Pentium- und PowerPC-Prozessoren keine Prozess-Tags verwenden und nach einem Prozesswechsel alle TLB-Einträge explizit gelöscht werden müssen.

Selbsttestaufgabe 3.6 *Welche der folgenden Maßnahmen tragen dazu bei, die Häufigkeit von Seitenfehlern zu reduzieren?*

a) Vergrößerung des Hauptspeichers,

b) Vergrößerung des Sekundärspeichers,

c) Verbesserung der Verdrängungsstrategie bei der Auslagerung von Seiten.

Selbsttestaufgabe 3.7 *Der folgende, aus der digitalen Signalverarbeitung stammende Algorithmus soll auf einem Prozessor mit virtueller Speicherverwaltung ausgeführt werden:*

(01) FOR $i = 0$ *TO* $2^{13} - 1$

(02) y = 0;
(03) FOR j = 0 TO 2
*(04) y = y + x[i+j] * w(j);*
(05) END
(06) y[i] = y;
(07) END

Der Prozessor verfügt über drei Segmentregister: CS für Programmcode, DS für Datenstrukturen und SS für den Stapel (Stack). Die Auswahl eines Segmentregisters erfolgt implizit durch den auszuführenden Befehl. Die Bildung der virtuellen Adresse erfolgt durch Addition der Basisadresse mit der logischen Adresse. Die Seitengröße beträgt 4096 Wörter, wobei der Hauptspeicher maximal 4 Seiten aufnehmen kann. Seiten der CS- und der SS-Segmente werden vom Betriebssystem nicht ausgelagert. Als Ersetzungsstrategie kommt LRU zum Einsatz. Die Segmentregister sind wie folgt belegt:

Segment	*Basisadresse*	*Länge (hex.)*
CS	*3A17C000*	*1000*
DS	*D29E4000*	*4000*
SS	*51F60000*	*1000*

Das Feld x im obigen Programm beginnt an der logischen Adresse 2000, y bei 0. Die Seitentabelle enthält zu Beginn folgende Einträge:

Virtuelle Adresse	*Physikalische Adresse*	*Hauptspeicher*
3A17C	*0*	*ja*
51F60	*1*	*ja*
D29E6	*2*	*ja*
D29E7	*3*	*ja*

a) Erstellen Sie ein Protokoll der ein- und ausgelagerten Seiten während der Ausführung des Programms, solange keine Zugriffsverletzung auftritt.
Geben Sie jeweils die Iteration und die Zeilennummer an, bei der ein Seitenfehler auftritt. Wie viele Seitenwechsel werden insgesamt durchgeführt?

b) Gibt es eine Strategie zur Seitenersetzung, die besser ist als LRU? Wenn ja, wie viele Ein-/Auslagerungen müssen dann durchgeführt werden?

3.2 Innovative Mikroprozessor-Techniken

Das sog. Moore'sche Gesetz besagt, dass die Anzahl der Transistoren pro Prozessor-Chip etwa alle 18 Monate verdoppelt werden kann und dass sich im gleichen Zeitraum auch die Verarbeitungsleistung der Mikroprozessoren verdoppelt. Man rechnet heute damit, dass diese Prognose noch zumindest bis zum Jahr 2029 weiter gültig sein wird.

Mit dem Hundertfachen heutiger Transistorzahlen pro Prozessor-Chip in einem Jahrzehnt wird deshalb der Einsatz völlig neuer Techniken in der Architektur und der Mikroarchitektur zukünftiger Mikroprozessoren ermöglicht.

Somit erhebt sich die Frage, welche Architektur- und Implementierungstechniken für Mikroprozessoren mit einer Milliarde oder gar noch mehr Transistoren adäquat sind.

3.2.1 Stand und Grenzen heutiger Prozessortechniken

Der Stand der Technik heutiger Mikroprozessoren ist geprägt durch eine entkoppelte, vielstufige Befehls-Pipeline sowie durch eine superskalare Befehlszuordnung auch außerhalb der Programmordnung und durch die VLIW-/EPIC-Technik. Die bekanntesten kommerziellen Vertreter superskalarer Prozessoren in Desktop PCs sind die AMD- und Intel-Prozessoren. Beide Prozessorfamilien sind schon recht lange im Einsatz und haben ihre Leistungen mehrere Male durch verbesserte Technologie gesteigert, d.h. insbesondere mehr Transistoren für größere Caches, höhere Taktraten durch Spannungssenkung und kleinere, schnellere Transistoren.

Seit etwa 2005 sah man allerdings auch in diesem Bereich eine Trendwende, die in der Forschung und im Server-Bereich schon früher eingesetzt hat und die wir im nächsten Abschnitt besprechen werden. Die beiden bekanntesten Mikroprozessoren zu dieser Zeit waren der AMD Athlon und der Intel Pentium 4, die wir in den folgenden Abschnitten kurz darstellen – und das, obwohl sie nicht die modernsten Prozessoren ihrer jeweiligen Herstellerfirma sind. Die Grundprinzipien ihrer Architekturen sind jedoch auch in den neueren Prozessorfamilien weiterhin zu finden.

Beide genannten Prozessorfamilien haben im Laufe ihrer Marktpräsenz eine Reihe von Variationen erfahren. Zum einen gibt es mehrere Varianten, so zum Beispiel vom Athlon die leistungsstärkeren Varianten Athlon II und Athlon X2 sowie die preiswerte Variante Duron mit kleineren Caches, oder vom Pentium 4 die Variante Xeon für Server und die preiswerte Variante Celeron. Beide haben auch mehrere Erneuerungen erfahren, ohne dass aber das Grundprinzip wesentlich geändert wurde, so kamen z.B. bei AMD Opteron und Sempron hinzu. Alle Varianten sind in mehreren Betriebsfrequenzen erhältlich.

3.2.1.1 Athlon

Im Oktober 1998 wurde der Athlon (Codename K7) als Nachfolger der AMD-K6-Serie der Öffentlichkeit vorgestellt. Der Athlon implementiert die IA-32-Architektur, allerdings mit einem anderen Satz von Multimediabefehlen – der 3DNow!-Erweiterung –, und ähnelt in seinem Aufbau und seiner Leistung ansonsten dem Pentium III-Prozessor. Das Blockschaltbild (s. Abbildung 3.10) zeigt den Aufbau und die Einheiten des Athlon. Es soll im Folgenden kurz erklärt werden.

Die Befehle werden durch einen 64 kB großen, zweifach satzassoziativen Code-Cache zur Verfügung gestellt. Einfache x86-Befehle werden durch einen der drei unabhängig arbeitenden Befehlsdecodern in so genannte **MacroOps** umgewandelt, die einen oder zwei RISC-artige Operationen enthalten. Komplexe Befehle werden über den so genannten *Vector Path* und ein Microcode-ROM in mehrere MacroOps übersetzt. Die MacroOps entsprechen weitgehend

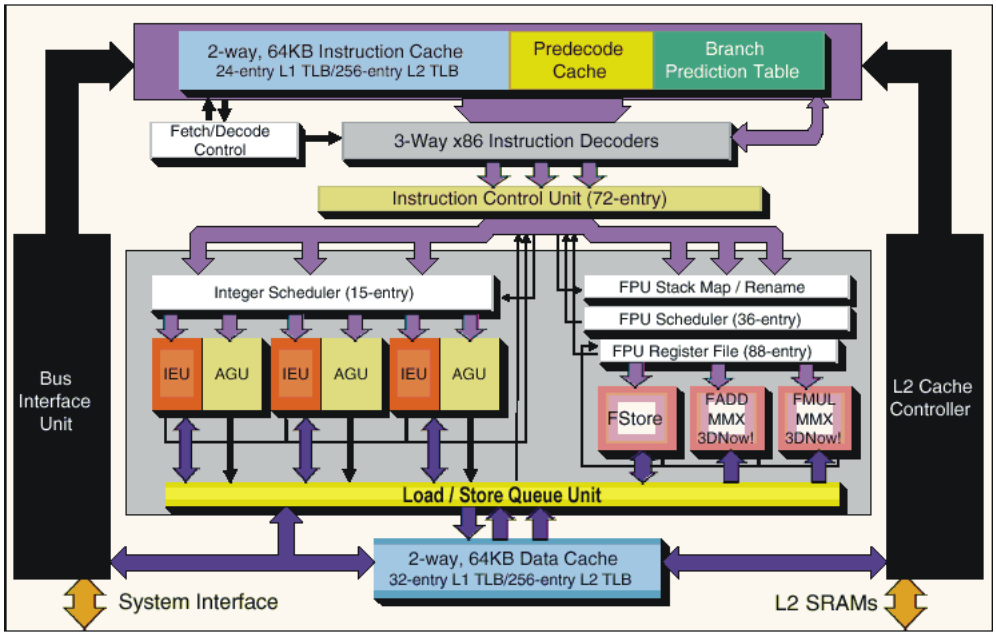


Abbildung 3.10: Blockschaltbild des Athlon.

den Micro-Ops der Intel Pentium-Prozessoren, können allerdings komplexere Operationen umfassen.

Diese MacroOps kommen in einen 72 Plätze fassenden Rückordnungspuffer in der Befehlssteuereinheit (*Instruction Control Unit* – ICU). Die ICU ist für die Registerumbenennung, die Zuordnung zu den Ausführungseinheiten, die Rückordnung und die Behandlung von Ausnahmen zuständig. Bei der Rückordnung werden die gespeicherten Befehle wieder gelöscht. In einem Takt werden bis zu drei MacroOps an die beiden Zuordnungspuffer für die Integer-Rechenwerke (*Integer Execution Unit* – IEU) bzw. Adressberechnungseinheiten (*Address Generation Unit* – AGU) und für die Gleitkommaeinheiten (*Floating Point Unit* – FPU) gegeben.

Der Scheduler für die Integer-Einheiten prüft die Ausführbarkeit der MacroOps und kann bis zu sechs MacroOps – potenziell auch außerhalb der Programmordnung (*out-of-Order*) – an die Integer- und Adressberechnungseinheiten zuweisen. Die Einheiten haben einen Durchsatz von 1. Gleichzeitig werden bis zu drei MacroOps an die Gleitkommaeinheiten gegeben, die mit einer Pipeline implementiert sind und damit in jedem Takt eine neue MacroOp aufnehmen können. Die Gleitkommaeinheiten berechnen alle normalen Gleitkommabefehle sowie die Multimediabefehle von 3DNow!. Damit können bis zu neun RISC-artige Befehle pro Takt ausgeführt werden. Die Pipeline für Integer-Berechnungen hat eine Länge von 10 Stufen und die Gleitkommabefehle müssen 15 Stufen durchlaufen.

Nach der Ausführung der Lade-/Speicher-Befehle werden diese in einen Lade-/Speicher-Puffer (*Load/Store Queue*) geschrieben. Hier wird der Zugriff auf die Daten organisiert, die im L1-Daten-Cache liegen oder aus dem L2-Cache bzw. dem Arbeitsspeicher nachgeladen werden müssen.

Der L1-Daten-Cache ist 64 kB groß, zweifach satzassoziativ und kann zwei Cache-Zugriffe pro Takt ausführen (*dual-ported*). Er unterstützt das MOESI-Kohärenz-Protokoll mit Busschnüffeln für Mehrprozessorsysteme – ein dem MESI-Protokoll ähnliches Protokoll mit einem zusätzlichen Zustand O (*Owned*), den man auch als *Modified Shared* bezeichnen kann.

Die Sprungvorhersage wurde durch einen 2-Bit-Prädiktor mit einer 2048 Einträge umfassenden Sprungverlaufstabelle (*Branch History Table*), unterstützt von einem Sprungzieladress-Cache (*Branch Target Buffer*), realisiert. (Neuere Athlon-Versionen nutzen eine andere Sprungvorhersage, auf die wir hier nicht eingehen können.) Als spezielle Unterstützung für Unterprogrammaufrufe ist ein Rücksprungadress-Stapelspeicher (*Return Stack*) implementiert, der bei einem Call-Befehl die Rücksprungadresse speichert. Er kann die Rücksprungadressen von bis zu 12 Unterprogrammaufrufen speichern.

Der Athlon kam – nach Stand Mitte 2000 – noch mit lediglich 37 Millionen Transistoren aus.

3.2.1.2 Pentium 4

Der Pentium 4 mit der so genannten P7-Mikroarchitektur kam Ende 2000 auf den Markt und wurde bis zum Jahr 2008 produziert. Er stellte somit den Stand der Intel-Prozessortechnik zu Beginn des 21. Jahrhunderts dar. Der Pentium 4 implementierte eine IA-32-Architektur (*Intel Architecture 32 Bit*) mit einem zum Pentium III kompatiblen, aber um die SSE2-Befehle erweiterten Befehlssatz. Die zusätzlichen 144 Befehle der *Internet Streaming SIMD Extensions 2* – SSE2 genannt – erweitern die graphikorientierten Multimediabefehle und kombinieren diese mit den Gleitkommabefehlen. Sie arbeiten auf 128 Bit breiten kombinierten Multimedia-/Gleitkomma-Registern, wobei mit einem SSE2-Befehl pro Takt die Ausführung von vier einfach oder zwei doppelt genauen Gleitkommabefehlen angestoßen wird. Die Gleitkommabefehle sind IEEE-754-konform und arbeiten mit 32- oder 64-Bit-Genauigkeit, im Gegensatz zu der 80 Bit erweiterten Genauigkeit aller früheren Pentium-Prozessoren.

Der zuletzt gefertigte Pentium-4-Prozessor, der Anfang 2006 erschien und auf den sich alle genannten Daten beziehen, wurde in 65-nm-Technologie gefertigt und bestand aus bis zu 188 Millionen Transistoren. Er wurde mit Taktraten von 3,0 bis 3,6 GHz angeboten, war jedoch von seiner Mikroarchitektur her auf noch wesentlich höhere Taktraten bis hin zu 5 GHz entworfen. Wesentliches Entwurfskriterium dafür ist die sehr lange Pipeline, die mit 20 Stufen allein für die Ganzzahlbefehle doppelt so lang wie die Pentium-III-Pipeline ist. Und darin sind die Stufen für die Befehlsbereitstellung und das Decodieren noch nicht einmal mitgerechnet. So hohe Taktraten werden erst durch die Einfachheit jeder der Stufen ermöglicht.

Einer der Nachteile einer langen Pipeline ist die hohe Anforderung an die Sprungvorhersage. Beim Pentium 4 können bis zu 126 Befehle „*on-the-fly*“ in der Pipeline vorhanden sein. Das erfordert eine vielstufige Sprungvorhersage und bedingt einen hohen Aufwand nach einer Fehlspekulation. Über die Sprungvorhersagetechnik des Pentium 4 ist leider nichts bekannt, außer dass eine „erweiterte“ Sprungvorhersage auf Micro-Ops-Basis mit 4 kB großen Ta-

bellen eingesetzt wird, welche die Fehlvorhersagen gegenüber dem Pentium III um ein Drittel verringern sollte.

Der Pentium-4-Prozessor enthält einen 16 kB großen L1-Daten-Cache und einen bis zu 2 MB großen, achtfach assoziativen Sekundär-Cache-Speicher – *Advanced Transfer Cache* genannt – auf dem Prozessor-Chip.

Der Daten-Cache kann zwei Zugriffe pro Takt ausführen und besitzt eine Lese-Latenzzeit von zwei Takten. Er ist mit einer Hardware-Vorabladeeinrichtung versehen, die die Zugriffsmuster überwacht und bei einem regelmäßigen Zugriffsmuster Daten vorausschauend in den Daten-Cache lädt.

Der Sekundär-Cache enthält als *unified Cache* sowohl Code als auch Daten. Er arbeitet blockierungsfrei und ist mit einer 256 Bit breiten Schnittstelle zum Prozessorkern versehen. Der maximale Durchsatz zum Prozessorkern beträgt damit für einen 3,0-GHz-Prozessor ungefähr 96 GB pro Sekunde. Die Cache-Blöcke sind 128 Bytes groß. Um den Nachteil großer Cache-Blöcke beim Transfer in den Speicher nach einem Schreibzugriff abzumindern, sind die Cache-Blöcke in zwei 64 Byte breite Sektionen mit eigenen *Cache Tags* unterteilt. Nach einem Schreibzugriff muss nur noch die betreffende Sektion zurückgeschrieben werden.

Der 800 MHz-Systembus ist ein so genannter *Quad-pumped*-Bus, der durch ein spezielles Signalisierungsschema mit einem 200-MHz-Takt betrieben wird und eine Transferrate von 6,4 GB pro Sekunde erreicht.

Ein neuer Bestandteil der Mikroarchitektur ist der Trace Cache, der als *Level 1 Execution Trace Cache* bezeichnet wird und nach der Decodierphase angesetzt ist. Der Trace Cache ersetzt den L1-Code-Cache und kann 12 k Micro-Ops speichern. IA-32-Befehle werden direkt aus dem Sekundär-Cache geladen, decodiert und in den Trace Cache eingefügt. Bei der Decodierung eines Befehls werden ein oder mehrere Micro-Ops erzeugt. Jeder Micro-Op besitzt die Komplexität eines RISC-Befehls, ist jedoch viel länger. Einfache IA-32-Befehle, die nur mit Registern arbeiten, werden direkt in einzelne Micro-Ops umgesetzt. Komplexe IA-32-Befehle erzeugen jeweils mehrere Micro-Ops. Noch komplexere Befehle führen zum Aufruf eines im Prozessor gespeicherten Mikroprogramms, das einige Dutzend Micro-Ops umfassen kann. Die Micro-Ops kann man als RISC-Anteile zusammengesetzter CISC-Befehle ansehen. Alle heutigen IA-32-kompatiblen Prozessoren nutzen ähnliche Techniken, um die komplexen CISC-Befehle in interne „RISC-Befehle“ umzusetzen und diese dann auf einem superskalaren RISC-Prozessorkern auszuführen.

Nach einer Anlaufphase werden die meisten Befehle im Trace Cache gefunden, sodass die Pipeline mit dem Befehleladen aus dem Trace Cache beginnt. Um Platz zu sparen, speichert der Trace Cache für die komplexen Befehle nicht alle ihre Micro-Ops, sondern nur den ersten davon. Den Rest erzeugt dann wieder ein Mikrocode-Sequenzer. Weiterhin können die Trace-Cache-Blöcke miteinander verkettet sein, sodass nach einem Trace-Cache-Block sofort der in dynamischer Programmordnung nächste zugeordnet werden kann.

Der Trace Cache kann pro Takt drei Micro-Ops liefern. Für die Registerumbenennung stehen 128 Allzweckregister zur Verfügung. Weiterhin können bis zu sechs Micro-Ops pro Takt den Ausführungseinheiten aus zwei getrennten Umordnungspuffern, einem gemeinsamen für die Gleitkomma- und SSE-Einheiten

und einem gemeinsamen für die Integer- und Adressgenerierungseinheiten, zugeordnet werden. Außerdem können drei Micro-Ops pro Takt rückgeordnet werden.

Der Pentium 4 besitzt 11 Ausführungseinheiten. Die zwei arithmetisch-logischen Einheiten (*Integer ALUs*) werden als *Rapid Execution Engine* bezeichnet, da sie mit doppelter Prozessortaktrate betrieben werden, also in jedem Halbtakt jeweils eine Ganzzahloperation beenden können.

3.2.2 Grenzen heutiger Prozessortechniken

Superskalare Mikroprozessoren sind bis zu sechsfach superskalar, d.h. sie können bis zu sechs Befehle gleichzeitig zuordnen und ausführen. Die PC-Prozessoren von Intel und AMD holen und decodieren pro Takt bis zu drei CISC-Befehle des Intel IA-32-Befehlssatzes, erzeugen daraus einfachere interne Befehle (Micro-Ops) und können davon pro Takt bis zu fünf (Intel Pentium II und III) bzw. bis zu neun (AMD Athlon) den Ausführungseinheiten zuordnen. Die VLSI-Technologie wird es in naher Zukunft ermöglichen, Mikroprozessoren mit noch höheren superskalaren Zuordnungsraten zu implementieren. Damit eine solch hohe Zuordnungsrate einen Laufzeitgewinn erzielt, muss der Compiler oder die Hardware ein hohes Maß an Parallelität aus dem Befehlsstrom nutzbar machen. Die Befehlsebenen-Parallelität in einem sequenziellen Befehlsstrom ist jedoch insbesondere für nicht numerische Anwendungen sehr beschränkt, was anhand der durchschnittlichen Befehlsausführungen pro Takt (*Instructions Per Cycle* – IPC) für Testprogramme, sog. Benchmark-Programme, auf realen Prozessoren verdeutlicht werden kann.

Benchmark

Vielfach superskalare Prozessoren, also solche, die 8-, 16- oder 32-fach superskalar sind, stoßen in hohem Maße an die Grenzen der in einem sequenziellen Befehlsstrom enthaltenen feinkörnigen Parallelität. Die Frage, wie viel Parallelität sich im Zielcode eines Programms befindet, das aus einer sequenziellen imperativen Programmiersprache in einen RISC-Befehlssatz übersetzt wird, wurde in den letzten Jahren häufig untersucht. Untersuchungen haben gezeigt, dass Programme häufig etwa fünffache und selten mehr als siebenfache feinkörnige Parallelität besitzen. Höhere Parallelitätsgrade können jedoch erreicht werden, wenn sich ein Code mit sehr langen Basisblöcken, d.h. Befehlsfolgen ohne Verzweigungen, erzeugen lässt, was üblicherweise nur bei numerischen Programmen in Verbindung mit Compilertechniken der Fall ist, die Schleifen in dem Sinne „abrollen“ (*Loop Unrolling*), dass sie die Schleife durch mehrfaches Hinschreiben des Schleifenrumpfes ersetzen.

Weitere Studien zeigen die Grenzen der Prozessorleistung bei Superskalarprozessoren auf. So wurden beispielsweise für eine „genormte” Menge von Testprogrammen, der sog. SPEC95-Suite, auf einem PentiumPro-Prozessor IPC-Werte zwischen 0,6 und 2,0 ermittelt. Messungen kommerzieller Datenbank-Transaktionssysteme auf einem symmetrischen Multiprozessor mit vier Pentium-Pro-Prozessoren zeigten einen IPC-Wert von 0,29 und ähnliche Messungen auf einem Alpha-21164-Prozessor einen noch ungünstigeren IPC-Wert von nur 0,14.

Die niedrigen IPC-Werte ergeben sich aus Fehlspekulationen sowie aus Taktverlusten, welche durch Cache-Fehlzugriffe entstehen, die insbesondere bei Da-

tenbank-Transaktionssystemen durch den großen Satz von Arbeitsdaten begründet sind.

Abbildung 3.11 veranschaulicht die Verluste, die durch nicht ausgefüllte Befehlsfächer bei einem Superskalarprozessor entstehen. (Nicht berücksichtigt sind jedoch die Verluste durch Fehlspekulationen.) Falls wegen eines Cache-Fehlzugriffs über mehrere Takte hinweg überhaupt kein Befehl zugeordnet werden kann, so entstehen **vertikale Verluste**, die in Abbildung 3.11 durch Zeilen nur mit leeren Kästchen veranschaulicht sind. Falls nicht die volle Zuordnungsbandbreite genutzt werden kann, entstehen **horizontale Verluste**. Diese sind durch Zeilen mit einzelnen leeren Kästchen in Abbildung 3.11 gegeben.

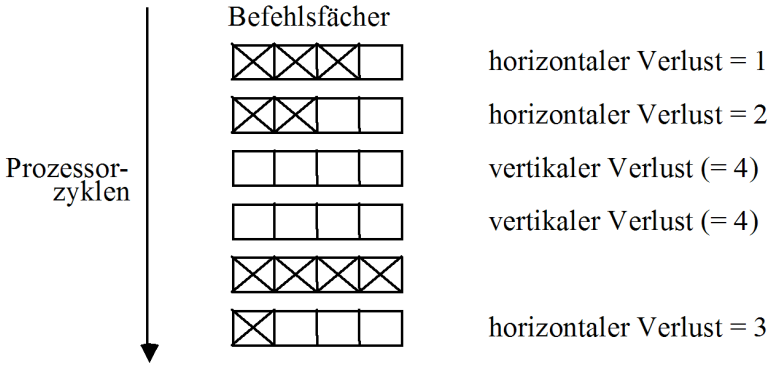


Abbildung 3.11: Verluste durch nicht ausgefüllte Befehlsfächer.

Bei der Weiterentwicklung der Superskalarprozessoren wird versucht, die leeren Befehlsfächer durch spekulative Befehle zu füllen. Es ist jedoch zweifelhaft, ob größere Zuordnungsraten, größere Befehlspuffer und die Nutzung von mehr spekulativer Parallelität bei der Weiterentwicklung heutiger Superskalarprozessoren zu einer wesentlich besseren Leistung führen werden. Der wesentliche Leistungsgewinn wurde deshalb in der jüngeren Vergangenheit durch die enorme Erhöhung der Taktrate auf Werte bis zu 4 GHz erzielt. So ist der Pentium 4 zwar weniger breit superskalar als sein Vorgänger, dafür jedoch mit seiner zwanzigstufigen Pipeline auf hohe Taktraten hin optimiert.

Allerdings stößt auch die Erhöhung der Taktrate an ihre Grenzen. Zum einen führt sie – trotz technologisch erzielter Absenkung der Versorgungsspannung – zu einer stetigen Erhöhung der Leistungsaufnahme des Mikroprozessors. Da ein großer Teil der aufgenommenen Leistung in Wärme umgesetzt wird, wird die Kühlung der Prozessor-Oberfläche immer schwieriger. In Rechenzentren führt der gesamte Energieverbrauch zu immer höheren Kosten für die Energie selbst und für die Kühlung der Rechnerräume. In mobilen Rechnern führt der Energieverbrauch zu kürzeren Abständen zwischen Akku-Aufladungen oder zu höheren Kosten durch leistungsfähigere Akkus. Zum zweiten lässt sich die Taktrate nur weiter erhöhen, wenn die Schaltzeiten von Transistoren durch Verkleinerung verkürzt werden oder wenn in einer Pipeline-Stufe weniger Transistoren durchlaufen werden müssen. Beides wird zunehmend schwieriger und führt zu immensen Kosten in der Fertigung.

Eine Leistungssteigerung ohne Erhöhung der Taktrate lässt sich erzielen, indem die Größe der Datenwörter im Prozessor erhöht wird. AMD, IBM, Intel und SUN bieten Mikroprozessoren mit 64-Bit-Architektur an. Während Intel mit dem Itanium einen völlig neuen Instruktionssatz etablieren wollte, setzte AMD mit dem AMD64 auf eine Kompatibilität zum IA32-Instruktionssatz und zwang so Intel, ebenfalls 64-Bit-Prozessoren einzuführen, die zu IA32 kompatibel sind. Neben einer besseren Verarbeitungen großer Datenmengen erlaubt die 64-Bit-Architektur auch die Nutzung größerer Adressräume.

Eine weitere Möglichkeit, die ungenutzten Befehlsfächer zu füllen und damit die Verarbeitungsleistung zu erhöhen, ohne die Taktrate steigern zu müssen, besteht darin, zusätzlich grobkörnige Parallelität auf Kontrollfadenebene (*Threads*) oder Prozessebene zu nutzen. Dies kann durch die mehrfädige Prozessortechnik oder durch Mehrkern-Prozessoren geschehen. Beide Varianten werden schon heute – auch in Kombination – für Desktop-Prozessoren eingesetzt. So vereinte bereits der IBM Power-4-Prozessor zwei vollständige Prozessoren auf einem Chip. Der Intel Pentium 4 *Hyperthreading*, der seit Ende 2002 erhältlich war, war zweifädig. Der Intel Core Duo und der AMD 64 X2 enthalten je zwei Prozessoren auf einem Chip. Der Intel Core 2 Quad umfasst vier Prozessorkerne. Im Server-Bereich ist der SUN UltraSPARC T1 Prozessor zu erwähnen, der acht vierfädige SPARC-Prozessoren auf einem Chip enthält.

Mehrkern-Prozessoren

Doch die Erhöhung der Verarbeitungsleistung ist nicht mehr allein das vorrangige Entwurfsziel. Ein geringer Energiebedarf pro Instruktion ist ein zunehmend wichtiger werdendes Optimierungsziel zukünftiger Prozessoren für tragbare, Batterie-getriebene Geräte – wie z.B. Laptops, PDAs (*Personal Digital Assistants*) und Mobiltelefone (Handys).

In den nächsten Abschnitten werden Prozessortechniken für zukünftige Mikroprozessoren vorgestellt, die den Fokus der Forschung in den letzten Jahren bildeten und teilweise bereits ihren Eingang in die kommerzielle Fertigung gefunden haben[11]. Diese Prozessortechniken sind gemäß dreier grundlegender Trends geordnet:

- Durchsatzerhöhung *eines* Kontrollfadens (*Thread*) durch mehr spekulative Parallelität auf Befehlsebene, d.h. durch eine bessere Sprungvorhersage, aber auch durch Datenabhängigkeits- und Wertespekulation. Der bereits beschriebene Trace Cache kann verbessert und im Hinblick auf eine Vorhersage des als nächstes auszuführenden Traces ausgebaut werden. All diese Techniken werden bei Ansätzen der vielfach superskalaren Prozessoren vereint.

- Durchsatzerhöhung *mehrerer* Kontrollfäden durch Nutzung von grobkörniger Parallelität und Befehlsebenen-Parallelität (vgl. Abschnitt 3.2.3). Dazu gehören die Techniken der Mehrkernprozessoren, der Prozessor-Speicher-Integration (vgl. Unterabschnitt 3.2.3.2) und der „explizit“ mehrfädigen Prozessoren.

- Die Techniken der Kontrollfadenspekulation vereinen beide Ansätze mit dem Ziel, wiederum den Durchsatz *eines* Kontrollfadens zu erhöhen. Bei

[11] vgl. insbesondere den zweitgenannten Trend

diesen „implizit“ mehrfädigen Prozessortechniken wird ein Programm dynamisch in Befehlsfolgen zerlegt und diese werden spekulativ parallel zueinander ausgeführt. Aus Platzgründen können wir auf die Kontrollfadenspekulation nicht näher eingehen.

3.2.3 Erhöhung des Durchsatzes einer mehrfädigen Last

3.2.3.1 Mehrkernprozessoren

Ein Mehrkernprozessor, manchmal auch als Chip-Multiprozessor (*Chip Multiprocessor – CMP*) bezeichnet, vereint mehrere Prozessoren auf einem Chip. Jeder der Prozessoren kann die Komplexität eines heutigen Mikroprozessors besitzen und eigene Primär-Cache-Speicher (*Primary Cache*) für Code und Daten haben. Die Prozessoren auf dem Chip sind als speichergekoppelte Multiprozessoren mit gemeinsamem Adressraum organisiert. Abbildung 3.12 demonstriert die möglichen Organisationsformen. Reale Mehrkernprozessoren unterscheiden sich z.B. darin, ob der gemeinsame Sekundär-Cache (*Secondary Cache*) mit auf dem Prozessorchip integriert oder extern vorhanden ist. Der Hauptspeicher (*Global Memory*) ist in jedem Fall extern realisiert.

Untersuchungen zeigten, dass die Organisationsform eines gemeinsamen Sekundär-Cache-Speichers denjenigen gemeinsamer Primär-Cache-Speicher oder gemeinsamer Hauptspeicher überlegen ist. Deshalb kommt meist die Organisationsform (b) in Abbildung 3.12 mit einem großen gemeinsamen Sekundär-Cache-Speicher *auf* dem Prozessor-Chip zum Einsatz. Cache-Kohärenzprotokolle, wie sie bei symmetrischen Multiprozessoren verwendet werden (vgl. das MESI-Protokoll in Unterabschnitt 3.1.3.7), sorgen für einen korrekten Zugriff auf gemeinsame Speicherzellen innerhalb und außerhalb des Prozessor-Chips durch die Prozessoren.

Ein frühes Beispiel eines Mehrkernprozessors ist der TMS320C80-Prozessor der Firma Texas Instruments, der vier digitale Signalprozessoren und einen (skalaren) RISC-Prozessor auf einem Prozessor-Chip vereint.

IBM Power4

Der IBM Power4 ist ein symmetrischer Multiprozessor mit zwei 64-Bit-Prozessoren mit jeweils 64 kB Code- und 32 kB Daten-Caches. Die zwei Prozessoren teilen sich einen einheitlichen 1,41 MB großen Sekundär-Cache-Speicher auf dem Chip. Der Cache-Controller für einen Tertiär-Cache, der als zusätzlicher Chip vorgesehen ist, befindet sich ebenfalls noch auf dem Prozessor-Chip. Der Chip umfasst 174 Millionen Transistoren und arbeitet mit einer Taktrate von 1,6 GHz. Vier Power4-Chips werden in einem Mehr-Chip-Modulgehäuse (*Multi-Chip Module* – MCM) vereint. Vier MCMs können wiederum zu einem symmetrischen Multiprozessor mit 32 Prozessoren verbunden werden.

3.2.3.2 Prozessor-Speicher-Integration

Die Prozessor-Speicher-Integration, auch PIM (*Processor in Memory*) oder IRAM (*Intelligent RAM*) genannt, vereint einen oder mehrere Prozessoren mit einem dynamischen Schreib/Lesespeicher (DRAM) auf einem Chip. Die Zugriffsrate und die Zugriffslatenz von DRAM-Bausteinen soll dadurch an die

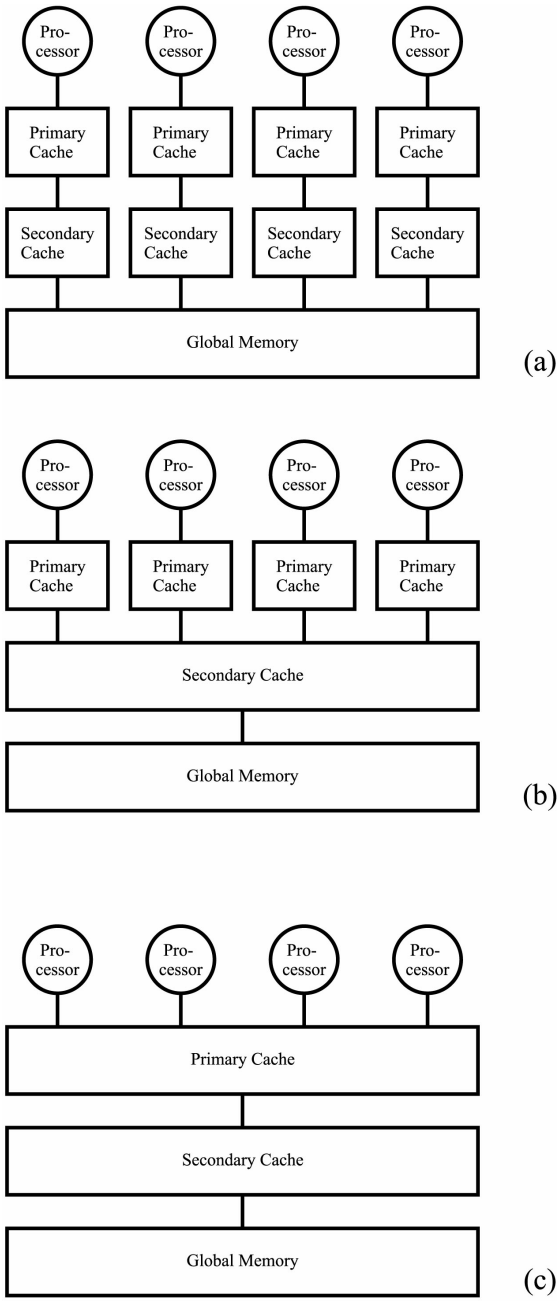


Abbildung 3.12: Mehrkernprozessor, (a) verteilte Caches, (b) gemeinsamer Sekundär-Cache, (c) gemeinsame Caches.

hohe Prozessorgeschwindigkeit angepasst werden. Im Idealfall kommt der Entwurf ohne Cache-Speicher und nur mit DRAM-Speicher aus. Damit wird die Speicherredundanz durch Cache-Speicher vermieden, denn für jedes Wort in einem Cache-Speicher wird bei heutigen Rechnern ein Speicherplatz im Hauptspeicher freigehalten.

Die Prozessor-Speicher-Integration führt zu einem speicherzentrierten Entwurf, bei der der Speicher und nicht der Prozessor im Mittelpunkt des Entwurfs steht. Bei nicht genügend Speicherplatz werden zusätzliche Prozessor-Speicher-

Chips in ähnlicher Weise verwendet, wie heute eine externe Speichererweiterung erfolgt.

Das wesentliche Problem für die Prozessor-Speicher-Integration besteht jedoch in der VLSI-Technologie: Prozessor- und Speichertechnologie sind derzeit schwer vereinbar. Mikrocontroller und sog. Ein-Chip-Systeme (*System on a Chip* – SoC), die oftmals einen Prozessorkern mit DRAM-Speicher auf dem Chip vereinen, erreichen bei weitem nicht die hohen Taktraten heutiger Hochleistungsprozessoren.

Ein-Chip-Systeme

3.2.3.3 Mehrfädige Prozessoren

Mehrfädigkeit Eine wichtige Aufgabe beim Prozessorentwurf ist die Reduzierung von Verzögerungszeiten, den sog. Latenzzeiten (*Latencies*). Diese entstehen insbesondere bei Speicherzugriffen. Die Speicherlatenz, d.h. die Zeit, die vergeht, bis ein benötigtes Datum von einem Speicher geliefert wird, kann durch die Verwendung von Cache-Speichern, Speicherpuffern und breiten Verbindungswegen im Mittel stark verringert werden. Jedoch bleibt eine solche Latenzzeit bei Cache-Fehlzugriffen bestehen.

Latenzzeit

Im Fall eines Zugriffs auf einen nicht lokalen Speicher bei einem speichergekoppelten Multiprozessorsystem wird die Speicherlatenz jedoch um die Übertragungszeit durch das Kommunikationsnetz oder über den Multiprozessorbus erhöht. Diese Übertragungszeit dauert bis zu einigen Zehnerpotenzen länger als die Zeit für die Ausführung eines Maschinenbefehls. Derart hohe Speicherlatenzen können durch Cache-Techniken nicht mehr überbrückt werden.

Latenzzeiten entstehen weiterhin bei der Synchronisation von parallelen Kontrollfäden, wenn eine Synchronisationsbedingung nicht erfüllt ist und einer der Kontrollfäden warten muss.

Die Zeit, die der Prozessor auf einen nicht lokalen Speicherzugriff oder auf das Eintreten einer Synchronisationsbedingung warten muss, führt entweder zum Leerlauf (*Idle Time*) des Prozessors oder der Prozessor muss auf einen anderen Kontrollfaden umschalten. Leerlauf vergeudet bei längeren Wartezeiten wertvolle Ressourcen. Ein Wechsel des Kontrollfadens (*Thread*-Wechsel) führt durch den hohen Verwaltungsaufwand, den die meisten heute verfügbaren Mikroprozessoren dafür benötigen, zu einem Effizienzverlust. Eine Lösung dafür bietet die mehrfädige Prozessortechnik.

Ein **mehrfädiger** (*multithreaded*) **Prozessor** speichert die Kontexte mehrerer Kontrollfäden in separaten Registersätzen auf dem Prozessor-Chip und kann Befehle verschiedener Kontrollfäden gleichzeitig (oder zumindest überlappt parallel) in der Prozessor-Pipeline ausführen. Insbesondere können Latenzzeiten, die durch Cache-Fehlzugriffe, lang laufende Operationen oder sonstige Daten- oder Steuerflussabhängigkeiten entstehen, durch Befehle eines anderen Kontrollfadens überbrückt werden. Die Prozessorauslastung steigt, und der Durchsatz einer Last aus mehreren Kontrollfäden wird erhöht.

Derartige mehrfädige Prozessoren, bei denen ein Kontrollfaden einem Betriebssystem-Thread oder Prozess entspricht, werden im Folgenden als **explizit mehrfädig** bezeichnet. Explizit mehrfädige Prozessoren sind für die Ausführung einer Last paralleler Threads oder Prozesse entworfen.

Als **implizit mehrfädig** werden Prozessorentwürfe für zukünftige Prozessoren bezeichnet, deren Ziel es ist, spekulativ parallele Kontrollfäden aus einem einzigen sequenziellen Programm zu erzeugen und diese simultan zueinander auszuführen. In den nächsten Abschnitten werden die explizit mehrfädigen Prozessortechniken vorgestellt. Auf die implizit mehrfädigen Architekturansätze aus der Forschung kann im Rahmen dieses Kurses leider nicht eingegangen werden.

Grundtechniken der Mehrfädigkeit Nach der Art, wie Befehle, die aus mehreren Kontrollfäden stammen, auf einem Prozessor zur Ausführung ausgewählt werden, können bei der mehrfädigen Prozessortechnik zwei Grundtechniken unterschieden werden:

- Bei der ***Cycle-by-Cycle*-Interleaving-Technik**, die oft auch als *Fine-Grain Multithreading* bezeichnet wird, wechselt der Kontext (*Context Switch*) mit jedem Prozessortakt. Eine Anzahl von Kontrollfäden ist auf dem Prozessor geladen, ein Hardware-Scheduler wählt in jedem Takt einen der „ausführbereiten" Kontrollfäden aus. Von diesem wird der in der Programmreihenfolge nächste Befehl in die Befehls-Pipeline eingefüttert. In aufeinander folgenden Takten wird jeweils ein Befehl aus einem anderen Kontrollfaden ausgewählt. Ein Kontrollfaden ist erst dann wieder „ausführbereit", wenn keiner seiner Befehle sich noch in der Befehls-Pipeline befindet.

- Bei der ***Block-Interleaving*-Technik**, die auch als *Coarse-Grain Multithreading* bezeichnet wird, werden die Befehle *eines* Kontrollfadens so lange wie möglich direkt aufeinander folgend ausgeführt, bis ein Ereignis eintritt, das zum Warten des Prozesses bzw. Kontrollfadens zwingt. Ausgelöst durch ein solches Kontextwechselereignis (z.B. einen Cache-Fehlzugriff) wird ein Thread-Wechsel durchgeführt.

Abbildung 3.13 demonstriert die grundlegenden Unterschiede beider Techniken bei der Ausführung von vier Kontrollfäden auf einem vierfädigen skalaren Prozessor. Die Abbildung zeigt die Befehlsfächer (vgl. Abschnitt 3.2.2) der Zuordnungsstufe einer hypothetischen Pipeline-Stufe. Die Zahlen in den Klammern bezeichnen Befehle verschiedener Kontrollfäden. Leere Fächer, d.h. leere Kästchen in der Abbildung 3.13, bedeuten Leertakte in der Pipeline. Wie man sieht, können Leertakte, die bei einem konventionellen Prozessor auftreten, bei mehrfädigen Prozessoren durch Ausführung von Befehlen anderer Kontrollfäden genutzt werden. Bei der Block-Interleaving-Technik ergibt sich jedoch ein kleiner Kontextwechselaufwand von in der Regel ein bis mehreren Takten, der in der Abbildung durch zwei leere Kästchen verdeutlicht ist.

Die beiden Grundtechniken der Mehrfädigkeit sind jedoch nicht nur für skalare Prozessoren, sondern insbesondere auch für VLIW-Prozessoren anwendbar. Abbildung 3.14 zeigt die Cycle-by-cycle-Interleaving-Technik unter Annahme eines vierfach superskalaren Prozessors bzw. eines vierfachen VLIW-Prozessors. Falls in einem Prozessortakt kein Befehl zugeordnet werden kann, so bleiben

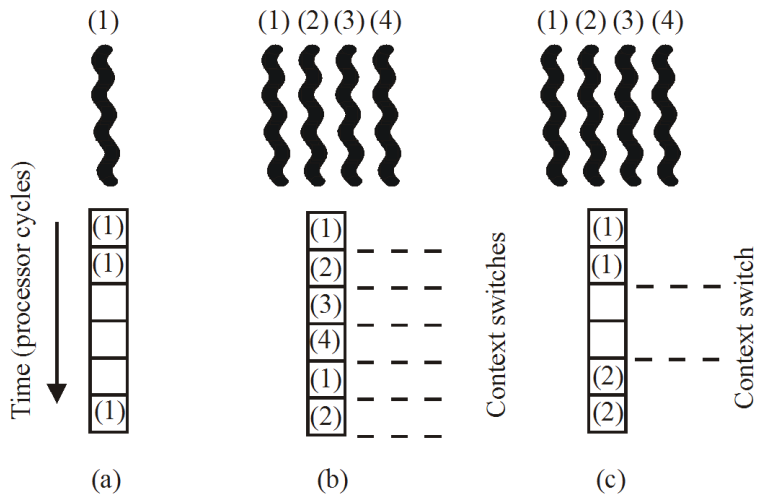


Abbildung 3.13: (a) konventioneller, einfädiger und skalarer Prozessor, (b) skalarer Prozessor mit Cycle-by-cycle-Interleaving-Technik, (c) skalarer Prozessor mit Block-Interleaving-Technik.

die vier horizontal angeordneten Befehlsfächer frei. Dies wird als vertikaler Verlust bezeichnet. Allerdings kann es auch passieren, dass wegen eines Mangels an gleichzeitig ausführbaren Befehlen nicht die gesamte Zuordnungsbandbreite genutzt werden kann. In diesem Fall bleiben beim Superskalarprozessor die zugehörigen Befehlsfächer frei oder werden beim VLIW-Prozessor durch Leerbefehle (N für Noop-Befehle in der Abbildung) aufgefüllt. Dies wird horizontaler Verlust genannt). Wie man sieht, kann die Cycle-by-cycle-Interleaving-Technik (wie auch die Block-Interleaving-Technik) zwar die vertikalen, jedoch nicht die horizontalen Verluste ausgleichen. Zur zusätzlichen Vermeidung der horizontalen Verluste müsste der Prozessor pro Takt auch Befehle aus mehreren Kontrollfäden zuweisen können. Dies ist bei der im nächsten Unterabschnitt vorgestellten simultan mehrfädigen Prozessortechnik der Fall.

vertikaler Verlust

horizontaler Verlust

Block Interleaving benötigt für einen Thread-Wechsel meist einige Takte mehr als Cycle-by-cycle Interleaving. Der Vorteil der ersten Technik gegenüber der zweiten ist die höhere Leistung bei der Ausführung eines *einzelnen* Kontrollfadens, da dessen Befehle direkt aufeinander folgend ausgeführt werden, während beim Cycle-by-cycle Interleaving bei einer Befehls-Pipeline von n Stufen nur in jedem n-ten Takt ein Befehl desselben Kontrollfadens zur Ausführung kommt. Zur Überbrückung von Speicherlatenzen sollten demnach möglichst genauso viele Kontrollfäden wie die für den Speicherzugriff benötigte Taktzahl auf dem Prozessor zur Verfügung stehen.

Falls jedoch nicht genügend Kontrollfäden als Last vorhanden sind, bleibt ein Teil der Befehlsfächer eines Cycle-by-cycle-Interleaving-Prozessors leer. Falls überhaupt nur ein einziger Kontrollfaden als Last zur Verfügung steht, so entspricht die Ausführungszeit dieses Kontrollfadens derjenigen eines Prozessors ohne Pipelining. Eine *Explicit-Dependence Lookahead* genannte Technik erlaubt es jedoch bei bestimmten Cycle-by-cycle-Interleaving-Prozessoren, auch eine begrenzte Anzahl von Befehlen desselben Kontrollfadens überlappend auszu-

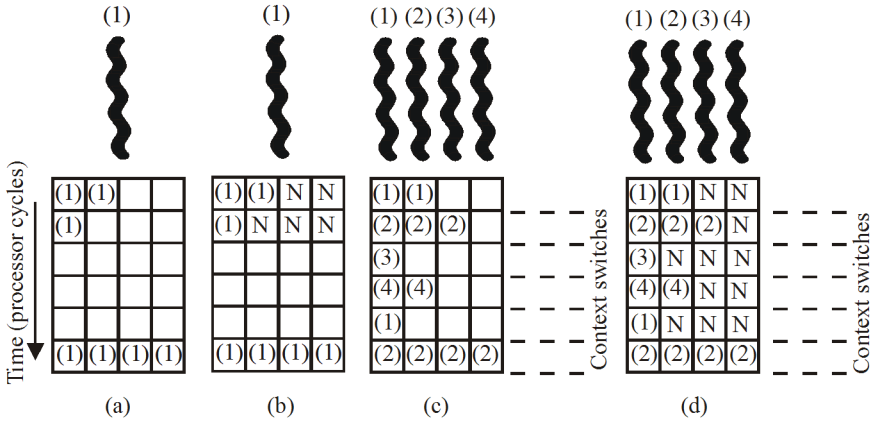


Abbildung 3.14: (a) vierfach Superskalarprozessor, (b) vierfach VLIW-Prozessor, (c) vierfach Superskalarprozessor mit Cycle-by-cycle-Interleaving-Technik, (d) vierfach VLIW-Prozessor mit Cycle-by-cycle-Interleaving-Technik.

führen. Dabei kann der Compiler im Opcode jeden Befehls die Anzahl der von diesem Befehl unabhängigen Folgebefehle angeben. Diese Information wird vom Scheduler der Befehlsbereitstellung genutzt, um bei zu geringer Last auch unabhängige Befehle desselben Kontrollfadens in die Pipeline einzufüttern, ohne die Beendigung der Ausführung eines vorherigen Befehls des Kontrollfadens abwarten zu müssen.

Ein Vorteil des Cycle-by-Cycle Interleaving gegenüber dem Block Interleaving ist der Thread-Wechsel nach jedem Takt. Ein Leeren der Pipeline entfällt beim Thread-Wechsel, und so kann auch eine Prozessor-Pipeline mit hoher Stufenzahl effizient genutzt werden. Da ein Nachfolgebefehl eines Kontrollfadens erst nach Beendigung seines Vorgängers in die Pipeline eingefüttert wird, wirken sich die Kontroll- und Datenabhängigkeiten einer Befehlsfolge nicht aus. Die komplexe Hardware-Logik zur Überwachung dieser Abhängigkeiten kann eingespart werden.

Vergleichende Untersuchungen ergaben, dass Block Interleaving dem Cycle-by-cycle Interleaving meist überlegen ist. Simulationen der Block-Interleaving-Technik zeigten, dass eine erhebliche Steigerung der Prozessorauslastung bei *zwei* bis *vier* Kontrollfäden pro Prozessor erreicht wird. Eine weitere Erhöhung der Anzahl der Kontrollfäden bringt nur noch wenig Gewinn oder gar eine Verschlechterung der Prozessorauslastung.

Mehrfädige Prozessoren mit Block-Interleaving-Technik lassen sich gemäß der Auslösebedingung des Kontextwechsels wie in Abbildung 3.15 dargestellt klassifizieren. Danach unterscheidet man:

- *Statisches Block Interleaving*: Ein Kontextwechsel wird in Abhängigkeit von bestimmten Befehlen bei jeder Ausführung eines solchen Befehls ausgelöst. Hier steht der Kontextwechsel bereits beim Übersetzen fest und ist im Befehl codiert. Dabei gibt es noch folgende Unterscheidungen:
  - *Explicit-Switch*: Es existiert ein expliziter Kontextwechselbefehl, der den Kontextwechsel auslöst.

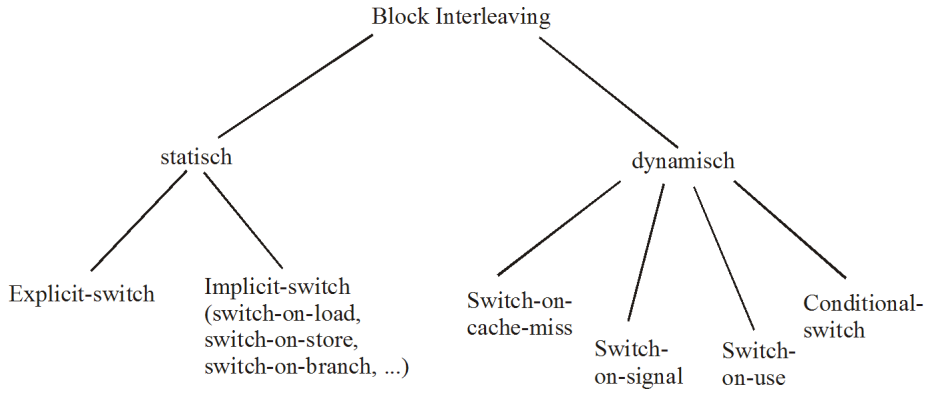


Abbildung 3.15: Klassifizierung von Block-Interleaving-Techniken.

  - *Implicit-Switch*: Die Zugehörigkeit zu einer Befehlsklasse löst den Kontextwechsel aus. In Abhängigkeit von der Befehlsart kann noch weiter in *Switch-on-Load*, *Switch-on-Branch* etc. unterschieden werden.

- *Dynamisches Block Interleaving*: Hier hängt das den Kontextwechsel auslösende Ereignis vom dynamischen Programmablauf ab. Man unterscheidet:

  - *Switch-on-Cache-Miss*: Ein Kontextwechsel wird von einem Cache-Fehlzugriff ausgelöst.
  - *Switch-on-Signal*: Ein Kontextwechsel wird von einem externen oder internen Signal, d.h. von einem Interrupt oder Trap, ausgelöst.
  - *Switch-on-Use*: Der Kontextwechsel wird nicht sofort ausgelöst, sondern erst, wenn der Operand vom Befehl benötigt wird. Dies entspricht der Switch-on-Cache-Miss-Strategie mit verzögerter Ausführung.
  - *Conditional-Switch*: Verknüpfung der Explicit-Switch-Strategie mit einer Bedingung.

Der Vorteil der statischen Block-Interleaving-Technik ist, dass der Kontextwechsel bereits in der Befehlsbereitstellungsstufe der Pipeline erkannt und durchgeführt werden kann. Durch eine geschickte Codierung, die es erlaubt, kontextwechselauslösenden Befehle bereits in der Befehlsbereitstellungsstufe als solche zu erkennen, kann der Kontextwechselaufwand verringert werden. Der Kontextwechselaufwand entspricht somit einem einzigen Takt, falls der Befehl, der den Kontextwechsel auslöst, selbst nicht weiter ausgeführt wird (z.B. bei *Explicit-Switch*-Befehlen), und null Takten, falls der auslösende Befehl weiterverwendet wird (*Implicit-Switch*-Befehle). Die Anwendung eines Kontextwechselpuffers, der die Adressen von kontextwechselauslösenden Befehlen zur Laufzeit speichert, kann den Aufwand auch im ersten Fall auf nahezu null drücken.

Bei der dynamischen Block-Interleaving-Technik müssen alle bereits in der Pipeline befindlichen Befehle nach dem kontextwechselauslösenden Befehl gelöscht werden. Im Falle der Switch-on-Cache-Miss-Technik sind dies alle Befehle zwischen der Befehlsbereitstellungsstufe und der Speicherzugriffsstufe. Daraus ergibt sich ein Kontextwechselaufwand von mehreren Takten, also ein höherer Aufwand als bei den statischen Techniken. Allerdings wird bei den dynamischen Techniken ein Kontextwechsel immer nur dann ausgelöst, wenn dieser notwendig ist, während bei den statischen Techniken Kontextwechsel prinzipiell ausgelöst werden.

Die beiden Multithreading-Techniken wurden für skalare Prozessoren entwickelt. Sie sind im Prinzip mit der Superskalar- oder der VLIW-Technik kombinierbar. Beispielsweise könnten durch den Hardware-Scheduler beim Cycle-by-Cycle Interleaving pro Takt auch mehrere Befehle *eines* Kontrollfadens oder sogar mehrere Befehle aus *mehreren* Kontrollfäden ausgewählt und in die Befehls-Pipeline eingefüttert werden. Auch beim Block Interleaving könnten den Ausführungseinheiten Befehle aus mehreren Kontrollfäden gleichzeitig zugewiesen werden.

Simultan mehrfädige Prozessortechnik Die Kombination der Mehrfädigkeit mit der Superskalartechnik stellt eine dritte mehrfädige Prozessortechnik dar, die wohl als die zukunftsträchtigste betrachtet werden kann. Ein **simultan mehrfädiger Prozessor** (*Simultaneous Multithreading – SMT*) kann Befehle mehrerer Kontrollfäden in *einem* Taktzyklus den Ausführungseinheiten zuordnen. Wie bei der mehrfädigen Prozessortechnik üblich, ist jedem Befehlsstrom ein eigener Registersatz zugeordnet.

Vergleicht man die simultan mehrfädige Prozessortechnik mit den Mehrkernprozessoren (s. Abbildung 3.16) so zeigt sich, dass die simultan mehrfädige Technik im Idealfall alle Latenzen durch Befehle anderer Kontrollfäden füllen kann, im Gegensatz zum Mehrkernprozessor, der in jedem Prozessor Latenzzeiten aufweist. Nicht nur die vertikalen, sondern auch die horizontalen Verluste können verringert, im Idealfall sogar vollständig durch sinnvolle Befehle ausgeglichen werden.

Bei der simultan mehrfädigen Prozessortechnik werden alle Pipeline-Ressourcen – bis auf die Registersätze und die Befehlszähler – von den Kontrollfäden gemeinsam genutzt. Die einzelnen Befehle werden in der Pipeline mit *Tags* versehen, um ihre Zugehörigkeit zu den verschiedenen Kontrollfäden zu bezeichnen. Dadurch wird eine sehr geringe zusätzliche Hardware-Komplexität gegenüber einem Superskalarprozessor benötigt. Zusätzliche Komplexität ergibt sich insbesondere in der Befehlsbereitstellungsstufe, die fähig sein muss, Befehle verschiedener Kontrollfäden bereitzustellen, und in der Rückordnungsstufe, die eine den Kontrollfäden entsprechende selektive Rückordnung durchführen muss.

Die Befehlsbereitstellungseinheit kann ihre Bandbreite auf zwei oder mehr Kontrollfäden aufteilen und z.B. zweimal vier Befehle verschiedener Kontrollfäden aus dem Code-Cache holen, oder sie kann mit jedem Takt Befehle eines anderen Kontrollfadens in die Pipeline laden. Es wurden mehrere selektive Befehlsholestrategien untersucht, wie z.B. diejenigen, die bevorzugt Befehle von

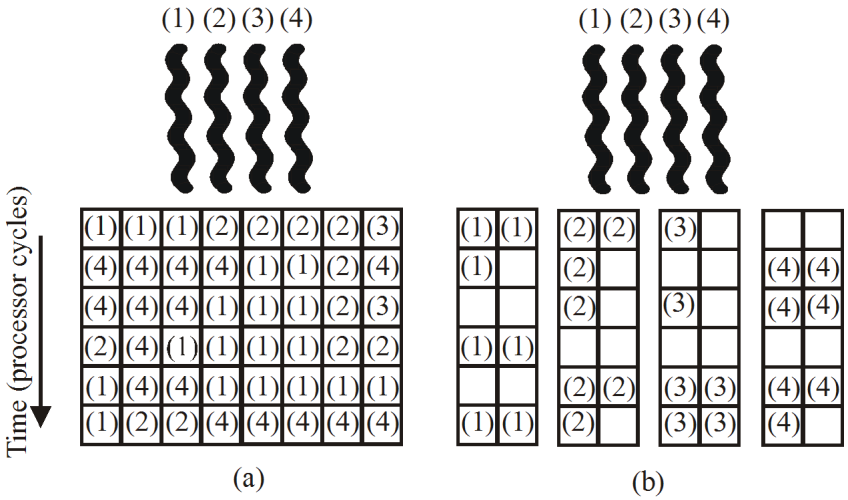


Abbildung 3.16: Befehlszuordnungen *(Issue Slots)* eines vierfädigen, achtfach superskalaren SMT-Prozessors (links) und eines Mehrkernprozessors aus vier je zweifach superskalaren Prozessoren (rechts); die Nummern stehen für die Kontrollfäden, leere Kästchen für ungenutzte Befehlsfächer der Zuordnungsbandbreite

Kontrollfäden laden, die keine Cache-Fehlzugriffe aufweisen oder deren Befehle nur eine geringe Spekulationstiefe aufweisen. Als beste Strategie hat sich dabei die ICOUNT genannte Kombinationsstrategie erwiesen, die den Füllstand verschiedener Befehlspuffer als entscheidendes Kriterium ausweist.

Mehrfädig superskalare Prozessortechnik Im Gegensatz zur simultan mehrfädigen Technik geht die als mehrfädig superskalar bezeichnete Technik von einem mehrfach superskalaren Mikroprozessor aus, dessen Ausführungseinheiten durch *eine* Zuordnungseinheit nicht nur aus einem, sondern aus mehreren Befehlspuffern gefüttert werden. Alle internen Puffer werden für die vorgesehenen Kontrollfäden vervielfacht. Jeder Befehlspuffer zwischen zwei Pipeline-Stufen repräsentiert einen anderen Befehlsstrom. Wie bei der mehrfädigen Prozessortechnik üblich, ist jedem Befehlsstrom ein eigener Registersatz zugeordnet.

Bei dieser Organisationsform der Ressourcenreplizierung wird mehr Chip-Fläche als bei der simultan mehrfädigen Technik benötigt, jedoch sind die einzelnen Kontrollfäden stärker voneinander entkoppelt. Damit werden insbesondere die Zuordnungs- und die Rückordnungsstufen vereinfacht und potenziell zu höheren Taktraten fähig gemacht als bei den gemeinsamen, großen Puffern, die bei der simultan mehrfädigen Technik vorgesehen sind.

Ansonsten sind beide Techniken weitgehend gleichartig. Simulationen zeigen, dass die mehrfädig superskalare Prozessortechnik durch ihre Fähigkeit der Latenzzeitüberbrückung eine zwei- bis dreifache Leistungssteigerung gegenüber vergleichbaren (einfädigen) Superskalarprozessoren erreichen kann. Mehrfädig superskalare Prozessoren sind bisher noch nie realisiert worden.

Vergleicht man die simultan mehrfädigen Prozessoren mit den Mehrkernprozessoren, so erscheint der Wettbewerb um die höhere Leistung noch unentschieden. Die simultan mehrfädige Prozessortechnik füllt die Befehlsfächer eines superskalaren Prozessors, überbrückt Cache-Latenzen sehr gut und bringt schon mit zwei Kontrollfäden einen deutlichen Gewinn. Sie ermöglicht bei acht Kontrollfäden und achtfacher Zuordnungsbandbreite eine bis zu dreifache Beschleunigung gegenüber einem einzelnen superskalaren Prozessor. Der Latenzzeitnutzung bei mehrfädig superskalaren Prozessoren steht eine einfachere Implementierbarkeit der Mehrkernprozessoren gegenüber. Die Verträglichkeit der mehrfädig superskalaren Prozessortechnik mit hohen Taktraten ist allerdings noch kaum erforscht. Derzeit scheint sich außerdem die Kombination der Mehrfädigkeit mit der Mehrkernprozessortechnik durchzusetzen. Um eine weitere Erhöhung der Taktraten möglich zu machen, müssen dafür kritische Einheiten des Prozessors dezentralisiert werden, was den Mehrkernprozessoransatz bevorzugt, und andererseits soll die höhere Flexibilität der simultan mehrfädigen Prozessortechnik beibehalten werden.

3.2.4 Abschließende Bemerkungen

Bei den Hochleistungsprozessoren erscheint Superskalartechnik wegen mangelnder Befehlsebenen-Parallelität schon heute weitgehend ausgereizt. Die nächste Prozessorgeneration wird trotzdem versuchen, den Durchsatz *eines* Kontrollfadens durch noch mehr Spekulation und die Technik des Trace Caches zu erhöhen. Während der Trace Cache unumstritten erscheint, ist der Nutzen von Datenabhängigkeits- und Wertespekulation noch unklar. Eine Alternative ist, den Durchsatz aus *mehreren* Kontrollfäden (Betriebssystem-Threads oder Prozessen) zu erhöhen. Ob dies am effizientesten durch Mehrkernprozessoren oder durch mehrfädig superskalare Prozessoren geschieht, ist noch unentschieden. Häufig wird Mehrfädigkeit auch in Mehrkernprozessoren verwendet. Die Kontrollfadenspekulation ist noch wenig erforscht.

Bei heutigen Prozessoren ist der Prozessor-Speicher-Engpass im Entwurf vorherrschend. Eine heute noch unklare Sonderstellung haben die Techniken der Prozessor-Speicher-Integration sowie die dynamisch rekonfigurierbaren Prozessoren inne. Hier sind technologische Durchbrüche in der Chip-Fertigung nötig, um diesen Techniken zu einem allgemeinen Erfolg zu verhelfen.

Im Bereich der Mikrocontroller finden wir eine breite Palette verfügbarer Einheiten für nahezu alle nur erdenklichen Leistungsklassen. Hierbei ist zu beobachten, dass Architektur und Implementierung bei Prozessorkernen von industriellen Mikrocontrollern etwa 5 Jahre hinter dem Stand von universellen Mikroprozessoren zurückliegen. Dies ist im Wesentlichen durch das Anwendungsspektrum sowie durch Kostenoptimierungen begründet.

Neue Forschungsansätze bei Mikrocontrollern konzentrieren sich im wesentlichen auf Energiespartechniken, die Weiterentwicklung der Grundideen von Mikrocontrollern zu *Systems-on-a-Chip* (SoC) sowie auf den Einsatz innovativer Konzepte für Echtzeitanwendungen. Gerade Echtzeitanwendungen repräsentieren ein Kernanwendungsfeld von Mikrocontrollern, welches viel Spielraum für die Forschung erlaubt.

Man sieht, dass noch viele Ideen zu Architektur- und Implementierungstechniken erforscht und für eine Leistungssteigerung in zukünftigen Mikroprozessoren und Mikrocontrollern eingesetzt werden können. Neue Denkanstöße werden sich zudem durch die Marktreife neuer, z.B. optischer Chip-zu-Chip-Übertragungstechnologien ergeben. Die in diesem Kurs diskutierten Techniken sind somit nur eine kleine Auswahl, die wir in zukünftigen Mikroprozessoren und Mikrocontrollern erwarten können.

Selbsttestaufgabe 3.8 *(Kontrollfragen)*

a) Wie unterscheidet sich die North Bridge von der South Bridge in einem PC-System?

b) Was ist das gemeinsame Ziel der neuen Prädiktoransätze?

c) Welche Arten von Datenabhängigkeits-, Adress- und Wertespekulationen gibt es?

d) Was unterscheidet die Cycle-by-cycle-Technik von der Block-Interleaving-Technik und beide von der simultan mehrfädigen Prozessortechnik?

e) Wie unterscheiden sich die Ziele der Superskalar-, VLIW-, EPIC-, Mehrkernprozessortechnik und der simultan mehrfädigen Prozessortechnik bzgl. der Programmbeschleunigung, d.h. der Beschleunigung eines einzelnen Programms oder eines Programmmixes?

3.3 Lösungen der Selbsttestaufgaben

Selbsttestaufgabe 3.1 von Seite 132

Eine vereinfachte RISC-Maschine verfüge über acht globale Register R0–R7 und über achtzig weitere Register als Umlaufspeicher für die überlappenden Registerfenster. Ein Fenster umfasse acht *In*-Register R8–R15, acht lokale Register R16–R23 und acht *Out*-Register R24–R31. Die Maschine besitze außerdem ein Statusregister SR und ein Register für den Zeiger auf das aktuelle Fenster CWP.

a) Schreiben Sie für die Funktion f(x), die die Fibonacci-Zahlen berechnet, f(x) = if x<=1 then x else f(x-1)+f(x-2)
ein Programm für den gegebenen RISC-Prozessor, wobei die globalen Register R0 mit dem Wert 0 und R1 mit dem konstanten Wert 1 vorbelegt seien. Der Parameter x soll vom aufrufenden Assemblerprogramm im Register R25 übergeben werden und das Ergebnis soll nach Abarbeitung der zu implementierenden Rechenvorschrift im Register R26 zu lesen sein. Das Register R24 diene zur Aufnahme der Rückkehradresse.

Lösung:
Registerfenster:

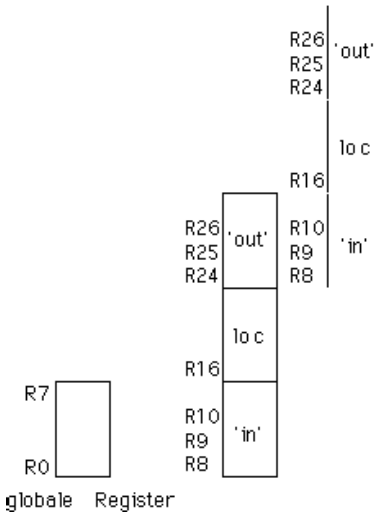


Abbildung 3.17:

Vorbelegungen:
R0 = 0
R1 = 1
R24 <- Rückkehradresse
R25 <- x
R26 <- erg

```
fib  CMP R9,R1 ; R9 = R25 der aufrufenden Funktion = Eingabewert
     JMP LE,else
then SUB R9,R1,R25
```

```
     CALL R24,fib ;f(x-1)
@1   ADD R0,R26,R16
     SUB R25,R1,R25
     CALL R24,fib ;f(x-2)
@2   ADD R16,R26,R10 ;f(x) = f(x-1) + f(x-2)
     RET R8
else ADD R0,R9,R10 ;f(x) = x
     RET R8 ; R8 = R26 der aufrufenden Funktion = Ausgabewert
```

b) Testen Sie Ihr Programm für f(3) und zeichnen Sie sich die Fensterwechsel für diesen Testlauf auf. Trennen Sie die einzelnen Registertypen voneinander und machen Sie sich die Überlappung der einzelnen Registerfenster klar, indem Sie die Fenster in zeitlicher Reihenfolge nebeneinander anordnen und dieselben Register auf dieselbe Höhe ausrichten. Die Inhalte der verwendeten Register sollen in die Fenster eingetragen werden.

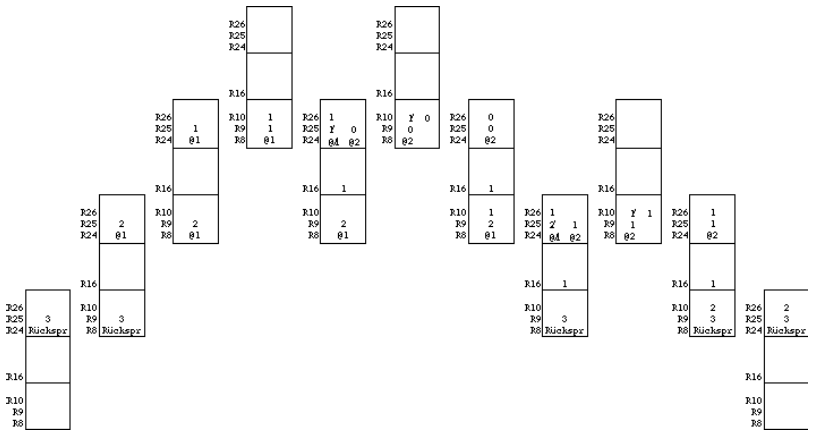


Abbildung 3.18:

c) Für welche Argumentwerte reicht die Gesamtanzahl der Register aus? Was passiert bei größeren Argumentwerten? Welche Techniken könnten Sie sich vorstellen, um auch größere Rekursionstiefen zu ermöglichen?

Registeranzahl ohne globale Register / (Anz. *In*-Reg. + Anz. *Loc*-Reg)
= 80 / (8 + 8) = 5 Registerfenster

Zieht man ein Fenster für das aufrufende Programm ab, so bleiben vier Fenster übrig. Außerdem wird ein Registerfenster als *invalid* gekennzeichnet, da es sich mit dem aufrufenden Programm überschneidet. Somit lässt sich maximal fib(3) berechnen. Für größere Rekursionstiefen müssen Fenster zwischengespeichert werden.

Selbsttestaufgabe 3.6 von Seite 158

Welche der folgenden Maßnahmen tragen dazu bei, die Häufigkeit von Seitenfehlern zu reduzieren?

a) Vergrößerung des Hauptspeichers? Lösung: ja
b) Vergrößerung des Sekundärspeichers? Lösung: nein
c) Verbesserung der Verdrängungsstrategie bei der Auslagerung von Seiten? Lösung: ja

Selbsttestaufgabe 3.7 von Seite 158

Der folgende aus der digitalen Signalverarbeitung stammende Algorithmus soll auf einem Prozessor mit virtueller Speicherverwaltung ausgeführt werden:

```
(01) FOR i = 0 TO 2^13-1
(02) y = 0;
(03) FOR j = 0 TO 2
(04) y = y + x[i+j] * w(j);
(05) END
(06) y[i] = y;
(07) END
```

Der Prozessor verfügt über drei Segmentregister: CS für Programmcode, DS für Datenstrukturen und SS für den Stapel. Die Auswahl eines Segmentregisters erfolgt implizit durch den auszuführenden Befehl. Die Bildung der virtuellen Adresse erfolgt durch Addition der Basisadresse mit der logischen Adresse. Die Seitengröße beträgt 4096 Wörter, wobei der Hauptspeicher maximal 4 Seiten aufnehmen kann. Seiten der CS- und der SS-Segmente werden vom Betriebssystem nicht ausgelagert. Als Ersetzungsstrategie kommt LRU zum Einsatz. Die Segmentregister sind wie folgt belegt:

Segment	Basisadresse	Länge (hex.)
CS	3A17C000	1000
DS	D29E4000	4000
SS	51F60000	1000

Das Feld x im obigen Programm beginnt an der logischen Adresse 2000, y bei 0. Die Seitentabelle enthält zu Beginn folgende Einträge:

Virtuelle Adresse	Physikalische Adresse	Hauptspeicher
3A17C	0	ja
51F60	1	ja
D29E6	2	ja
D29E7	3	ja

a) Erstellen Sie ein Protokoll der ein- und ausgelagerten Seiten während der Ausführung des Programms, solange keine Zugriffsverletzung auftritt.
Geben Sie jeweils die Iteration und die Zeilennummer an, bei der ein Seitenfehler auftritt. Wie viele Seitenwechsel werden insgesamt durchgeführt?

Bei der LRU-Ersetzungsstrategie werden insgesamt 14 Seiten ein- bzw. ausgelagert:

Zeile	i	j	Seite	Aktion	Seite	Aktion
6	0		D29E7	auslagern	D29E4	anlegen -> 3
4	FFE	2	D29E4	auslagern	D29E7	einlagern -> 3
6	FFE		D29E6	auslagern	D29E4	einlagern -> 2
4	FFF	0	D29E7	auslagern	D29E6	einlagern -> 3
4	FFF	1	D29E4	auslagern	D29E7	einlagern -> 2
6	FFF		D29E6	auslagern	D29E4	einlagern -> 3
6	1000		D29E4	auslagern	D29E5	anlegen -> 3
6	1FFE	2	D29E8	Zugriffsverletzung		

b) Gibt es eine Strategie zur Seitenersetzung, die besser ist als LRU? Wenn ja, wie viele Ein-/Auslagerungen müssen dann durchgeführt werden?

Bei der folgenden Ersetzungsstrategie reduziert sich die Anzahl der Ein-/Auslagerungen auf 10:

Zeile	i	j	Seite	Aktion	Seite	Aktion
6	0		D29E7	auslagern	D29E4	anlegen -> 3
4	FFE	2	D29E6	auslagern	D29E7	einlagern -> 2
4	FFF	0	D29E7	auslagern	D29E6	einlagern -> 2
4	FFF	1	D29E6	auslagern	D29E7	einlagern -> 2
6	1000		D29E4	auslagern	D29E5	anlegen -> 3
6	1FFE	2	D29E8	Zugriffsverletzung		

Selbsttestaufgabe 3.2 von Seite 139

Bei einem byteadressierten Rechner seien ein direkt-abgebildeter Cache, ein zweifach satzassoziativer Cache und ein vollassoziativer Cache gegeben. Die drei Cachespeicher haben jeweils eine Speicherkapazität von 32 Bytes und werden in Blöcken von je 4 Bytes geladen. Die Hauptspeicheradresse umfasst 32 Bits.

a) Geben Sie für die drei Cache-Speicher an, wie viele Bits zur Verwaltung (Tag- und Zustandsbits) eines Cache-Blocks benötigt werden. Dabei sollen für den Zustand des Cache-Blocks zwei Statusbits verwendet werden (Valid-Bit und Dirty-Bit).

Lösung:
DM: 29-Bit
A2: 30-Bit
AV: 32-Bit

Betrachten Sie die Folge der Lesezugriffe auf die folgenden, in hexadezimaler Schreibweise angegebenen Hauptspeicheradressen:
$46, $09, $44, $B9, $11, $FB, $55, $F9, $5C, $06

b) Nehmen Sie an, die Caches seien zu Beginn leer. Ermitteln Sie, ob es sich beim Lesezugriff auf die jeweiligen Adressen um einen Treffer (*Cache Hit*) oder einen Fehlzugriff (*Cache Miss*) handelt. Falls notwendig, wird die LRU-Ersetzungsstrategie (*Least Resently Used*) verwendet. Geben Sie die Tag-Zustände des Cache-Speichers nach dem letzten Zugriff an.

Lösung:

Adresse	$46	$09	$44	$B9	$11	$FB	$55	$F9	$5C	$06	Hits
DM-Cache	–	–	X	–	–	–	–	X	–	–	2x
A2-Cache	–	–	X	–	–	–	–	X	–	–	2x
AV-Cache	–	–	X	–	–	–	–	X	–	–	2x

	DM	A2		AV
0		00	0001	010001
1	000			000010
2	000	01	0000	101110
3			0101	000100
4	000	10	1111	111110
5	010		1011	010101
6	111	11	0101	010111
7	010			000001

Selbsttestaufgabe 3.3 von Seite 146

Gegeben seien ein direkt-abgebildeter Cache, ein 4-fach satzassoziativer Cache und ein vollassoziativer Cache. Alle drei Cache-Speicher werden durch 32 Bit adressiert und haben eine Kapazität von 16 Blöcken, wobei ein Block jeweils 4 Datenworte umfasst. Bei den assoziativen Caches wird die LRU-Ersetzungsstrategie verwendet.

Die folgende Sequenz zeigt die hexadezimalen Adressen der ersten zwanzig Lesezugriffe eines Programms zur Multiplikation zweier Matrizen mit je 10000 Zeilen/Spalten:
1F296FFA, 378CF121, 1F296FFB, 378D1831, 1F296FFC, 378D3F41, 1F296FFD, 378D6651, 1F296FFE, 378D8D61, 1F296FFF, 378DB471, 1F297000, 378DDB81, 1F297001, 378E0291, 1F297002, 378E29A1, 1F297003, 378E50B1

a) Geben Sie unter Berücksichtigung der jeweiligen Cache-Organisationsform an, wie viele Adressbits für den Tag- bzw. den Index-Teil benötigt werden. Wie breit ist die Wortadresse?

Lösung:

Die Wortadresse ist bei allen drei Cache-Organisationsformen 2 Bits breit, da ein Block jeweils 4 Datenwörter umfasst. Durch den Index-Teil wird ein Satz ausgewählt, d.h. Länge(Index) = $\log_2$(Anzahl der Sätze). Für den Tag-Teil gilt: Länge(Tag) = 32 – Länge(Index) – Länge(Wort).

Cache	Sätze	Blöcke/Satz	#Index	#Tag
direkt-abgebildet	16	1	4	26
4-fach satzassoziativ	4	4	2	28
vollassoziativ	1	16	0	30

b) Bestimmen Sie für alle drei Cache-Typen, bei welchen Zugriffen auf die oben genannten Adressen es sich um Cache-Treffer handelt. Führen Sie dazu Buch über die Belegungen der einzelnen Cache-Blöcke. Nehmen Sie an, dass die Caches zu Beginn leer sind. Wie hoch sind die Trefferraten?

Lösung:

Die Tabellen links geben an, bei welchen Speicherzugriffen es sich um Treffer handelt, während die Tabellen auf der rechten Seite die Cache-Belegungen widerspiegeln.

Direkt-abgebildeter Cache: Trefferrate: $6/20 = 0{,}3$

Adresse	**Tag**	**Index**	**Treffer**
1F296FFA	1F296F,11	1110	–
378CF121	378CF1,00	1000	–
1F296FFB	1F296F,11	1110	+
378D1831	378D18,00	1100	–
1F296FFC	1F296F,11	1111	–
378D3F41	378D3F,01	0000	–
1F296FFD	1F296F,11	1111	+
378D6651	378D66,01	0100	–
1F296FFE	1F296F,11	1111	+
378D8D61	378D8D,01	1000	–
1F296FFF	1F296F,11	1111	+
378DB471	378DB4,01	1100	–
1F297000	1F2970,00	0000	–
378DDB81	378DDB,10	0000	–
1F297001	1F2970,00	0000	–
378E0291	378E02,10	0100	–
1F297002	1F2970,00	0000	+
378E29A1	378E29,10	1000	–
1F297003	1F2970,00	0000	+
378E50B1	378E50,10	1100	–

Satz	**Tags**			
0000	378D3F,01	1F2970,00	378DDB,10	1F2970,00
0001				
0010				
0011				
0100	378D66,01	378E02,10		
0101				
0110				
0111				
1000	378CF1,00	378D8D,01	378E29,10	
1001				
1010				
1011				
1100	378D18,00	378DB4,01	378E50,10	
1101				
1110	1F296F,11			
1111	1F296F,11			

4-fach satzassoziativer Cache: Trefferrate: $7/20 = 0{,}35$

Adresse	Tag	Index	Treffer
1F296FFA	1F296FF	10	–
378CF121	378CF12	00	–
1F296FFB	1F296FF	10	+
378D1831	378D183	00	–
1F296FFC	1F296FF	11	–
378D3F41	378D3F4	00	–
1F296FFD	1F296FF	11	+
378D6651	378D665	00	–
1F296FFE	1F296FF	11	+
378D8D61	378D8D6	00	–
1F296FFF	1F296FF	11	+
378DB471	378DB47	00	–
1F297000	1F29700	00	–
378DDB81	378DDB8	00	–
1F297001	1F29700	00	+
378E0291	378E029	00	–
1F297002	1F29700	00	+
378E29A1	378E29A	00	–
1F297003	1F29700	00	+
378E50B1	378E50B	00	–

Satz	Block	Tags		
00	0	378CF12	378D8D6	378E029
	1	378D183	378DB47	378E29A
	2	378D3F4	1F29700	
	3	378D665	378DDB8	378E50B
01	0			
	1			
	2			
	3			
10	0	1F296FF		
	1			
	2			
	3			
11	0	1F296FF		
	1			
	2			
	3			

Vollassoziativer Cache: Trefferrate: 7/20 = 0,35

Adresse	Tag	Treffer
1F296FFA	1F296FF,10	–
378CF121	378CF12,00	–
1F296FFB	1F296FF,10	+
378D1831	378D183,00	–
1F296FFC	1F296FF,11	–
378D3F41	378D3F4,00	–
1F296FFD	1F296FF,11	+
378D6651	378D665,00	–
1F296FFE	1F296FF,11	+
378D8D61	378D8D6,00	–
1F296FFF	1F296FF,11	+
378DB471	378DB47,00	–
1F297000	1F29700,00	–
378DDB81	378DDB8,00	–
1F297001	1F29700,00	+
378E0291	378E029,00	–
1F297002	1F29700,00	+
378E29A1	378E29A,00	–
1F297003	1F29700,00	+
378E50B1	378E50B,00	–

Block	Tags
0	1F296FF,10
1	378CF12,00
2	378D183,00
3	1F296FF,11
4	378D3F4,00
5	378D665,00
6	378D8D6,00
7	378DB47,00
8	1F29700,00
9	378DDB8,00
10	378E029,00
11	378E29A,00
12	378E50B,00
13	
14	
15	

Selbsttestaufgabe 3.4 von Seite 152

a) Geben Sie die Speicherhierarchie für heutige PCs geordnet nach absteigenden Zugriffsgeschwindigkeiten und aufsteigenden Speicherkapazitäten an.

Antwort: Register –> Primär-Cache –> Sekundär-Cache –> Hauptspeicher –> Sekundärspeicher

b) Was ist die Voraussetzung für eine effiziente Speicherhierarchie?
Antwort: Die Nutzung der zeitlichen und räumlicher Lokalität und damit ein Programmablauf, der dies zulässt.

c) Was versteht man unter Speicherverschränkung?
Antwort: Das Speichern von Speicherwörtern auf verschiedenen Speicherbänken nach einer Verschränkungsregel.

d) Welcher technologische Unterschied besteht zwischen Cache und Hauptspeicher?
Antwort: Ein Cache-Speicher ist auf dem Prozessor-Chip oder als externer SRAM-Speicher (statisches RAM) implementiert, während für den Hauptspeicher DRAM-Speicher (dynamisches RAM) verwendet werden. SRAMs werden durch Transistorschaltungen ähnlich Flipflops realisiert, sind schnell und platzaufwändig, während DRAMs mittels Speicherung der Information in kleinen Kondensatoren implementiert werden und damit im Zugriff langsamer sind, aber mehr Bits pro Chip-Fläche unterbringen können.

e) Wodurch unterscheidet sich die Rückschreibe- von der Durchschreibe-Strategie?
Antwort: Bei der Rückschreibestrategie wird ein zu speicherndes Wort zunächst nur in den Cache geschrieben und erst auf Anforderung, z.B. durch das Cache-Kohärenzverfahren oder beim Ersetzen des Cache-Blocks in den Hauptspeicher, übertragen. Beim Durchschreibeverfahren wird jedes zu speichernde Wort in den Cache und außerdem sofort in den Hauptspeicher geschrieben.

f) Welche Vor- und Nachteile hat das Durchschreib-Verfahren gegenüber dem Rückschreibverfahren?
Antwort: Durchschreib-Verfahren:
Vorteil: Es garantiert die Datenkonsistenz zwischen Cache und Arbeitsspeicher, da jeder Schreibzugriff auf den Cache gleichzeitig auch im Arbeitsspeicher ausgeführt wird.
Nachteil: Schreibzugriffe benötigen dann die langsame Zykluszeit auf den Arbeitsspeicher und belasten den Systembus.
ückschreib-Verfahren:
Vorteil: auch Schreibzugriffe können mit der schnellen Cache-Zykluszeit abgewickelt werden. Hierbei wird ein Datum erst dann in den Arbeitsspeicher zurückgeschrieben, wenn es im Cache durch ein neues verdrängt werden soll. Dadurch wird der Systembus auch entlastet.
Nachteil: Es besteht ein Problem der Datenkonsistenz zwischen Cache und Arbeitsspeicher, wenn andere Komponenten des Systems, z.B. ein DMA-Controller, auf den Arbeitspeicher zugreifen können.

g) Welche Vor- und Nachteile weisen virtuell bzw. physikalisch adressierte Cache-Speicher auf?

Antwort: Bei virtuell adressierten Cache wird die Zugriffsgeschwindigkeit durch vermeiden der Adressübersetzung erhöht, während beim physikalisch adressierten Cache vor dem Zugriff ein TLB-Zugriff nötig ist. Nachteilig ist beim virtuell adressierten Cache, die Notwendigkeit des Löschens des gesamten Cache-Inhalts vor einem Prozesswechsel sowie die Notwendigkeit einer zusätzlichen physikalischen Adresstabelle für das Schnüffeln auf dem Bus zur der Implementierung der Cache-Köhärenz (auf dem Bus gibt es natürlich nur physikalische Adressen).

h) Was bedeutet Cache-Kohärenz? Nennen Sie ein Cache-Kohärenzprotokoll?

Antwort: Eine Cache-Speicherverwaltung heißt **Cache-kohärent**, falls ein Lesezugriff immer den Wert des zeitlich letzten Schreibzugriffs auf das entsprechende Speicherwort liefert. Ein Beispiel für ein Cache-Kohärenzprotokoll ist das MESI-Protokoll.

Selbsttestaufgabe 3.5 von Seite 152

Gegeben sei ein speichergekoppeltes Multiprozessorsystem mit zwei Prozessoren, die über einen Bus mit einem globalen Speicher verbunden sind. Zur Wahrung der Cache-Kohärenz wird das MESI-Protokoll verwendet. Die Caches der beiden Prozessoren haben je eine Größe von drei Cache-Lines die jeweils genau ein Speicherwort aufnehmen können. Die Caches werden von der niedrigsten Cache-Line aufwärts gefüllt, falls noch freie Lines zur Verfügung stehen. Im Falle eines voll besetzten Caches werden sie nach der Strategie LRU (*Least Recently Used*) überschrieben. Ergänzen Sie die folgende Tabelle, wobei die Buchstaben die Zustände: M = *exclusive modified*, E = *exclusive unmodified*, S = *shared unmodified*, I = *invalid* repräsentieren. Die Zahlen hinter den Aktionen und Zuständen bezeichnen Speicheradressen.

Lösung:

Zur besseren Übersichtlichkeit sind in der folgenden Tabelle Einträge, die sich gegenüber dem vorherigen Takt nicht verändert haben, leer.

Proz.	**Aktion**	**Cache1 Line1**	**Cache1 Line2**	**Cache1 Line3**	**Cache2 Line1**	**Cache2 Line2**	**Cache2 Line3**
-	-	E/8	E/12	I/-	E/6	I/-	I/-
2	read 10					E/10	
1	write 8	M/8					
1	read 10			S/10		S/10	
2	read 8	S/8					S/8
1	write 8	M/8					I/-
1	write 8						
1	read 18		E/18				
2	write 10			I/-		M/10	
2	write 18		I/-				M/18

Selbsttestaufgabe 3.8 von Seite 177

a) Wie unterscheidet sich die North Bridge von der South Bridge in einem PC-System?

Die North Bridge stellt den Systembus dar, der den Prozessor mit Hauptspeicher, Graphikkarte (AGP-Bus) und PCI-Bus verbindet. Die South Bridge ist für die Peripheriesteuerung zuständig und ermöglich erheblich geringere Übertragungsraten.

b) Was ist das gemeinsame Ziel der neuen Prädiktoransätze?

Die destruktiven Interferenzen in neutrale zu verwandeln.

c) Welche Arten von Datenabhängigkeits-, Adress- und Wertespekulationen gibt es?

Datenabhängigkeitsspekulationen spekulieren typischerweise auf Adressunabhängigkeit bei aufeinander folgenden Speicher- und Ladebefehlen. Bei Wertespekulationen kann auf eine Ladeadresse, auf den zu ladenden Wert, auf konstante Operanden und auf konstante Wertezunahmen spekuliert werden.

d) Was unterscheidet die Cycle-by-cycle-Technik von der Block-Interleaving-Technik und beide von der simultan mehrfädigen Prozessortechnik?

Bei der Cycle-by-cycle-Interleaving-Technik wechselt der Kontext mit jedem Prozessortakt. Bei der Block-Interleaving-Technik werden die Befehle *eines* Kontrollfadens so lange wie möglich direkt aufeinander folgend ausgeführt, bis ein Ereignis eintritt, das zum Warten des Prozessors zwingt. Ein simultan mehrfädiger Prozessor (*Simultaneous Multi-Threading* – SMT) kann Befehle mehrerer Kontrollfäden in einem einzigen Taktzyklus den Ausführungseinheiten zuordnen.

e) Wie unterscheiden sich die Ziele der Superskalar-, VLIW-, EPIC-, Mehrkernprozessor- und der simultan mehrfädigen Prozessortechnik bzgl. der Programmbeschleunigung?

Die Superskalar-, VLIW- und EPIC-Technik optimieren die Ausführungsgeschwindigkeit eines einzelnen Kontrollfadens. Die Mehrkernprozessor- und die simultan mehrfädigen Prozessortechnik optimieren den Durchsatz einer mehrfädigen Last – also eines Programmixes.

Kapitel 4

Multiprozessorsysteme

Kapitelinhalt

Zusammenfassung

Dieses Kapitel behandelt die heute als Server und als Supercomputer eingesetzten Multiprozessorsysteme. Durch das Aufkommen kommerziell verfügbarer Mehrkernprozessoren werden aber zunehmend auch Desktop-Systeme eine ähnliche Form annehmen. Die ersten beiden Abschnitte stellen Grundlagen für die speicher- und nachrichtengekoppelten Multiprozessormodelle vor, die in den Abschnitten 3 und 4 behandelt werden. Im ersten Abschnitt werden quantitative Maßzahlen für Parallelrechner eingeführt. Abschnitt 2 klassifiziert die für Multiprozessoren wichtigen Verbindungsstrukturen.

Lernziele

Die Lernziele dieses Kapitels sind:

- Kenntnis von Modellen und Maßzahlen von Multiprozessorsystemen,
- Verständnis der Verbindungsstrukturen von Multiprozessoren,
- Kenntnis von speichergekoppelten und nachrichtengekoppelten Multiprozessorsystemen,
- Kenntnis von Konsistenzmodellen für speichergekoppelte Multiprozessorsysteme.

4.1 Quantitative Maßzahlen für parallele Systeme

Leistungsangaben über ein Rechensystem müssen mit Skepsis gesehen werden, wenn nicht bekannt ist, unter welchen Bedingungen und mit welcher spezifischen Last diese Leistung erreicht worden ist. Dies gilt besonders für Multiprozessorsysteme, deren Leistung eigentlich nicht durch eine einzige Angabe über „Anzahl der ausgeführten Operationen pro Zeiteinheit“ erfasst werden kann, denn in der Realität treten oft technische und funktionale Besonderheiten auf, sodass die Leistung gerade dieser Rechensysteme vielfältigen Einflüssen ausgesetzt ist. Daher sollte jede Leistungsangabe mit zusätzlichen Informationen über die Bedingungen versehen sein, unter denen diese Leistung erreicht wurde, wobei möglichst alle Einflussfaktoren erfasst werden sollten. Dazu können z.B. Art und Zahl der beteiligten Hardware-Komponenten, deren Auslastung, Verbindungsstruktur, Aufteilung von Daten auf Speichermodule, Zugriffe von Prozessoren auf diese Daten, Operationsmodus des Betriebssystems, Art des Anwenderprogramms, Granularität der Parallelverarbeitung usw. gehören.

Benchmark

Ein Programmm, während dessen Ausführung die Leistung eines Rechners gemessen wird, nennt man *Benchmark*. Die gemeinsame Nutzung von Benchmarks durch viele Hersteller schafft Vergleichbarkeit. Die Verfügbarkeit von Benchmarks für verschiedene Anwendungszwecke erlaubt es einem Käufer, ein geeignetes Rechnersystem für den von ihm intendierten Einsatz auszuwählen. Man unterscheidet *synthetische* Benchmarks, die in kleinen Codestücken die Charakteristiken verschiedener Anwendungen nachbilden, sowie Benchmarks, die aus Teilen oder ganzen Anwendungsprogrammen bestehen. Oft bilden Benchmarks so genannte *Suiten* mit Anwendungen aus verschiedenen Bereichen. Zum Test von Einzelrechnern spielt die SPEC-Suite (*Standard Performance Evaluation Corporation*) (s. `http://www.spec.org`) eine wichtige Rolle, die Integer– und Gleitkomma-Rechenleistungen bewertet. Daneben gibt es auch Benchmarks für Graphik-Adapter und Festplatten. Ein wichtiges Benchmark für Datenbank-Rechner stellt TPC-C des *Transaction Processing Performance Council* (s. `http://www.tpc.org`) dar, das Maßzahlen in Transaktionen pro Sekunde misst. Für Parallelrechner stellt im Bereich des wissenschaftlichen Rechnens das Benchmark *Linpack* (s. `http://www.top500.org/about/linpack`) einen Quasi-Standard dar, auf dem die Liste der 500 schnellsten Rechner der Welt basiert, die halbjährlich aktualisiert wird (s. `http://www.top500.org`). Daneben gibt es aber auch parallele Benchmark-Suiten mit Anwendungen aus verschiedenen Bereichen, wie zum Beispiel die Splash-Benchmarks (s. `http://www-flash.stanford.edu/apps/SPLASH/`).

Neben einer Angabe über „Anzahl der ausgeführten Operationen pro Zeiteinheit“, meist in MIPS (*Millions of Instructions per Second*) oder MFLOPS (*Millions of Floating-point Operations per Second*) spezifiziert, können auch andere Eigenschaften des Multiprozessorsystems wie z.B. Fehlertoleranz, Verfügbarkeit, Zuverlässigkeit und Kompatibilität für den Anwender eine wichtige Rolle spielen und in einer Wechselwirkung mit Leistungsangaben stehen. Die im Folgenden vorgestellten Ansätze sind zwar weit verbreitet, sie berücksichtigen jedoch immer nur einen geringen Teil aller möglichen Einflüsse und Wechsel-

wirkungen. Sie können jedoch als Leistungseinschätzung beim Entwurf eines parallelen Programms dienen.

Übertragungszeit = Startzeit + Transferzeit

Aus Hardware-Sicht stellen die Zugriffszeit auf den entfernten Speicher bzw. die Übertragungszeit einer Nachricht wichtige Leistungsparameter dar. Die **Übertragungszeit einer Nachricht** T_{msg} ist die Zeit, die für das Verschicken einer Nachricht einer bestimmten Länge zwischen zwei Prozessen (oder Prozessoren) benötigt wird. Sie setzt sich zusammen aus der **Startzeit** t_s (*Message Start-up Time*), die benötigt wird, um die Kommunikation in Software und Hardware zu initiieren, und der **Transferzeit** t_w pro übertragenem Datenwort, die von der physikalischen Bandbreite des Kommunikationsmediums abhängt.

Die Übertragungszeit einer Nachricht mit L Datenwörtern beträgt:

$$T_{msg} = t_s + t_w \cdot L$$

Voraussetzung ist dabei, dass das Verbindungsnetz konfliktfrei ist. Ansonsten kann die Übertragungszeit nicht fest berechnet werden. Abbildung 4.1 stellt die Zusammenhänge graphisch dar.

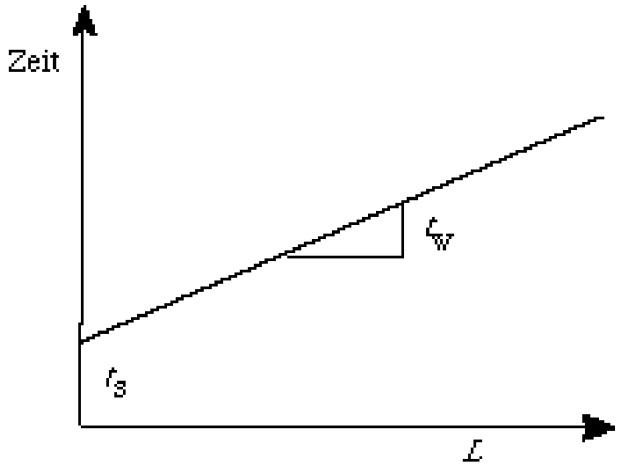


Abbildung 4.1: Übertragungszeit einer Nachricht

Ausführungszeit eines parallelen Programms

Die (Gesamt-)**Ausführungszeit** T (*Execution Time*) **eines parallelen Programms** ist die Zeit zwischen dem Starten der Programmausführung auf einem der Prozessoren und dem Zeitpunkt, an dem der letzte Prozessor die Arbeit an dem Programm beendet hat. Während der Programmausführung sind alle Prozessoren in einem der drei Zustände „rechnend“, „kommunizierend“ oder „untätig“. Dabei geht man zur Vereinfachung von einem dezidiert zugeordneten Parallelrechner ohne *Multiprogramming*-Betrieb und Prozesswechsel aus. Dementsprechend wird definiert:

- die **Berechnungszeit** T_{comp} (*Computation Time*) eines Algorithmus als die Zeit, die für Rechenoperationen verwendet wird,
- die **Kommunikationszeit** T_{comm} (*Communication Time*) eines Algorithmus als die Zeit, die für Sende- und Empfangsoperationen verwendet wird, und

- die **Untätigkeitszeit** T_{idle} (*Idle Time*) als die Zeit, die mit Warten (auf zu empfangende Nachrichten oder auf das Versenden) verbraucht wird.

Es gilt: $T = T_{comp} + T_{comm} + T_{idle}$.

Die Leistungsangaben zu Multiprozessorsystemen werden mit Leistungsangaben zu Einprozessorsystemen in Beziehung gesetzt. Notwendig für den Vergleich ist dann mindestens ein Programm, das auf den beiden zu vergleichenden Systemen ablaufen kann.

Maßzahlen für den Vergleich Multi– mit Einprozessorsystem

Die wichtigsten Maßzahlen, die für einen Vergleich eines Multiprozessorsystems mit einem Einprozessorsystem verwendet werden, sind die Beschleunigung und die Effizienz. Zu deren Definition werden einige Hilfsgrößen benötigt, die sich auf *ein* Programm beziehen.

$P(1)$**:** Anzahl der auszuführenden (Einheits-)Operationen des Programms auf einem Einprozessorsystem.

$P(n)$**:** Anzahl der auszuführenden (Einheits-)Operationen des Programms auf einem Multiprozessorsystem mit n Prozessoren.

$T(1)$**:** Anzahl der Taktzyklen auf einem Einprozessorsystem.

$T(n)$**:** Anzahl der Taktzyklen auf einem Multiprozessorsystem mit n Prozessoren.

Die Schritte (oder Takte) lassen sich für einen konkreten Rechner auf Zeiteinheiten wie Sekunden etc. umrechnen.

Es wird von folgenden vereinfachenden Voraussetzungen ausgegangen:

$T(1) = P(1)$, da in einem Einprozessorsystem jede (Einheits-)Operation in genau einem Schritt ausgeführt werden kann.

$T(n) \leq P(n)$**,** da in einem Multiprozessorsystem mit n Prozessoren ($n \geq 2$) in einem Schritt mehr als eine (Einheits-)Operation ausgeführt werden kann.

Die **Beschleunigung** (Leistungssteigerung, *Speed-up*) $S(n)$ ist definiert als

$$S(n) = \frac{T(1)}{T(n)}$$

und gibt die Verbesserung in der Verarbeitungsgeschwindigkeit an, wobei sich der Wert natürlich nur auf das jeweils bearbeitete Programm bezieht bzw. als Mittelwert für eine Reihe von Programmen angesehen werden kann.

Üblicherweise gilt

$$1 \leq S(n) \leq n$$

Danach arbeitet also im schlechtesten Fall das Multiprozessorsystem gerade so schnell wie das Einprozessorsystem und im besten Fall ist die Leistungssteigerung linear zur Anzahl der eingesetzten Prozessoren[1]. Diese Ungleichung kann als Grundlage für alle weiteren Überlegungen angesehen werden.

Effizienz

[1] Die folgenden Möglichkeiten können in seltenen Fällen ebenfalls auftreten:
$S(n) < 1$, weil es zu einem Leistungsabfall auf einem Multiprozessorsystem kommen kann.
$S(n) > n$, weil aufgrund so genannter „synergetischer Effekte“ (siehe später) eine überproportionale Leistungssteigerung entstehen kann.

Die **Effizienz** $E(n)$ (*Efficiency*) ist definiert als

$$E(n) = \frac{S(n)}{n}$$

und gibt die *relative* Verbesserung in der Verarbeitungsgeschwindigkeit an, da die Leistungssteigerung mit der Anzahl der Prozessoren n normiert wird. Unter Voraussetzung der obigen Ungleichung gilt:

$$\frac{1}{n} \leq E(n) \leq 1$$

absolute vs. relative Beschleunigung und Effizienz

Man kann die Begriffe Beschleunigung und Effizienz „algorithmenunabhängig“ oder „algorithmenabhängig“ definieren. Im ersten Fall setzt man den besten bekannten sequentiellen Algorithmus für das Einprozessorsystem in Beziehung zum vergleichenden parallelen Algorithmus für das Multiprozessorsystem. Man erhält dann die so genannte **absolute Beschleunigung** bzw. die **absolute Effizienz**. Im zweiten Fall benutzt man den parallelen Algorithmus so, als sei er sequentiell, und misst dessen Laufzeit auf einem Einprozessorsystem. Man spricht dann von **relativer Beschleunigung** und **relativer Effizienz**. Da bei einem Einprozessorsystem alle Daten im (gleichen) Hauptspeicher liegen, muss die Kommunikation softwaremäßig nachgebildet werden. Dieser eigentlich unnötige Mehraufwand erhöht die resultierende Beschleunigung. In diesem Fall kommt der für die Parallelisierung erforderliche Zusatzaufwand an Kommunikation und Synchronisation „ungerechterweise“ auch für den sequentiellen Algorithmus zum Tragen.

Skalierbarkeit eines Parallelrechners

Von **Skalierbarkeit** eines Parallelrechners spricht man, wenn das Hinzufügen von weiteren Verarbeitungselementen zu einer kürzeren Gesamtausführungszeit führt, ohne dass das Programm geändert werden muss. Insbesondere meint man damit eine lineare Steigerung der Beschleunigung mit einer Effizienz nahe bei Eins.

Wichtig für die Skalierbarkeit ist eine angemessene Problemgröße. Bei fester Problemgröße und steigender Prozessorzahl wird ab einer bestimmten Prozessorzahl eine Sättigung eintreten. Die Skalierbarkeit ist in jedem Fall beschränkt.

gleichzeitige Skalierung der Problemgröße

Skaliert man mit der Anzahl der Prozessoren auch die Problemgröße (*Scaled Problem Analysis*), so tritt dieser Effekt bei gut skalierenden Hardware- oder Software-Systemen nicht auf. Die Belastung eines Systems muss der jeweiligen Leistungsfähigkeit entsprechen. Die folgende Analogie demonstriert diese Zusammenhänge: Ein Gärtner pflanzt einen Baum in einer Stunde. Dann ist nicht zu erwarten, dass 60 Gärtner *einen* Baum in einer Minute pflanzen. Auch hier ist klar, dass sich eine sinnvolle Leistungssteigerung erst beim Pflanzen von tausend oder zehntausend Bäumen einstellen kann.

Mehraufwand

Der **Mehraufwand** für die Parallelisierung $R(n)$ ist definiert als

$$R(n) = \frac{P(n)}{P(1)}$$

und beschreibt den bei einem Multiprozessorsystem erforderlichen Mehraufwand für die Organisation, Synchronisation und Kommunikation der Prozessoren. Es gilt:

$$R(n) \geq 1$$

Das bedeutet, dass die Anzahl der auszuführenden Operationen eines parallelen Programms größer als diejenige des vergleichbaren sequentiellen Programms ist.

Der **Parallelindex** $I(n)$ (*Parallel Index*) ist definiert als Parallelindex

$$I\left(n\right) = \frac{P\left(n\right)}{T\left(n\right)}$$

und gibt den mittleren Grad an Parallelität bzw. die Anzahl der parallelen Operationen pro *Taktzyklus* an. Alternativ dazu kann man den Parallelindex auch als Anzahl der parallelen Operationen pro *Zeiteinheit* definieren. In diesem Fall teilt man durch die Zeit, die das parallele Programm benötigt. Wenn T_C die Taktperiodendauer bezeichnet, so ergibt sich dieser Parallelindex I^* zu:

$$I^*\left(n\right) = \frac{P\left(n\right)}{T\left(n\right) \cdot T_C}$$

Man beachte, dass diese Maßzahl die Einheit s^{-1} hat. Da die Taktperiodendauer stark von der verwendeten Technologie abhängig ist, wird I^* nicht zu Vergleichen auf Mikroarchitekturebene benutzt. Wir werden uns daher im Folgenden auf die erstgenannte Definition beschränken, die eine dimensionslose Zahl liefert.

Die **Auslastung** $U(n)$ (*Utilization*) ist definiert als Auslastung

$$U\left(n\right) = \frac{I\left(n\right)}{n}$$

und entspricht dem normierten Parallelindex. Diese Maßzahl gibt an, wieviele Operationen *pro Prozessor* im Durchschnitt pro Taktzyklus ausgeführt werden.

Bei der nachfolgenden Herleitung wird gezeigt, dass $U(n) = R(n) \cdot E(n)$ gilt. Dabei wird vereinfachend angenommen (s.o.), dass $P(1) = T(1)$ gilt, d.h. dass pro Operation genau ein Taktzyklus benötigt wird.

$$\begin{aligned} U\left(n\right) &= \frac{I\left(n\right)}{n} \\ &= \frac{P(n)}{T(n)} \cdot \frac{1}{n} \cdot \frac{P(1)}{P(1)} \\ &= \frac{P(n)}{P(1)} \cdot \frac{P(1)}{T(n)} \cdot \frac{1}{n} \\ &= R\left(n\right) \cdot E\left(n\right) \end{aligned}$$

Aus diesen Definitionen lassen sich einige **Folgerungen** ableiten:

- Alle definierten Ausdrücke haben für $n = 1$ den Wert 1.
- Der Parallelindex gibt eine obere Schranke für die Leistungssteigerung:

$$1 \leq S\left(n\right) \leq I\left(n\right) \leq n$$

- Die Auslastung ist eine obere Schranke für die Effizienz:

$$\frac{1}{n} \leq E(n) \leq U(n) \leq 1$$

Die Bedeutung dieser Definitionen soll mit einem **Zahlenbeispiel** verdeutlicht werden. Ein Einprozessorsystem benötige für die Ausführung von 1000 Operationen 1000 Schritte. Ein Multiprozessorsystem mit 4 Prozessoren benötige dafür 1200 Operationen, die aber in 400 Schritten ausgeführt werden können. Damit gilt also

$$P(1) = T(1) = 1000, \quad P(4) = 1200 \text{ und } T(4) = 400$$

Daraus ergibt sich:

$$S(4) = 2,5 \text{ und } E(4) = 0,625$$

Die Leistungssteigerung verteilt sich also zu 62,5% auf jeden Prozessor.

$$I(4) = 3 \text{ und } U(4) = 0,75$$

Es sind im Mittel also drei Prozessoren gleichzeitig tätig, d.h., jeder Prozessor ist nur zu 75% der Zeit aktiv.

$$R(4) = 1,2$$

Bei der Ausführung auf einem Multiprozessorsystem sind 20% mehr Operationen als bei der Ausführung auf einem Einprozessorsystem notwendig.

Bei diesen Berechnungen muss man sich darüber im klaren sein, dass die Ergebnisse lediglich Mittelwerte für die Programme darstellen, für die Messungen durchgeführt werden. Um fundierte Aussagen für ein bestimmtes System zu erhalten, ist darauf zu achten, dass das ganze Anwendungsgebiet abgedeckt wird und entsprechend viele Algorithmen ausgewertet werden.

Abschätzung für die Leistungssteigerung

Die wichtigste Abschätzung für die Leistungssteigerung $S(n)$ durch Nutzung eines Multiprozessorsystems stammt von Amdahl. Dieser betrachtet einen weiteren Parameter a $(0 \leq a \leq 1)$, der den Bruchteil eines Programms darstellt und der nur sequentiell bearbeitet werden kann. Die entstehende Formel ist als **Amdahls Gesetz** bekannt geworden.

Amdahls Gesetz

Die Gesamtausführungszeit errechnet sich aus der Summe der Ausführungszeit des nur sequentiell ausführbaren Programmteils a und der Ausführungszeit des parallel ausführbaren Programmteils $1 - a$, die letztere dividiert durch die Anzahl n der parallel arbeitenden Prozessoren. Es gilt somit unter der Vernachlässigung von Synchronisations– und Kommunikationszeiten:

$$T(n) = T(1) \cdot \frac{1-a}{n} + T(1) \cdot a$$

Aus der Formel folgt:

$$S(n) = \frac{T(1)}{T(n)} = \frac{T(1)}{T(1) \cdot \frac{1-a}{n} + T(1) \cdot a} = \frac{1}{\frac{1-a}{n} + a}$$

Bildet man den Grenzwert für $n \to \infty$, so erhält man:

$$\lim_{n\to\infty} S(n) = \frac{1}{a} .$$

Somit besagt Amdahls Gesetz: $S(n) \leq 1/a$.

Amdahls Gesetz zufolge kann eine kleine Anzahl von sequentiellen Operationen die mit einem Parallelrechner erreichbare Beschleunigung signifikant begrenzen. Falls z.B. $a = 1/10$ des parallelen Programms nur sequentiell ausgeführt werden kann, so kann das gesamte Programm maximal zehnmal schneller als ein vergleichbares, rein sequenzielles Programm verarbeitet werden. Amdahls Gesetz wurde auch als eines der stärksten Argumente gegen die weitere Entwicklung der Multiprozessorsysteme bezeichnet. Wenn nämlich Multiprozessorsysteme niemals mehr als vielleicht zehn- bis zwanzigmal schneller als Einprozessorsysteme sein könnten, so würden sich die Entwicklungskosten für Multiprozessorsysteme kaum lohnen.

Es hat sich jedoch gezeigt, dass es viele parallele Programme gibt, die einen sehr geringen sequentiellen Anteil ($a << 1$) besitzen. Viele Anwendungen verlangen nach Rechnern, die um mehrere Größenordnungen schneller als die existierenden Einprozessorrechner sind. Als Erfahrungstatsache gilt, dass ein Problem für Multiprozessoren dann interessant ist, wenn der parallel bearbeitbare Anteil die Anzahl der zur Verfügung stehenden Prozessoren weit übersteigt. Heutige Erfahrungen zeigen, dass bei derartigen Problemen typischerweise eine fast lineare Beschleunigung erreicht werden kann.

Ähnlich verhält sich der Speedup, wenn die Problemgröße proportional zur verfügbaren Rechnerleistung wächst. Während Amdahl von einer konstanten Problemgröße ausgeht, stellte Gustafson die Frage, wie sich der Speedup bei konstanter Laufzeit aber mitwachsender Problemgröße berechnet. Für die sequentiellen Anteile von null bzw. hundert Prozent liefert Gustafsons Gesetz für den *skalierten* Speedup zwar die gleichen Resultate wie Amdahls Gesetz (n bzw. 1). Während nach Amdahl der Speedup zwischen diesen Extremwerten jedoch hyperbelförmig abfällt, sinkt der skalierte Speedup nach Gustafsons Gesetz nur linear.

Der synergetische Effekt oder superlineare Speed-up

Bei einigen sehr speziellen Problemen kann sogar ein so genannter **synergetischer Effekt**[2] oder **superlinearer Speed-up** ($S > n$, $E > 1$) beobachtet werden. Von der Algorithmentheorie her kann es jedoch einen superlinearen Speed-up eigentlich nicht geben: Jeder parallele Algorithmus lässt sich auf einem Einprozessorsystem simulieren, indem in einer Schleife jeweils der nächste Schritt jedes Prozessors der parallelen Maschine emuliert wird.

Ein superlinearer Speed-up kann jedoch bei Problemen beobachtet werden, die auf Einprozessormaschinen mittels Backtracking (*depth-first search*) gelöst werden. Bei der Übertragung auf Multiprozessorsysteme kann jeder Prozessor auf einen Zweig des Problemlösungsbaumes angesetzt werden und nun bei Erreichen einer Lösung den anderen Prozessoren diese signalisieren. Wo

[2] Ursprünglich stammt das Wort „Synergie“ aus dem Griechischen. Es bedeutet „Zusammenwirken“ und wird hauptsächlich in der Medizin verwendet. Hier hat man beobachtet, dass mehrere für sich schwach wirkende Medikamente eine starke Reaktion hervorrufen können, wenn sie kombiniert werden.

eine Einprozessormaschine eventuell tiefe, nutzlose Zweige untersuchen muss, kann auf einem Multiprozessorsystem die Bearbeitung unter Umständen zu einem frühen Zeitpunkt abgebrochen werden. Gegenargument: Der sequentielle Backtracking-Algorithmus und der parallele Algorithmus sind nicht miteinander vergleichbar.

Von einem superlinearen Speed-up wird auch bei der Lösung einer dreidimensionalen partiellen Differentialgleichung auf einem Rechnernetz versus einem einzelnen Rechner berichtet: Beim Programmlauf auf einem Rechner passten die Matrizen nicht in den Hauptspeicher des Rechners, sodass durch die virtuelle Speicherverwaltung ein häufiger Seitenwechsel ausgelöst wurde. Nach der Verteilung auf das Rechnernetz konnten die parallelen Programme vollständig in den Cache- und Hauptspeichern der einzelnen Rechner ablaufen.

Natürlich sind in einem solchen Fall Einprozessor- und Mehrprozessorrechner nicht direkt vergleichbar, da bei n Prozessoren die n-fache Speichergröße vorhanden ist. Da dies in der Praxis bei Multiprozessoren und bei Rechnernetzen häufig der Fall ist, sollte man den Fall so interpretieren, dass durch eine größere Anzahl von Prozessoren ein Problem *schneller* ausgeführt werden kann, und dass durch eine Vergrößerung der Speicherkapazität *größere* Probleme gerechnet werden können.

In der Theorie wird von einem beliebig großen Speicher ausgegangen. In der Praxis ist dies jedoch nicht der Fall. In der Praxis kann deshalb ein superlinearer Speed-up durchaus auftreten, obwohl er von der Theorie her nicht möglich ist.

In Multiprozessorsystemen treten auch noch andere Effekte auf, die ähnlich wie die synergetischen Effekte schwer abzuschätzen sind, im Gegensatz zu diesen jedoch zu einer Verschlechterung des Wirkungsgrades führen. Dazu gehören z.B.

- der Verwaltungsaufwand (*Overhead*), der mit der Zahl der zu verwaltenden Prozessoren ansteigt,
- die Möglichkeit von Systemverklemmungen (*Deadlocks*) und
- die Möglichkeit von Sättigungserscheinungen, die durch Systemengpässe (*Bottlenecks*) verursacht werden.

Zusammenfassend lässt sich sagen, dass durchaus für ein vorgegebenes Multiprozessorsystem in einer bestimmten Konfiguration sowie für eine vorgegebene Menge von Programmen mit einer vorgegebenen Menge von Eingabedaten die oben angegebenen Maßzahlen wie Leistungssteigerung, Effizienz, Mehraufwand, Parallelindex und Auslastung gemessen werden können. Detaillierte, allgemeingültige Aussagen sind jedoch wegen der vielen nichtdeterministischen Effekte, die in der Praxis auftreten, schwer zu treffen.

Selbsttestaufgabe 4.1 *Gegeben sei ein Multiprozessorsystem mit 16 Prozessoren. Die Leistungssteigerung bezüglich eines Einprozessorsystems sei S(16) = 4. Die Ausführungszeit auf dem Einprozessorsystem sei T(1) = 32 und die Anzahl der auszuführenden (Einheits-) Operationen auf dem Multiprozessorsystem sei P(16) = 40.*

1. *Geben Sie die Effizienz E(16) an.*

2. *Berechnen Sie die Ausführungszeit des Multiprozessorsystems T(16).*

3. *Geben Sie den Parallelindex I(16) an.*

4. *Welcher Bruchteil der Programme ist nach Amdahls Gesetz nur sequentiell ausführbar?*

5. *Durch welchen Wert wird die Leistungssteigerung begrenzt, wenn man die Anzahl der Prozessoren beliebig erhöht?*

4.2 Verbindungsstrukturen

4.2.1 Beurteilungskriterien und Klassifizierung

Verbindungsnetz

Das **Verbindungsnetz** ist das Medium, über das die Kommunikation der Prozessoren untereinander und der Zugriff auf gemeinsame Ressourcen abgewickelt werden. Bei der Untersuchung von Verbindungsnetzen werden im Folgenden Beurteilungskriterien zur Bewertung, Unterscheidungsmerkmale und Klassifikationen aufgrund dieser Unterscheidungsmerkmale vorgestellt. Verbindungsstrukturen lassen sich gemäß folgender Beurteilungskriterien bewerten:

- **Komplexität** oder **Kosten** bedeutet den Hardware-Aufwand für das Verbindungsnetz gemessen in der Anzahl und der Art der Schaltelemente und Verbindungsleitungen. Wenn Knoten direkt (und nicht über Schaltelemente) miteinander verbunden sind, so drücken sich niedrige Kosten für das Verbindungsnetz in einem beschränkten Verbindungsgrad jedes Knotens aus. Der **Verbindungsgrad** eines Knotens ist definiert als die Anzahl der Verbindungen, die von dem Knoten zu anderen Knoten bestehen.

- **Diameter** oder **Durchmesser**: Darunter versteht man die maximale Distanz für die Kommunikation zweier Prozessoren, also die Anzahl der Verbindungen, die durchlaufen werden müssen. Man spricht auch von der **maximalen Pfadlänge** zwischen zwei Knoten.

- **Regelmäßigkeit** des Verbindungsmusters: Ein regelmäßiges Verbindungsmuster lässt sich meist besser implementieren.

- Notwendige **Leitungslängen:** Kurze Leitungslängen für alle Verbindungen eines Verbindungsnetzes sind vorteilhaft für die Implementierung eines Netzes.

- **Blockierung:** Ein Verbindungsnetz heißt **blockierungsfrei**, falls jede gewünschte Verbindung zwischen den Prozessoren oder zwischen den Prozessoren und Speichern unabhängig von schon bestehenden Verbindungen hergestellt werden kann.

- **Erweiterbarkeit:** Multiprozessoren können begrenzt, stufenlos oder – wie beispielsweise bei Hyperkubussystemen – nur durch Verdopplung der Anzahl der Prozessoren erweiterbar sein.

- **Skalierbarkeit:** Darunter wird die Fähigkeit verstanden, die wesentlichen Eigenschaften des Verbindungsnetzes auch bei beliebiger Erhöhung der Knotenzahl beizubehalten. Insbesondere ist zu beachten, dass gewisse Verbindungsstrukturen nicht unbegrenzt skalierbar sind, sondern nur eine maximalen Anzahl von Knoten verbinden können.

- **Ausfallstoleranz** oder **Redundanz** ist die Möglichkeit, Verbindungen zwischen Knoten selbst dann noch zu schalten, wenn einzelne Elemente des Netzes (Schaltelemente, Leitungen) ausfallen. Ein fehlertolerantes Netz muss also zwischen jedem Paar von Knoten mindestens einen zweiten, redundanten Weg bereitstellen. Die Eigenschaft eines Systems, bei Ausfall einzelner Komponenten unter deren Umgehung funktionstüchtig zu bleiben, wenn auch mit verminderter Leistung, wird als *Graceful degradation* bezeichnet.

- **Durchsatz** oder **Übertragungsbandbreite:** Die maximale, meist theoretisch errechnete Übertragungsleistung des Verbindungsnetzes oder einzelner Verbindungen wird meist in Megabits pro Sekunde (MBit/s) gemessen.

- **Komplexität der Pfadberechnung** oder **Wegefindung:** Darunter versteht man die Art, wie der Weg einer Nachricht vom Sender- zum Zielknoten berechnet wird. Die Wegefindung sollte einfach sein, um mittels eines schnellen Hardware-Algorithmus in jedem Verbindungselement implementierbar zu sein. Zu einer Verbindungsstruktur kann es mehrere unterschiedliche Wegefindungsalgorithmen geben, welche auch die Eigenschaften der Blockierungsfreiheit und der Ausfalltoleranz des Verbindungsnetzes beeinflussen können.

Unterscheidungsmerkmal für Verbindungsnetze

Wesentliches **Unterscheidungsmerkmal** für Verbindungsnetze ist zunächst die Art der Verbindung zwischen den Knoten: Hier wird zwischen statischen und dynamischen Netzen unterschieden. Bei **statischen Netzen** existieren fest installierte Verbindungen zwischen Paaren von Netzknoten. **Dynamische Netze** dagegen enthalten eine Komponente „Schaltnetz“, an die alle Knoten über Ein- und Ausgänge angeschlossen sind. Direkte, fest installierte Verbindungen zwischen den Knoten existieren in diesem Fall nicht.

Der grundsätzliche Unterschied zwischen statischen und dynamischen Netzen besteht in der Art, wie Verbindungen zwischen Knoten hergestellt werden. In dynamischen Netzen sind alle notwendigen Steuerungsfunktionen im Schaltnetz konzentriert. Bei statischen Netzen ist die Steuerung des Verbindungsaufbaus Teil der Knoten selbst. Dort werden sie von den Prozessoren selbst oder von spezieller Vermittlungs-Hardware wahrgenommen.

Für den Datentransfer von einem Prozessor zu einem anderen oder zu einem Speichermodul gibt es – ähnlich wie bei Rechnernetzen – zwei unterschiedliche

Prinzipien: **Paketvermittlung** (*Packet Switching*) und **Durchschalte**– oder **Leitungsvermittlung** (*Circuit Switching*).

Paketvermittlung heißt, dass entweder Datenpakete fester Länge oder Nachrichten variabler Länge entsprechend einem Wegefindungsalgorithmus (*Routing*) vom Absender zum Empfänger geschickt werden. Bei Datenpaketen fester Länge müssen die zu übertragenden Daten vor dem Absenden in Datenpakete zerlegt und vom Empfänger nach dem Eintreffen wieder zusammengesetzt werden.

Mit **Durchschaltevermittlung** wird die Eigenschaft eines Netzes bezeichnet, eine direkte Verbindung zwischen zwei oder mehreren Knoten eines Netzes zu schalten. Die physikalische Verbindung bleibt für die gesamte Dauer der Informationsübertragung bestehen. Durchschaltenetze erreichen im allgemeinen wesentlich höhere Übertragungsraten als Paketvermittlungsnetze, bei denen die Übertragungsrate oft von der Gesamtbelastung des Netzes abhängt. In Durchschaltenetzen beeinflusst die Netzbelastung jedoch in der Regel die Möglichkeiten, überhaupt Verbindungen zwischen Knoten herzustellen.

Bei der Paketvermittlung erfordert die Mehrfachnutzung der Kanäle einen erhöhten Verwaltungsaufwand, denn jedes Paket (und jede Nachricht) muss mit einer Zieladresse versehen werden, die in den Zwischenknoten oder Schaltelementen interpretiert werden muss. Die Paketvermittlung ist die für Multiprozessoren bevorzugte Verbindungsart. Die im folgenden eingeführten Begriffe (siehe Abbildung 4.2) beziehen sich deshalb weitgehend auf die Paketvermittlung.

Begriffe der Wegefindung

Bei der Art der Adressierung kann man zwischen **zielbasiert** (*Destination-based Routing*) und **quellenbasiert** (*Source-based Routing*) unterscheiden. Im Fall der zielbasierten Adressspezifikation wird der Kopfteil eines Pakets (oder einer Nachricht) mit einer systemweit eindeutigen Empfängeradresse versehen, die bei der Wegefindung von jedem Zielknoten zur Auswahl eines Übertragungskanals genutzt wird. Bei der quellenbasierten Adressspezifikation wird das Paket mit allen Informationen versehen, die es benötigt, um über die Zwischenknoten zum Empfänger zu gelangen. Für jeden Zwischenknoten wird im voraus die Abzweigung bestimmt, die das Paket nehmen muss. Die zielbasierte Adressspezifikation wird überwiegend bei statischen Verbindungsnetzen und die quellenbasierte Adressspezifikation bei dynamischen Verbindungsnetzen angewandt.

Eine Nachricht selbst wird in eine Anzahl von **Übertragungseinheiten** (*Phits – Physical Transfer Units –* oder auch *Flits – Flow Control Digits –* genannt) zerlegt, die wiederum in geordneter Reihenfolge immer über dieselbe Verbindung übertragen werden. Ein *Phit* ist dabei die Datenportion, die zu einem Zeitpunkt zwischen zwei Knoten übertragen werden kann. Ein Phit entspricht typischerweise der Breite der physikalischen Verbindung zwischen den Knoten.

Wegewahl

Falls Nachrichten zwischen zwei Knoten immer den gleichen Weg gehen müssen (also immer nur eine Verbindung geschaltet werden kann), spricht man von **deterministischer Wegewahl**. Es ergibt sich der Vorteil einer einfachen Pfadberechnung, aber die Nachteile der erhöhten Möglichkeit zu Blockierungen und des Mangels an Fehlertoleranz, da eine eventuell vorhandene Redun-

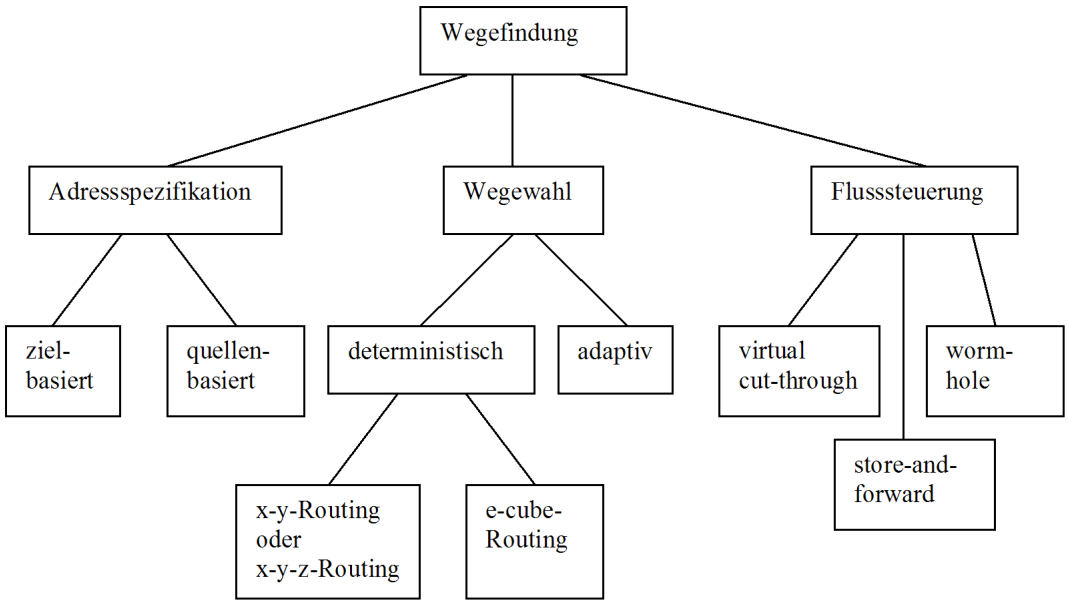


Abbildung 4.2: Gliederung des Begriffs „Wegefindung“.

danz im Netz nicht ausgenutzt wird. Durch die Möglichkeit der so genannten **adaptiven Wegewahl**, auch andere Wege zu nehmen, werden die Nachteile der deterministischen Wegewahl durch einen etwas höheren Hardware-Aufwand vermieden. Blockierte Strecken und ausgefallene Knoten oder Schaltelemente können umgangen werden.

Wichtige Definitionen der Wegewahlmodi!

Bei der Nachrichtenübertragung zwischen nicht benachbarten Sender- und Empfängerknoten müssen in den Zwischenknoten Puffer eingerichtet sein, um Nachrichten, Pakete oder zumindest einzelne Übertragungseinheiten von Paketen zwischenzuspeichern. Bei der Zwischenspeicherung muss der Knoten dafür sorgen, dass sein Puffer nicht überläuft. Dies geschieht durch eine zusätzliche **Flusssteuerung** (*Flow Control*), die entweder über spezielle Leitungen (z.B. *Ack*- oder *Strobe*-Leitung) vom empfangenden (Zwischen-)knoten zum sendenden (Zwischen-)knoten oder über ein Flusssteuerungspaket vorgenommen wird. Je nach Organisation der Flusssteuerung kann man folgende Übertragungsmodi unterscheiden:

- ***Store-and-forward*-Modus:** Dabei wird die Nachricht von jedem Zwischenknoten in Empfang genommen, vollständig zwischengespeichert und dann auf dem Pfad, auf dem die Nachricht weiter in Richtung Empfänger transportiert werden kann, zum nächsten benachbarten Zwischen- oder dem Empfängerknoten übertragen. In diesem Fall existiert zu keinem Zeitpunkt eine durchgehende Verbindung vom Absender zum Empfänger. Der Store-and-forward-Modus wurde bei der ersten Generation von Multiprozessoren angewandt und dort in Software implementiert. Er ist bei Weitverkehrsnetzen auch heute noch der bevorzugte Übertragungsmodus.

- ***Virtual-Cut-through*-Modus:** Bei diesem Modus wird eine Nachricht als Kette von Übertragungseinheiten transportiert. Der Kopfteil der Nachricht enthält die Empfängeradresse und bestimmt damit den einzuschlagenden Weg. Bei Ankunft der ersten Übertragungseinheit an einem Zwischenknoten wird diese sofort an den nächsten Knoten weitergeleitet, falls der Pfad dorthin frei ist. Alle anderen Übertragungseinheiten folgen ähnlich wie bei einer Pipeline-Verarbeitung. Im Gegensatz zum Store-and-forward-Modus werden die ankommenden Daten *nur im Konfliktfall* im Knoten vollständig zwischengespeichert. In jedem Knoten werden Puffer bereit gehalten, die auch ein maximal großes Nachrichtenpaket zwischenspeichern können.

- ***Wormhole-Routing*-Modus:** Dieser Modus ist, solange keine Übertragungskanäle blockiert sind, mit den Virtual-Cut-through-Modus identisch. Falls der Kopfteil der Nachricht auf einen Kanal trifft, der gerade belegt ist, wird er abgeblockt. Alle nachfolgenden Übertragungseinheiten der Nachricht verharren dann ebenfalls an ihrer augenblicklichen Position, bis die Blockierung aufgehoben ist. Durch das Verharren werden die Puffer nachfolgender Kanäle auch für weitere Nachrichten blockiert.

Beim Wormhole-Routing-Modus müssen die Puffer in den einzelnen Kanälen nur für die Aufnahme einer oder weniger Übertragungseinheiten ausgerüstet sein. Die Pufferung eines ganzen Pakets oder einer ganzen Nachricht, wie beim Virtual-Cut-through-Modus verlangt, ist nicht nötig. Allerdings verringern die bei großer Last auftretenden Blockierungen beim Wormhole-Routing-Modus die Übertragungseffizienz des Verbindungsnetzwerks. Deshalb wird häufig auch eine ***Buffered Wormhole Routing*** genannte Kompromisslösung zwischen Virtual-Cut-through- und Wormhole-Routing-Modus eingesetzt, bei der ein begrenzter Puffer zur Aufnahme kleinerer Pakete vorhanden ist, und größere Pakete im Blockierungsfall – ähnlich dem Wormhole-Routing-Modus – in den Puffern mehrerer Knoten zwischengespeichert werden.

Klassifizierung der Verbindungsnetze

Im folgenden wird eine Klassifizierung der Verbindungsnetze (s. Abbildung 4.3) vorgestellt. In dieser Klassifizierung wird zwischen statischen und dynamischen Verbindungsnetzen unterschieden. Sowohl bei speicher- als auch bei nachrichtengekoppelten Multiprozessoren können die gleichen Verbindungsstrukturen gewählt werden. In den nächsten beiden Abschnitten werden entsprechend der Klassifizierung zunächst die statischen und dann die dynamischen Verbindungsnetze vorgestellt.

4.2.2 Statische Verbindungsnetze

Multiprozessorsysteme mit statischen Verbindungsnetzwerken können speichergekoppelte oder nachrichtengekoppelte Systeme sein. Bei den einzelnen Netzknoten kann es sich demnach um Prozessoren, Speichermodule oder Prozessoren mit lokalem Speicher handeln. Allerdings finden statische Verbindungsnetzwerke überwiegend bei nachrichtengekoppelten Multiprozessorsystemen Verwendung.

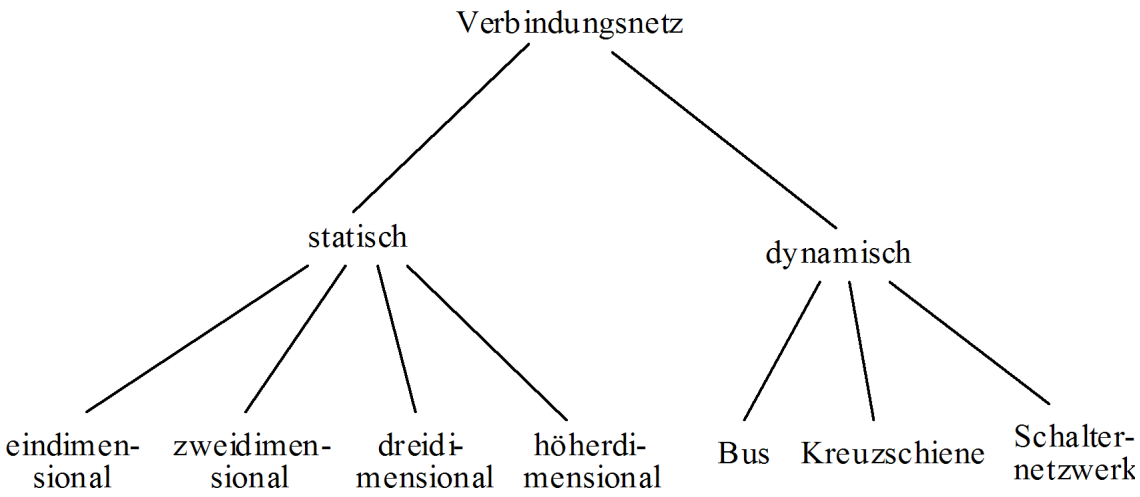


Abbildung 4.3: Klassifizierung von Verbindungsnetzen

Typische Verbindungstopologien für statische Verbindungsnetzwerke lassen sich intuitiv nach ihrer **Dimension** ordnen. In den folgenden Beispielen wird von einem anschaulichen Dimensionsbegriff ausgegangen:

Eindimensionales statisches Verbindungsnetzwerk

Ein **eindimensionales statisches Verbindungsnetzwerk** ist zum Beispiel die **Kette**, die in Abbildung 4.4 dargestellt ist. Wegen ihres großen Durchmessers ist die Kette als Verbindungsstruktur bei größeren Knotenzahlen uninteressant.

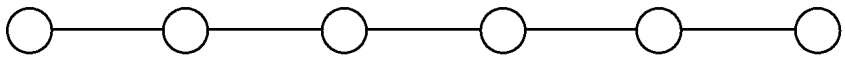


Abbildung 4.4: Kette

Zweidimensionale statische Verbindungsnetzwerke

Zweidimensionale statische Verbindungsnetzwerke sind zum Beispiel Ring, Chordaler Ring (Ring mit Sehnen), Stern, Baum, Gitter mit vier Nachbarknoten und Gitter mit acht Nachbarknoten, die in Abbildung 4.5 dargestellt sind.

Baum- und Stern-Netzwerke haben den Nachteil, dass die Wurzel, also der Verteilerknoten im Zentrum des Sterns, zu einem Flaschenhals für die Kommunikation wird. Die Stern-Topologie wird wegen der guten *Broadcast*-Eigenschaft bei zentralen Schaltern eingesetzt.

Mit dem so genannten ***Fat Tree*** lässt sich ein Baum-Netzwerk so konstruieren, dass dieser Nachteil nicht ins Gewicht fällt. Bei einem Fat Tree werden die Kommunikationswege mit um so größerer Kommunikationsbandbreite versehen, je näher sie bei der Wurzel liegen.

x-y-Routing

Gitterstrukturen sind wegen der Regelmäßigkeit des Verbindungsnetzes, der leichten Erweiterbarkeit, der unbegrenzten Skalierbarkeit und der einfachen technischen Implementierung durch kurze Leitungslängen vorteilhaft. Als statischer Wegewahlalgorithmus wird gerne das *x-y-Routing* angewandt. Dabei wird eine Nachricht zunächst so lange in x-Richtung geschickt, bis die entsprechende

Zielspalte erreicht ist. Dann geht es in y-Richtung bis zum Empfängerknoten weiter.

Der Durchmesser eines Gitters lässt sich halbieren, wenn das Gitter nach Art eines zweidimensionalen Torus zusätzlich an den Rändern verbunden ist. Beim zweidimensionalen Torus kann das x-y-Routing ebenfalls angewandt werden. Falls die Verbindungen darüber hinaus noch bidirektional sind, kann in x- und in y-Richtung jeweils nochmals eine positive oder negative Orientierung gewählt werden. Man benutzt für jede Verbindungsrichtung diejenige Orientierung, in der weniger Zwischenknoten vom Start- zum Zielknoten notwendig sind.

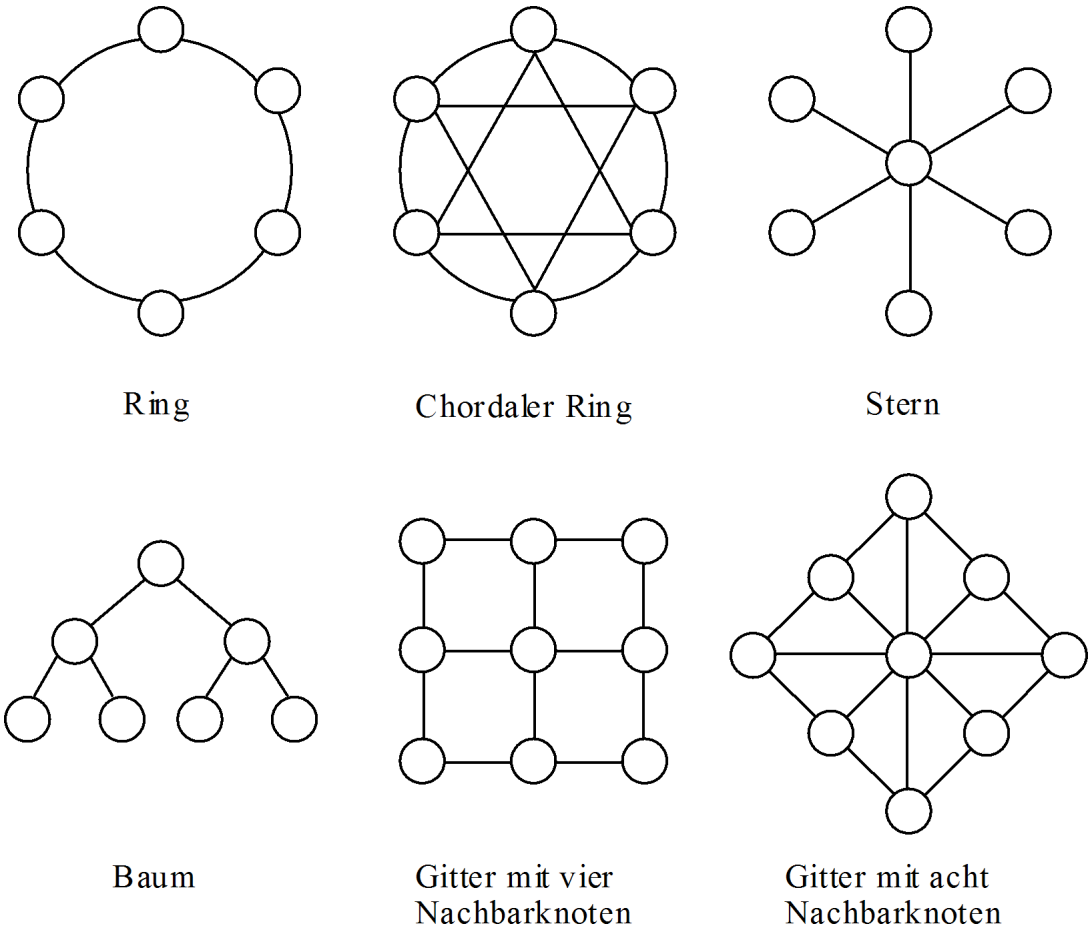


Abbildung 4.5: Ring, chordaler Ring, Stern, Baum, Gitter mit vier und Gitter mit acht Nachbarknoten

Drei- oder mehrdimensionale statische Verbindungsnetzwerke sind zum Beispiel Pyramide und Würfel sowie daraus abgeleitete komplexere Strukturen. Beispiele dafür sind der Hyperkubus, ein n-dimensionaler Würfel ($n > 3$), und das Ring-Würfel-Netzwerk, bei dem die Ecken eines Würfels durch Ringe aus Netzknoten ersetzt werden. Aus Platzgründen können wir auf diese Verbindungsnetzwerke nicht näher eingehen.

4.2.3 Dynamische Verbindungsnetze

Multiprozessorsysteme mit einem Einfachbus

Multiprozessorsysteme mit einem Einfachbus sind die am häufigsten angewandte Verbindungsstruktur für symmetrische Multiprozessoren. Der Multiprozessorbus koppelt typischerweise eine kleine Anzahl von zwei bis acht Prozessoren mit einem globalen Speicher. Diese Beschränkung der Prozessoren ist notwendig, da die Kommunikationsleistung auf einem Bus mit der Anzahl der zu übertragenden Daten stark sinkt. Außerdem stellt die verwendete Busschiene normalerweise nur eine beschränkte Anzahl von Einschubplätzen zur Verfügung.

Abbildung 4.6 zeigt ein speichergekoppeltes Multiprozessorsystem mit einem Einfachbus. Fast alle Arbeitsplatzrechner sind heute als speichergekoppelte Einfachbus-Systeme angelegt, wobei als Prozessorknoten Standardmikroprozessoren eingesetzt werden.

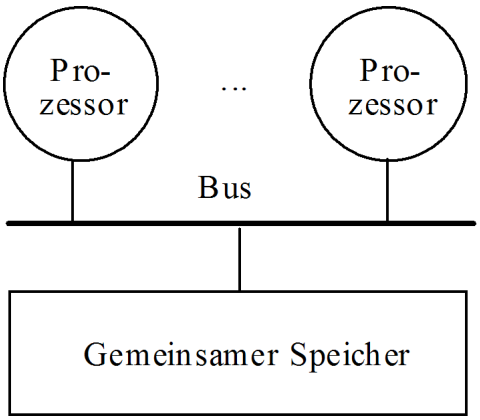


Abbildung 4.6: Speichergekoppelter Multiprozessor mit einem Einfachbus

Alle heutigen Mikroprozessoren besitzen bereits kleine On-Chip-Cache-Speicher als Primär-Cache-Speicher, welche durch zusätzliche Sekundär-Cache-Speicher innerhalb oder außerhalb des Prozessor-Chips ergänzt werden können (s. Abbildung 4.7). Über den Bus werden dann nur ganze Cache-Blöcke von meist 32 bis 128 Byte in mehreren hintereinander ablaufenden Buszyklen übertragen. Diesem Modell entsprechen auch die heute verfügbaren Mehrkernprozessoren Intel Core Duo und AMD64 X2.

Multiprozessorsysteme mit einem Mehrfachbussystem

Bei **Multiprozessorsystemen mit einem Mehrfachbussystem** verbinden mehrere Busse jeden Prozessor mit dem Speicher (oder mit anderen Prozessoren). Diese Kopplungsstruktur eignet sich besonders für fehlertolerante Systeme. Abbildung 4.8 zeigt ein speichergekoppeltes Multiprozessorsystem mit einem parallel angeordneten Mehrfachbus.

Mehrfachbussysteme können aber auch hierarchisch angeordnet sein. Solche Systeme werden auch als **Cluster-Bussysteme** bezeichnet.

Die Busse eines Mehrfachbussystems können auch so angeordnet sein, dass sie Prozessorknoten, die in Form eines zwei- bzw. dreidimensionalen Netzes angeordnet sind, in x-, y- und eventuell sogar in z-Richtung verbinden. Im zweidimensionalen Fall ist jeder Prozessor mit zwei, im dreidimensionalen Fall

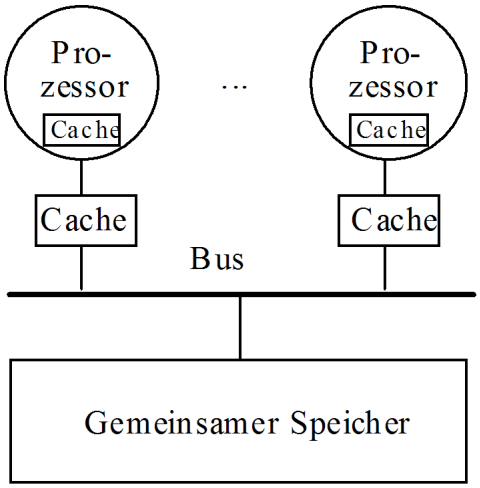


Abbildung 4.7: Speichergekoppelter Multiprozessor mit einem Einfachbus und Cache-Speichern

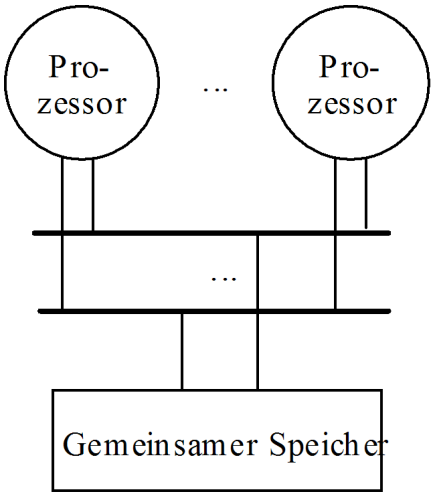


Abbildung 4.8: Speichergekoppelter Multiprozessor mit einem parallel angeordneten Mehrfachbus

mit drei Bussen verbunden. Auch in diesen Fällen hat man Clusterbussysteme und aus Sicht jedes Prozessors eine hierarchische Anordnung.

Unter einer **Kreuzschiene** oder einem **Kreuzschienenverteiler** (*Crossbar Switch*) versteht man eine Hardware-Einrichtung, die so geschaltet werden kann, dass in einer Menge von Prozessoren alle möglichen disjunkten Paare von Prozessoren gleichzeitig und blockierungsfrei miteinander kommunizieren können. Dasselbe gilt, wenn eine Kreuzschiene eine Menge von Prozessoren mit einer Menge von Speichermodulen verbindet. In Abhängigkeit vom Zustand der Schaltelemente im Kreuzschienenverteiler können dann je zwei beliebige Elemente aus den verschiedenen Mengen miteinander kommunizieren. Abbildung 4.9 zeigt ein speichergekoppeltes und ein nachrichtengekoppeltes Multiprozessorsystem mit einem Kreuzschienenverteiler. Ein mit einem Kreuzschienenver-

teiler realisiertes Verbindungsnetz ist nichtblockierend. Der Hardware-Aufwand ist jedoch sehr hoch: Falls jede der beiden Mengen n Elemente umfasst, benötigt man n^2 Schaltelemente. Häufig werden Verbindungseinrichtungen, die intern durch Schalternetzwerke (siehe weiter unten) aufgebaut sind, ebenfalls als Kreuzschienen bezeichnet. Dies ist jedoch nicht richtig, da es zu Blockierungen kommen kann.

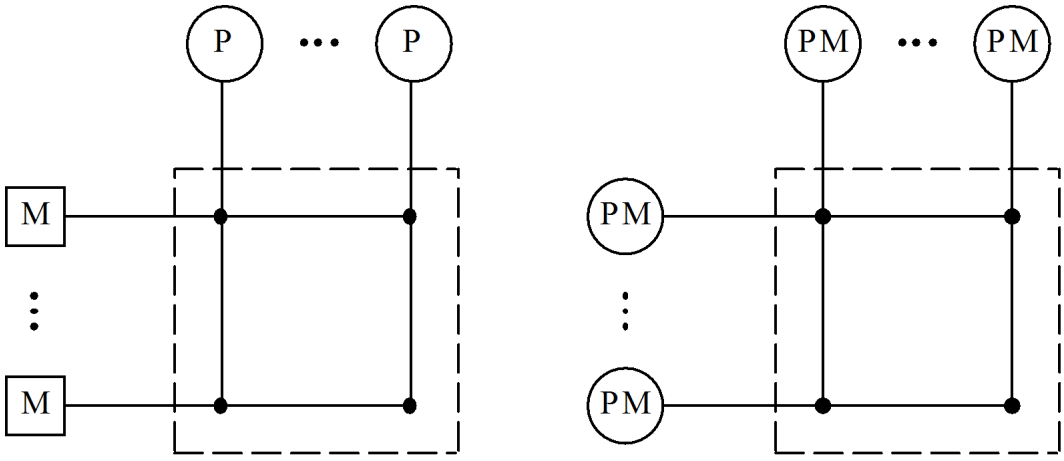


Abbildung 4.9: Speichergekoppelter und nachrichtengekoppelter Multiprozessor mit einem Kreuzschienenverteiler

Multiprozessorsysteme mit einem Schalternetzwerk

Multiprozessorsysteme mit einem Schalternetzwerk: Um den hohen Hardware-Aufwand eines Kreuzschienenverteilers zu vermeiden, kann ein Verbindungsnetz durch ein Schalternetzwerk realisiert werden. Die Schaltelemente eines Schalternetzwerks bestehen beispielsweise aus **Zweierschaltern** mit zwei Eingängen und zwei Ausgängen, die entweder durchschalten oder die Ein- und Ausgänge überkreuzen können wie in Abbildung 4.10 dargestellt.

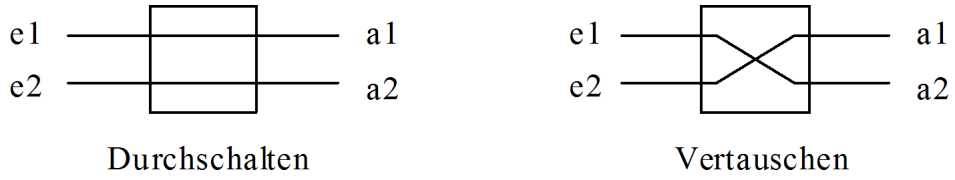


Abbildung 4.10: Schalterstellungen von Zweierschaltern

Diese Zweierschalter sind als Grundlage zum Aufbau von Netzen gut geeignet, da sie nur zwei Zustände kennen und somit nur ein Bit benötigen, um die Schaltfunktion zu steuern. Häufig werden jedoch auch andere Schaltelemente verwendet, nämlich Schaltelemente mit einer größeren Anzahl von Eingängen und Ausgängen (häufig Viererschalter) oder Schaltelemente mit zusätzlichen Schalterstellungen, beispielsweise zur Verbindung von einem Eingang mit mehreren Ausgängen (*Broadcast*).

Für Verbindungsnetze auf der Basis von Zweierschaltern gibt es auch die Bezeichnung **Permutationsnetze**. Dieser Name weist auf die Eigenschaft hin,

dass p Eingänge des Netzes gleichzeitig auf p Ausgänge geschaltet werden können und somit eine Permutation[3] der Eingänge erzeugt wird. **Einstufige Permutationsnetze** enthalten eine einzelne Spalte von Zweierschaltern, **mehrstufige Permutationsnetze** enthalten mehrere solcher Spalten. Die einzelnen Spalten von Zweierschaltern werden auch als „Stufen des Permutationsnetzwerks“ bezeichnet.

Es existiert eine ganze Reihe von häufig benutzten Grundmustern. Von diesem soll im folgenden nur ein einziges dargestellt werden. Dabei handelt es sich um das so genannte **Omega-Netzwerk**. Das Netzwerk für $p = 2^n$ Ein-/Ausgänge umfasst $n = \mathrm{ld}\, p$ Stufen von Zweierschaltern. Ein **speichergekoppeltes Omega-Netzwerk** verbindet Prozessoren mit Speichermodulen, wie in Abbildung 4.11 dargestellt.

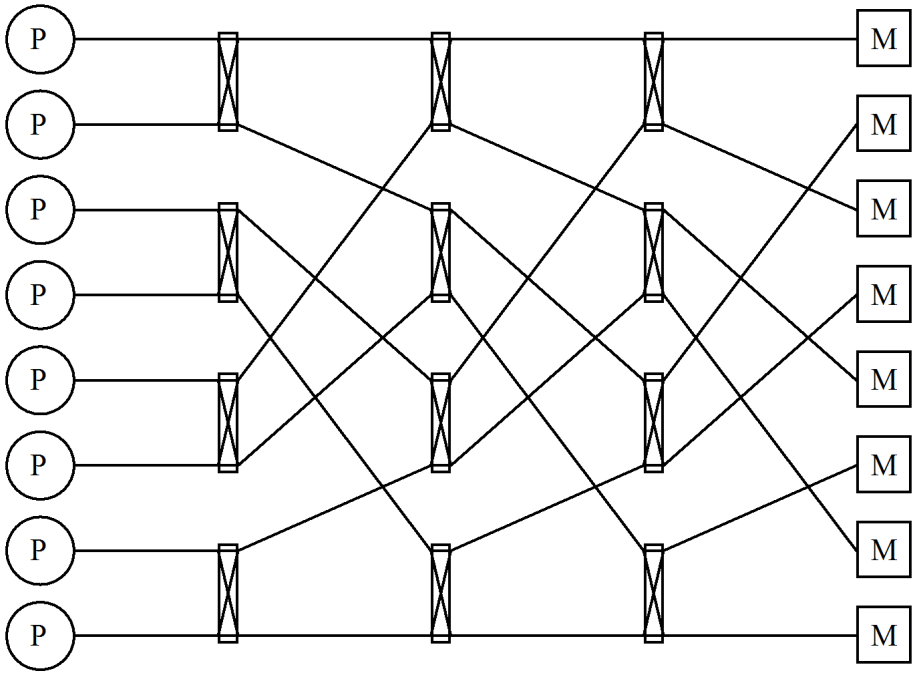
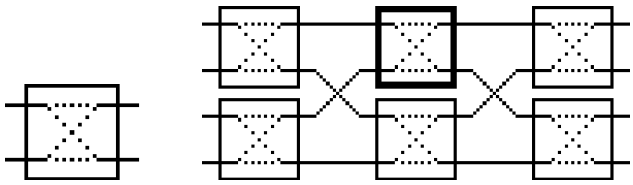


Abbildung 4.11: Speichergekoppeltes Omega-Netzwerk mit 8 Eingängen und 8 Ausgängen

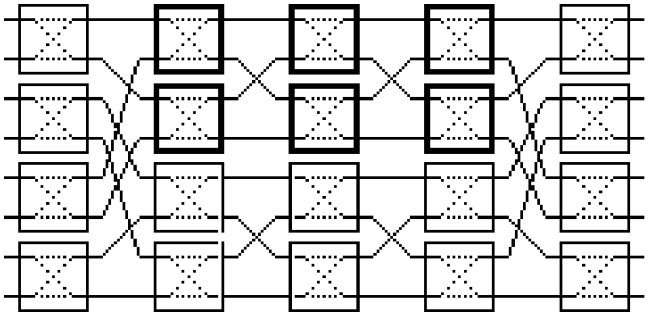
Die Gesamtzahl der Zweierschalter in einem Omega-Netzwerk mit $p = 2^n$ Ein-/Ausgängen beträgt $(p/2) \cdot \mathrm{ld}\, p$. Diese relativ geringe Anzahl von Schaltern ist der Grund dafür, dass Omega-Netzwerke blockierend sind. Beispielsweise erlaubt ein Netz mit $p = 8$ Ein-/Ausgängen und damit 12 Zweierschaltern nur $2^{12} = 4096$ unterschiedliche Zuordnungen (Permutationen) von 8 Eingängen zu 8 Ausgängen gegenüber der Gesamtzahl von $8! = 40320$ insgesamt möglichen Zuordnungen (Permutationen) von 8 Eingängen zu 8 Ausgängen.

[3]Unter einer Permutation von n Elementen versteht man in der Mathematik eine bijektive Abbildung einer Menge $\{a_1, a_2, \ldots, a_n\}$ auf sich. Definitionsbereich und Wertebereich stimmen überein. Bei einem Permutationsnetz werden jedoch die Ausgänge einer Stufe normalerweise auf die Eingänge der nachfolgenden Stufe geführt, wobei die beiden Stufen *nicht* identisch sind. Eigentlich handelt es sich hier lediglich um eine bijektive Abbildung der Menge der Ausgänge einer Stufe auf die Menge der Eingänge der nachfolgenden Stufe.

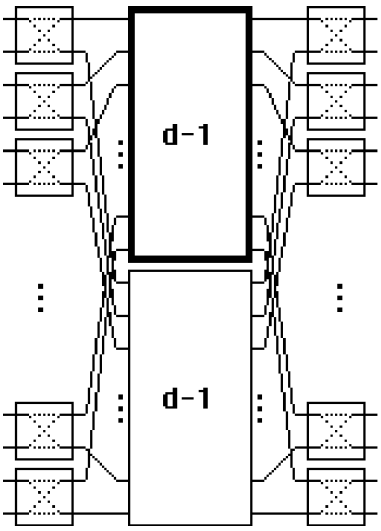
Selbsttestaufgabe 4.2 *Um mehrere Prozessoren eines Multiprozessorsystems miteinander zu vernetzen kann man Verbindungsnetzwerke aus Zweierschaltern verwenden. Das Benesnetzwerk ist ein solches Verbindungsnetzwerk mit 2^d Eingängen und 2^d Ausgängen. Es hat folgende Struktur:*



$d = 1$ $d = 2$



$d = 3$



Rekursiver Aufbau

a) Wieviele Zeilen und Spalten hat ein Benesnetzwerk der Ordnung d?

b) Zeigen Sie, dass das Benesnetzwerk universell ist, d.h. jede Permutation erzeugen kann. Es empfiehlt sich ein konstruktiver Induktionsbeweis über d.

c) Konstruieren Sie für $d = 3$ eine Schalterstellung, welche die Permutation $(3, 0, 1, 2, 4, 6, 5, 7)$ erzeugt.

4.3 Speichergekoppelte Multiprozessoren

4.3.1 Modelle speichergekoppelter Multiprozessoren

Speichergekoppelte Multiprozessoren

Bei **speichergekoppelten Multiprozessoren** besitzen alle Prozessoren einen gemeinsamen Adressraum. Je nach physikalischer Speicherorganisation unterscheidet man **symmetrische Multiprozessoren** (*Symmetric Multiprocessor* – SMP), bei denen gleichartige Prozessoren über einen Bus (oder eine Kreuzschiene) mit einem globalen Speicher verbunden sind, und ***Distributed-Shared-Memory*-Systeme** (DSM), bei denen ein gemeinsamer Adressraum besteht, obwohl die Speicher physikalisch auf die einzelnen Verarbeitungselemente verteilt sind.

Kommunikation und Synchronisation geschehen über gemeinsame Variablen. So kann beispielsweise bei einer Leser-/Schreiber-Kommunikation der „sendende“ Prozessor Daten in den gemeinsamen Speicher schreiben, und der „empfangende“ Prozessor liest die Daten. Natürlich ist hier zusätzlich eine Synchronisation notwendig, denn der „empfangende“ Prozessor muss so lange warten, bis der „sendende“ Prozessor die Daten im gemeinsamen Speicher bereitgestellt hat (Leser-Schreiber-Problem). Außerdem darf auf gemeinsame Daten nicht gleichzeitig von verschiedenen Prozessoren lesend und schreibend zugegriffen werden (kritischer Bereich). Programmtechnisch geschieht die Synchronisation durch Schloss- (*Mutex*), Semaphor- oder Bedingungsvariablen (*Condition*) und darauf definierte atomare Operationen. Auf speichergekoppelten Multiprozessoren werden deshalb Hardware-Mechanismen wie beispielsweise *Test-and-set*- oder *Swap*-Befehle zur Implementierung der Synchronisationsoperationen bereitgestellt.

Speichergekoppelte Systeme gelten als einfacher programmierbar als nachrichtengekoppelte Systeme. Die nutzbare Parallelität reicht von der Programmebene bis zur Blockebene. Parallelisierende Compiler sind ebenfalls einsetzbar, wobei die Parallelisierung in der Regel auf parallel ausführbare Schleifeniterationen beschränkt ist.

UMA und NUMA

Je nach Speicheranordnung werden bei speichergekoppelten Multiprozessoren folgende Modelle unterschieden:

- ***Uniform-Memory-Access*-Modell (*UMA*):** Alle Prozessoren greifen gleichermaßen auf einen gemeinsamen Speicher zu. Insbesondere ist die Zugriffszeit aller Prozessoren auf den gemeinsamen Speicher gleich. Jeder Prozessor kann zusätzlich einen lokalen Cache-Speicher besitzen.

- ***Non-Uniform-Memory-Access*-Modell (*NUMA*):** Die Zugriffszeiten auf Speicherzellen des gemeinsamen Speichers variieren je nach Ort, an dem sich die Speicherzelle befindet. Die Speichermodule des gemeinsamen Speichers sind physikalisch auf die Prozessoren aufgeteilt. Der Zugriff auf das „lokale“ Speichermodul eines Prozessors ist schneller als auf ein „entferntes“ Speichermodul. Alle „lokalen“ Speichermodule bilden zusammen einen gemeinsamen Adressraum – im Gegensatz zu den lokalen Speichern eines nachrichtengekoppelten Multiprozessors. Häufig wird eine ganze Hierarchie von unterschiedlichen Zugriffszeiten durch die Struktur des Verbindungsnetzes hervorgerufen.

Neben der Art der Verbindungsstruktur und der dadurch entstehenden unterschiedlichen Zugriffszeiten auf die gemeinsam adressierbaren Speichermodule kann man auch die Organisation der Cache-Speicher in die Klassifikation mit einbeziehen. Alle UMA- und NUMA-Systeme besitzen heute prozessorlokale Cache-Speicher, da Standard-Mikroprozessoren bereits On-Chip-Cache-Speicher enthalten und diese durch zusätzliche Sekundär-Cache-Speicher erweitert sein können. UMA-Systeme sind Cache-kohärent organisiert und nutzen das Bus-Schnüffel-Verfahren (siehe nächsten Abschnitt). Beim NUMA-Modell kann man folgende Unterscheidungen treffen:

- Falls die Cache-Speicher über das gesamte NUMA-System hinweg cache-kohärent organisiert sind, spricht man von einem ***Cache-Coherent Non-Uniform-Memory-Access*-Modell** (***CC-NUMA***). In einem CC-NUMA wird bei einem Zugriff auf ein entferntes Speicherwort, der einen Cache-Fehlzugriff auslöst, ein gesamter Cache-Block aus dem entfernten Speicher in den lokalen Cache-Speicher übertragen. Mittels eines verzeichnisbasierten Cache-Kohärenzprotokolls (*Directory-based Coherence Protocol*) wird die Cache-Kohärenz gesichert.

- ***Cache-only-Memory-Architecture*-Modell** (***COMA***): Das COMA-Modell ist ein Spezialfall des CC-NUMA-Modells, wobei die physikalisch verteilten Speichermodule Cache-Speicher darstellen, und der Speicher des gesamten Rechners somit nur aus Cache-Speichern besteht. Alle Cache-Speicher besitzen einen gemeinsamen Adressraum. Der Zugriff auf den Cache-Block eines entfernten Cache-Speichers wird durch verteilte Cache-Verzeichnisse unterstützt. Während beim NUMA-Modell die anfängliche Aufteilung der Daten auf die physikalischen Speichermodule von großer Bedeutung für die Laufzeit eines parallelen Algorithmus ist, da die Daten fest lokalisiert bleiben, wandern beim COMA-Modell die Daten in den Cache-Speicher des Prozessors, der sie benötigt.

- Wenn eine Cache-Kohärenz über das gesamte NUMA-System hinweg *nicht* gewährleistet ist, kann man ein solches System als ***Non-Cache-Coherent Non-Uniform-Memory-Access*-Modell** (***NCC-NUMA***) bezeichnen. In einem NCC-NUMA wird zwischen lokalen Zugriffen, welche über den Cache-Speicher gehen, und Zugriffen auf entfernte Speicherwörter unterschieden. Die letzteren werden am Cache-Speicher vorbeigeleitet. Die Granularität des entfernten Speicherzugriffs kann ein einzelnes Speicherwort, die Größe eines Cache-Blocks oder die Größenordnung von ganzen Speicherblöcken sein. Für die Kohärenz der parallelen Programme muss die Software durch Schlossvariablen- oder Barrieren-Synchronisation sorgen.

4.3.2 Cache-Kohärenz und Speicherkonsistenz

Falls in einem Multiprozessorsystem mehrere Prozessoren mit jeweils eigenen Cache-Speichern unabhängig voneinander auf Speicherwörter des Hauptspeichers zugreifen können, entstehen Gültigkeitsprobleme. Mehrere Kopien des gleichen Speicherworts müssen miteinander in Einklang gebracht werden.

Eine Cache-Speicherverwaltung heißt **(Cache-)kohärent,** falls ein Lesezugriff immer den Wert des zeitlich letzten Schreibzugriff auf das entsprechende Speicherwort liefert.

Kohärenz bedeutet das korrekte Voranschreiten des Systemzustands durch ein abgestimmtes Zusammenwirken der Einzelzustände. Im Zusammenhang mit einem Cache-Speicher muss das System dafür sorgen, dass immer die aktuellen und nicht die veralteten Daten gelesen werden.

Ein System ist **konsistent**, wenn alle Kopien eines Speicherworts im Hauptspeicher und den verschiedenen Cache-Speichern *identisch* sind. Dadurch ist auch die Kohärenz sichergestellt.

Bus-Schnüffeln und MESI-Protokoll, s. Kapitel 3

In symmetrischen Multiprozessoren, bei denen mehrere Prozessoren mit lokalen Cache-Speichern über einen Systembus an einen gemeinsamen Hauptspeicher angeschlossen sind, verwendet man das so genannte **Bus-Schnüffeln** (*Bus Snooping*). Als Cache-Kohärenzprotokoll in Zusammenarbeit mit dem Bus-Schnüffeln hat sich das **MESI-Protokoll** durchgesetzt. Üblicherweise wird bei symmetrischen Multiprozessoren ein Write-invalidate-Cache-Kohärenzprotokoll, oft das MESI-Protokoll, mit Rückschreibeverfahren eingesetzt.

In DSM-Multiprozessorsystemen kann nicht wie in SMP-Systemen die Broadcast-Eigenschaft des Busses genutzt werden. Bei DSM-Systemen, die als CC-NUMA-Systeme einzuordnen sind, wird deshalb die Cache-Kohärenz über Verzeichnis-Tabellen (*Directory Tables*) hergestellt. Diese Tabellen können vollständig in Hardware, in Software oder in einer Kombination von beiden implementiert werden. Die Tabellen können zentral gehalten oder über die einzelnen Prozessorknoten verteilt werden. Meist wird die letztere Variante angewandt, wobei eine Tabelle immer dort angeordnet ist, wo auch der zugehörige lokale Speicher steht. Die Tabelle protokolliert für jeden Blockrahmen im lokalen Speicher, ob die zugehörige Speicherportion in den lokalen oder einen entfernten Cache-Speicher als Cache-Block übertragen wurde. Dabei werden nicht nur die betreffenden Cache-Speicher, sondern auch die Zustände der Cache-Block-Kopien festgehalten.

Die Zustände, Zustandsänderungen und Transaktionen sind meist ähnlich denjenigen des MESI-Protokolls definiert. Beispielsweise werden vor einem Schreibzugriff *Invalidate*-Nachrichten an alle Cache-Speicher gesandt, welche eine Kopie des Cache-Blocks besitzen.

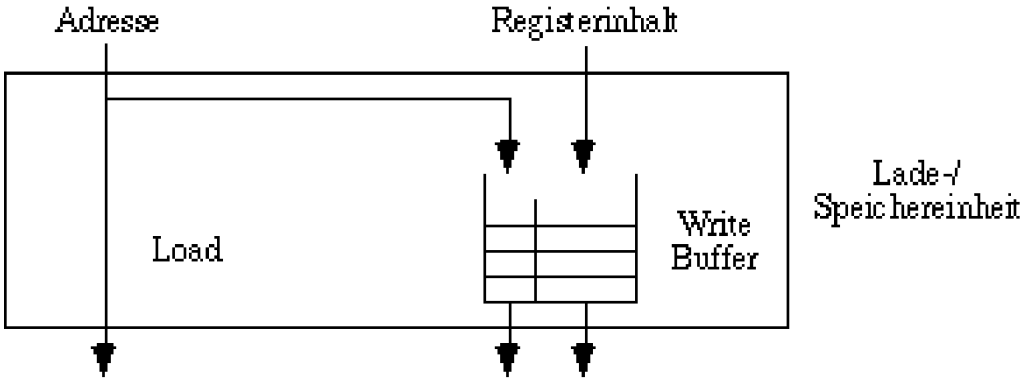
4.3.3 Speicherkonsistenzmodelle

Die Lade-/Speichereinheit eines Prozessors führt alle Datentransfer-Befehle zwischen den Registern und dem Daten-Cache-Speicher durch. Cache-Kohärenz wird erst dann wirksam, wenn ein Lade- oder Speicherzugriff durch die Lade-/Speichereinheit des Prozessors ausgeführt wird, also ein Zugriff auf den Cache-Speicher geschieht. Cache-Kohärenz sagt nichts über mögliche Umordnungen der Lade- und Speicherbefehle *innerhalb* der Lade-/Speichereinheit aus. Heutige Mikroprozessoren führen jedoch die Lade- und Speicherbefehle nicht mehr unbedingt in der Reihenfolge aus, wie sie vom Programm her vorgeschrieben ist. Als Beispiel betrachte man die Implementierung der Lade-/Speichereinheit in Abbildung 4.12. Ladebefehle werden sofort ausgeführt. Speicherbefehle wer-

den zunächst in einem internen, als FIFO organisierten Schreibpuffer (*Write Buffer*) der Lade-/Speichereinheit untergebracht.

Konzept der Ladezugriffe vor Speicherzugriffen, s. auch Kapitel 2

Während ein Speicherbefehl auf seinen Datenwert wartet, kann ein nachkommender Ladebefehl ihn überholen und vor ihm auf den Daten-Cache-Speicher zugreifen. Zuerst wird aber überprüft, dass der Lade- und der Speicherbefehl nicht dieselbe Zieladresse haben. Damit wird verhindert, dass statt eines abzuspeichernden Datenwerts, der im Schreibpuffer hängt und noch nicht geschrieben wurde, von einem in der Programmordnung nachfolgenden Lesebefehl fälschlicherweise ein veralteter Datenwert aus dem Cache-Speicher gelesen wird. Das Prinzip, dass Ladezugriffe vor Speicherzugriffe gezogen werden, sofern nicht dieselbe Adresse betroffen ist und kein Spezialbefehl (Synchronisationsbefehl, *swap*-Befehl oder markierter Befehl) dazwischenliegt, ist in vielen Mikroprozessoren verwirklicht.



Ladezugriffe werden vor Speicherzugriffen ausgeführt

Abbildung 4.12: Lade-/Speichereinheit

Siehe auch Kapitel 3

Das Konzept des **nichtblockieren Cache-Speichers** (*Non-Blocking Cache, Lock-up free Cache*) geht sogar noch weiter: Im Falle eines Cache-Fehlzugriffs können nachfolgende Befehle, die nicht denselben Cache-Block benötigen, auf den Cache-Speicher zugreifen, ohne dass auf die Ausführung des den Fehlzugriff auslösenden Befehls gewartet werden muss.

Die Prinzipien der vorgezogenen Ladezugriffe und des nichtblockierenden Cache-Speichers steigern die Verarbeitungsgeschwindigkeit des Prozessors, da neue Daten so schnell als möglich in die Register geladen werden, damit nachfolgende Verarbeitungsbefehle ausgeführt werden können – das Rückspeichern erscheint demgegenüber nicht so zeitkritisch. Wesentlich ist dabei bei beiden Prinzipien, dass aus Speichersicht die Programmordnung nicht mehr eingehalten wird. Unter der angegebenen Randbedingung – dem lokalen Adressabgleich der betroffenen Datenwerte – bleibt jedoch die Ausführung außerhalb der Programmreihenfolge ohne Auswirkung auf das Ergebnis, sofern nur ein einzelner Prozessor beteiligt ist.

Dies gilt jedoch nicht mehr bei einer parallelen Programmausführung auf einem Multiprozessor. Bei speichergekoppelten Multiprozessoren können im Prin-

zip zu jedem Zeitpunkt mehrere Zugriffe auf denselben Speicher erfolgen. Bei SMP-Systemen können trotz Cache-Kohärenz durch Verwendung von vorgezogenen Ladezugriffen oder durch nichtblockierende Cache-Speicher die Speicherzugriffsoperationen in anderer Reihenfolge als durch das Programm definiert am Speicher wirksam werden.

Das folgende abgewandelte Beispiel von **Dekkers Algorithmus** zeigt die möglicherweise auftretenden, unerwarteten Effekte: Dekkers Algorithmus

```
y=0; x=0; ... process p1: x=1;
            if (y==0)
            ... fuehre Aktion a2 aus

process p2: y=1;
            if (x==0)
            ... fuehre Aktion a1 aus
```

Beim Programmablauf sind folgende vier Fälle möglich:

- a1 wird ausgeführt, a2 wird nicht ausgeführt;
- a2 wird ausgeführt, a1 wird nicht ausgeführt;
- a1 und a2 werden beide nicht ausgeführt;
- a1 und a2 werden beide ausgeführt.

Die Erwartungshaltung des Programmierers würde hier den vierten Fall nicht in Betracht ziehen, da dieser nur dann auftreten kann, wenn die Programmordnungen der beiden Prozesse verletzt werden. Wenn jedoch vom Prozessor Lade- vor Speicherbefehle gezogen oder nichtblockierende Cache-Speicher verwendet werden, dann kann auch der vierte Fall eintreten. Auch Cache-Kohärenz ändert nichts an dieser Tatsache. Man beachte, dass für ein korrektes Programm die Zugriffe auf die gemeinsamen Variablen x und y einen kritischen Bereich darstellen und durch Synchronisationsoperationen geschützt werden sollten.

Ein weiteres Problem kann bei Distributed-shared-memory-Systemen auftreten, wenn die Speicherbefehle nicht **atomisch** sind, d.h., der neue Wert wird nicht überall gleichzeitig wirksam. Bei DSM-Systemen geschieht der Speicherzugriff über das Verbindungsnetz und dadurch mit einer durch das Netz gegebenen Latenz, die je nach Zugriffspfad erheblich variieren kann (NUMA-Modell). Dadurch kann es zu einem Wettlauf zwischen den einzelnen Speicherzugriffen kommen. Eine Änderung in einer Kopie der gleichen Speicherzelle wird in einem entfernten Speicher später als im lokalen Speicher durchgeführt. Ein zeitlich nachfolgender Lesezugriff eines anderen (weiter entfernten) Prozessors könnte dann (von der entfernten Speicherzelle) noch den alten Wert lesen, und ein zeitlich noch späterer Lesezugriff eines näher gelegenen Prozessors würde wiederum den neuen Wert (aus der näher gelegenen Kopie) lesen. Zugriffe, die durchaus in der richtigen Reihenfolge initiiert worden sind, können sich bei diesem Wettlauf überholen, wodurch die vorgeschriebene Ordnung der Speicherzugriffe und damit die Semantik der Programmausführung verändert wird.

Man lässt dieses „Chaos“ entweder in kontrolliertem Maße zu oder man kann es durch zusätzlichen Rechenaufwand unterbinden, indem man ein verzeichnisbasiertes Cache-Kohärenzverfahren verwendet, das die so genannte sequentielle Konsistenz (siehe nächsten Abschnitt) beachtet.

Konsistenzmodell

Aus Sicht des Programmierers ist weniger die implementierungstechnische Seite der Cache-Kohärenz interessant, sondern das zugrundeliegende **Konsistenzmodell**, das etwas über die zu erwartende Ordnung der Speicherzugriffe durch parallel arbeitende Prozessoren aussagt. Genauer gesagt:

*Ein **Konsistenzmodell** spezifiziert die Reihenfolge, in der Speicherzugriffe eines Prozesses von anderen Prozessen gesehen werden.*

sequenzielle Konsistenz

Im Normalfall setzt der Programmierer die **sequenzielle Konsistenz** voraus, die jedoch zu starken Einschränkungen bei der Implementierung führt. Insbesondere verbietet sie die Prinzipien der vorgezogenen Ladezugriffe und des nichtblockierenden Cache-Speichers. Es gibt jedoch auch schwächere Konsistenzmodelle, welche Konsistenz nur zum Zeitpunkt einer Synchronisationsoperation gewährleisten. Lese- und Schreibzugriffe der parallel arbeitenden Prozessoren auf den gemeinsamen Speicher zwischen den Synchronisationszeitpunkten können in beliebiger Reihenfolge geschehen. Im Rahmen dieses Kurses kann aus Platzgründen auf die verschiedenen Konsistenzmodelle nicht eingegangen werden. Als ergänzende Lektüre sei insbesondere auf [7], [8], [10] hingewiesen.

4.3.4 Distributed-shared-memory-Multiprozessoren

DSM-Multiprozessoren (*Distributed Shared Memory*) besitzen einen allen Prozessoren gemeinsamen Adressraum, die einzelnen Speichermodule sind jedoch auf die einzelnen Prozessoren verteilt. Der Zugriff eines Prozessors auf ein Datenwort in seinem lokalen Speicher geschieht schneller als der Zugriff auf ein Datenwort, das in einem der lokalen Speicher eines anderen Prozessors steht, weshalb DSM-Systeme ins NUMA-Modell eingeordnet werden.

In der Regel werden zusätzliche Cache-Speicher (prozessorinterne Primär-Cache-Speicher und optionale Sekundär-Cache-Speicher) verwendet. Diese können die Cache-Kohärenz über sämtliche Cache-Speicher des gesamten Systems sichern (CC-NUMA und COMA) oder sie puffern Cache-Blöcke nur im Falle eines prozessorlokalen Speicherzugriffs (NCC-NUMA).

Im ersten Fall wird ein verzeichnisbasiertes Cache-Kohärenzverfahren (siehe Abschnitt 3.1.3) eingesetzt, wobei entweder die sequenzielle Konsistenz zugesichert wird oder aus Effizienzgründen auf ein abgeschwächtes Konsistenzmodell übergegangen wird, bei dem die Speicherkonsistenz nur zum Zeitpunkt bestimmter Synchronisationsoperationen gewährleistet ist.

zwei Arten von DSM-Multiprozessoren

Entsprechend der Organisation des Zugriffs auf einen entfernten Speicher kann man zwei Arten von Distributed-shared-memory-Multiprozessoren unterscheiden:

- Der Zugriff geschieht in einer für das Maschinenprogramm transparenten Weise oder
- durch explizite Befehle, die nur dem entfernten Speicherzugriff dienen.

Die erste Variante ist bei CC-NUMAs üblich, die zweite Variante bei NCC-NUMA-Systemen. Im ersten Fall muss Spezial-Hardware anhand der Adresse zwischen einem prozessorlokalen und einem entfernten Speicherzugriff unterscheiden. Diese Spezial-Hardware ist bei CC-NUMA-Systemen meist mit der Überwachung der Cache-Verzeichnisse gekoppelt. Im zweiten Fall wird der Maschinenbefehlssatz des Prozessors erweitert.

DSM-Multiprozessoren stellen eine Kombination aus speicher- und nachrichtengekoppelten Systemen dar. Mit dem Distributed-shared-memory-Konzept soll die praktisch unbegrenzte Skalierbarkeit eines nachrichtengekoppelten Multiprozessorsystems mit der einfacheren Programmierbarkeit eines speichergekoppelten Systems kombiniert werden.

Der Begriff des „virtuell gemeinsamen Speichers“ (*Virtual Shared Memory*) benutzt die Bezeichnung „virtuell“ im Sinne von „so, als ob“ und fällt mit *Distributed Shared Memory* zusammen.

Die erste Implementierung des DSM-Konzepts geschah 1988 durch Softwaremechanismen auf einem Rechnernetz im Rahmen des IVY-Systems. Dafür wurden Mechanismen verwendet, welche die virtuelle Speicherverwaltung in herkömmlichen Betriebssystemen auf die Rechner eines lokalen Netzes ausdehnen. Falls ein Prozessor auf eine Seite zugreift, die sich nicht in seinem lokalen Speicher befindet, wird eine Unterbrechung (ein Seitenfehler der virtuellen Speicherverwaltung) ausgelöst und die Seite über das Verbindungsnetz aus dem lokalen Speicher des Prozessors, der die Seite gerade besitzt, nachgeladen. Es wurde der Begriff des **gemeinsamen virtuellen Speichers** (*Shared Virtual Memory*) geprägt, der sich jedoch gegenüber dem Oberbegriff *Distributed Shared Memory* nicht durchgesetzt hat.

In den folgenden Jahren wurde mit einer Vielzahl von software- oder hardwarebasierten DSM-Systemen experimentiert, bei denen der gemeinsame virtuelle Speicher in Seiten mit unterschiedlicher Granularität (von 16 Byte bei Dash bis zu 8 kB bei Mermaid) aufgeteilt ist. Bei den softwarebasierten DSM-Systemen auf Rechnernetzen zeigt sich die geringere Effizienz des Nachladens von Seiten über das Verbindungsnetz gegenüber dem schnellen Zugriff auf den Speicher eines Einprozessor- oder eines symmetrischen Multiprozessorsystems als nachteilig.

Bei der Wahl der Granularität der Seiten macht sich das so genannte *False Sharing* bemerkbar. Darunter versteht man den Effekt, dass verschiedene Datenwörter, die innerhalb einer Seite liegen, von verschiedenen Prozessoren z.B. für Schreibzugriffe benötigt werden. Da die Kohärenzmechanismen immer nur die gesamte Seite betreffen, muss die Seite vor jedem Schreibzugriff dem anderen Prozessor entzogen und anschließend erneut übertragen werden. Bei mehrfachen Schreibzugriffen kommt es nicht nur zu häufigen Blockierungen der Prozessoren, sondern auch zum so genannten **Flattern** (*Thrashing*), da die Seite immer wieder übers Netz übertragen werden muss. Das Problem des *False Sharing* lässt sich durch Verkleinern der Seitengröße mindern, da dadurch die Wahrscheinlichkeit, dass mehrere unabhängige Datenwörter auf einer gemeinsamen Seite liegen, verringert wird. Andererseits verringert eine größere Seite den Aufwand für die Seitenverwaltung.

Softwarebasierte DSM-Systeme, bei denen eine Betriebssystemerweiterung

einen gemeinsamen virtuellen Speicher vorspiegelt, haben sich wegen ihrer geringen Effizienz beim Nachladen von Seiten auch gegenüber den nachrichtengekoppelten Programmierumgebungen auf verteilten Systemen nicht durchsetzen können. Derzeit wird damit experimentiert, gemeinsame Variablenzugriffe bereits im Quellprogramm zu erkennen und per Precompiler entfernte Lese-/Schreibzugriffe direkt in den Programmcode einzusetzen. Durch Aufruf von Bibliotheksfunktionen werden nur die Datenobjekte verschoben, die benötigt werden. *False Sharing* wird damit ausgeschlossen, und das Konsistenzmodell kann per Software festgelegt werden.

4.4 Nachrichtengekoppelte Multiprozessoren

4.4.1 Nachrichtengekoppelte Multiproz. und verteilte Systeme

nachrichtengekoppelte Multiprozessoren

Bei den **nachrichtengekoppelten Multiprozessoren** gibt es keine gemeinsamen Speicher- oder Adressbereiche. Die Kommunikation geschieht durch Austauschen von Nachrichten über ein Verbindungsnetz. Alle Prozessoren besitzen nur lokale Speicher. In nachrichtengekoppelten Systemen sind die Prozessorknoten üblicherweise durch Punkt-zu-Punkt-Verbindungen gekoppelt. Die Kommunikation geschieht über serielle oder acht Bit breite Verbindungen, die einen relativ geringen Hardware-Aufwand benötigen.

Generationen

Die **nachrichtengekoppelten Multiprozessoren** lassen sich in verschiedene Generationen einteilen:

- Die **erste Generation** (1983–1987) ist durch einen softwaregesteuerten Nachrichtenaustausch charakterisiert. Dabei wird jede Nachricht, die zwischen nicht benachbarten Knoten übertragen werden muss, auf jedem der beteiligten Zwischenknoten vollständig gespeichert und dann erst weitergeleitet (*store-and-forward*). Speicherung und Weiterleitung wird von dem Prozessor des Zwischenknotens durchgeführt. Durch die langsame Kommunikation sind nur sehr grobkörnig parallele Algorithmen mit wenig Kommunikation effizient lauffähig.

- Die **zweite Generation** (1988–1992) besitzt eine hardwaregestützte Nachrichtenübermittlung. Bei der Übertragung über Zwischenknoten (*Multi-Hop Messages*) werden die Prozessoren der Zwischenknoten nicht von der Organisation der Nachrichtenweiterleitung belastet. Diese geschieht vollständig durch die Kommunikationshardware. Die Kommunikationsgeschwindigkeit zwischen zwei nicht benachbarten Knoten ist erheblich höher als bei den Multiprozessoren der ersten Generation. Meist werden hyperkubus- oder gitterartige Verbindungsstrukturen verwendet.

- Die **dritte Generation** (ab 1993) ist durch Kommunikationseinrichtungen auf den Prozessorchips charakterisiert sein, um durch die schnellere Kommunikation auch feinkörnig parallele Algorithmen effizient ausführen zu können. Insbesondere sollten Kommunikations- und Synchronisationsbefehle in stärkerem Maße als bei heutigen Prozessoren in den Befehlssatz

des Prozessors integriert sein. Dem widerspricht jedoch der Trend, Standardmikroprozessoren als Knoten für Multiprozessorsysteme einzusetzen, da diese billiger als eine Neuentwicklung sind. Die dritte Generation ist deshalb wohl eher in den Distributed-Shared-Memory-Systemen zu sehen, mit denen sich die Grenzen zwischen speichergekoppelten und nachrichtengekoppelten Multiprozessoren zu verwischen beginnen.

Die Skalierbarkeit ist bei nachrichtengekoppelten Multiprozessoren bei geeignetem Verbindungsnetz im Prinzip unbegrenzt. Nachrichtengekoppelte Multiprozessoren mit Tausenden von Prozessoren stellen deshalb heute die leistungsfähigsten Parallelrechner dar. Auf solchen Systemen lässt sich Parallelität in effizienter Weise auf Programm- oder Task-Ebene nutzen. Block- oder Anweisungsebenenparallelität sind nach heutiger Übertragungstechnologie auf nachrichtengekoppelten Multiprozessoren nicht effizient nutzbar. Parallelität auf Programmebene wird dadurch ermöglicht, dass jedem Programm eine Teilmenge der Prozessorknoten eines Systems zugeteilt werden. Man spricht hier von *Space Sharing* im Gegensatz zu dem bei Einprozessorsystemen üblichen *Time-Sharing*-Betrieb.

TERAFLOPS-Rechner

Wegen ihrer hohen Skalierbarkeit sind nachrichtengekoppelte Multiprozessoren auch die primären Kandidaten für TERAFLOPS-Rechner, die im Rahmen der auf 10 Jahre (1996 bis 2006) angelegten *Accelerated Strategic Computing Initiative* (*ASCI*) des für die Atombombenforschung der USA zuständigen *Department of Energy* entwickelt wurden. Von Seiten mehrerer Hersteller wurden und werden TERAFLOPS-Rechner gebaut und den Nuklearwaffenforschungsinstituten *Sandia National Laboratories*, *Los Alamos National Laboratory* und *Lawrence Livermore National Laboratory* zur Verfügung gestellt. Hauptaufgabe dieser TERAFLOPS-Rechner ist es, die extrem komplexen Berechnungen durchzuführen, die nötig sind, um die Auswirkungen der Alterung der bestehenden Nuklearwaffen und das Testen neuer Nuklearwaffen zu simulieren. Die Simulationen ersetzen unterirdische Atombombentests. Ziel ist es, jeweils im Abstand von 18 Monaten einen Rechner mit dreifacher Leistung zu installieren.

Als erster Rechner hat im Dezember 1996 ein ASCI-Red genannter Multiprozessor der *Intel Supercomputer Division* die Leistung von (theoretisch errechnet) einem TERAFLOPS erreicht. In seinem Endausbau wurde der Rechner mit 9624 PentiumPro-Prozessoren und einer theoretischen Höchstleistung von 1.8 TERAFLOPS an die *Sandia National Laboratories* geliefert. ASCI-Red besitzt einen Speicherausbau von 500 GB und kostete etwa 50 Millionen Dollar.

Die jeweils schnellsten Supercomputer der Welt werden in der Top500-Liste (siehe `http://www.top500.org/`) erfasst und auf der Basis der anhand des Linpack-Benchmarks gemessenen Maximalleistung in eine Rangliste gebracht. In der Liste vom November 2006 (siehe `http://www.top500.org/list/2006/11/`) führen auch Rechner obiger Institute, wobei die beiden Erstplatzierten die im ASCI-Projekt erwartete Leistung von 100 TFLOPS erreicht haben. Man stellt allerdings fest, dass immer öfter Rechner auf Basis von Standard-Prozessoren oder sogar Standard-Komponenten in die vorderen Ränge der Top 500 vordringen.

verteilte Systeme

Mit der zunehmenden Verbreitung von schnellen Computer-Netzwerken kom-

men auch verteilte Systeme immer mehr als Alternativen zu nachrichtengekoppelten Parallelrechnern zum Einsatz. Unter einem **verteilten System** versteht man eine Anzahl von vernetzten Rechnern, die zur Lösung einer gemeinsamen Aufgabe kooperieren können.[4] Die Vorteile solcher Systeme liegen auf der Hand:

- Oft stehen über ein lokales Netz miteinander verbundene Arbeitsplatzrechner zur Verfügung, die nur selten ausgelastet sind. Hieraus resultiert die Idee, diese Rechner gemeinsam zum „verteilten Rechnen“ zu benutzen, um Programme mit großem Rechenaufwand zu beschleunigen. Durch die Verbindung von mehreren solchen Rechnern erhält man ein einziges großes Rechenpotential.

- Vorhandene Ressourcen können durch verteiltes Rechnen besser ausgelastet werden.

- Verteilte Systeme stellen die vorhandenen Kapazitäten kostengünstig zur Verfügung, da bereits existierende Hardware genutzt wird. Um die ungenutzte Rechenleistung der vorhandenen Rechner zu erschließen, genügt es, die nötigen Programme für das verteilte Rechnen zu installieren. Multiprozessorsysteme müssten dagegen zusätzlich zu den bereits vorhandenen Arbeitsplatzrechnern beschafft werden.

- Häufig sind die neuesten Prozessoren zuerst in Arbeitsplatzrechnern und erst nach einer Zeitverzögerung in Multiprozessoren verfügbar. Da derzeit die Prozessorleistung etwa alle 18 bis 24 Monate verdoppelt wird, bietet ein vernetztes Cluster von neuesten Arbeitsplatzrechnern auch hinsichtlich der Leistung pro Prozessor Vorteile gegenüber einem Multiprozessor.

- Die Verwendung aller Speicher der verwendeten Rechner hilft, lokalen Platzmangel zu verhindern, der bei großen Datenmengen auf einem einzelnen Rechner entstehen kann.

- Ein aus vernetzten Arbeitsplatzrechnern zusammengesetzter **virtueller Parallelrechner** kann problemlos um weitere Rechner erweitert werden.

- Die Programmentwicklung kann auf gewohnten Rechnern durchgeführt werden. Mit PVM (*Parallel Virtual Machine*)und MPI (*Message Passing Interface*) stehen nachrichtengekoppelte Programmierschnittstellen zur Verfügung, die sowohl auf Multiprozessoren als auch auf sogar heterogenen, vernetzten Arbeitsplatzrechnern ablauffähig sind. Beide Programmierumgebungen zeichnen sich durch eine Quellcodekompatibilität für viele Rechner aus.

[4] Tanenbaum [15] fasst den Begriff weiter: *Nebenbei sei bemerkt, dass manche Autoren zwischen verteilten Systemen, die für die Zusammenarbeit vieler Benutzer konstruiert sind, und parallelen Systemen unterscheiden, deren einziges Ziel es ist, eine möglichst große Geschwindigkeitssteigerung bei der Lösung eines bestimmten Problems zu erreichen. Wir sind der Meinung, dass es schwierig ist, diese Unterscheidung aufrechtzuerhalten, da ein kontinuierliches Entwurfsspektrum vorliegt. Wir bevorzugen den Begriff* ***verteiltes System*** *in seiner allgemeinsten Bedeutung und verstehen darunter alle Systeme, in denen mehrere miteinander verbundene Prozessoren zusammenarbeiten.*

- Bestimmte Unterprogramme können auf speziell darauf zugeschnittenen Rechnern ausgeführt werden. Es können Spezialrechner für Graphik, Datenbanken, Vektorrechner und Multiprozessorsysteme miteinbezogen werden.

- Hinsichtlich der Fehlertoleranz bieten verteilte Systeme ebenfalls Vorteile: Fällt ein Rechner aus, bedeutet das bei fehlertoleranter Programmierung (oder einer fehlertoleranten Auslegung des Programmiersystems) nicht das Ende des gesamten Systems.

Die Verwendung von verteilten Systemen bringt aber auch einige Probleme und Nachteile mit sich:

- Die erschwerte Gewährleistung der Sicherheit ist wohl das größte Hindernis auf dem Weg zum kommerziellen Einsatz.

- Weitere Probleme entstehen beim verteilten Rechnen innerhalb heterogener Netze durch gerätespezifische Anforderungen an Hard- und Software.

- Ein großes Problem für das verteilte Rechnen ist die (zur Zeit) relativ langsame Nachrichtenübermittlung über das lokale Netz. Nur sehr grobkörnig parallele Programme mit wenig Kommunikationsanforderungen werden deshalb auf dem Rechnernetz effizient einsetzbar sein. Eine Lösung dafür kann sich durch neue Übertragungstechnologien wie die optischen Verbindungen ergeben.

Latenzzeit einer Übertragung entspricht der Übertragungszeit.

Wichtig ist für verteiltes Rechnen häufig weniger die Übertragungsbandbreite der Verbindung als die **Latenzzeit** einer Übertragung. Die **Übertragungsbandbreite** kommt nur bei hoher Last (eventuell durch nicht dezidiert zugeordnete Verbindungen) oder bei sehr großen Nachrichten zum Tragen. Die Kommunikation beim verteilten Rechnen betrifft jedoch häufig nur relativ kurze, aber dafür oft vorkommende Nachrichten und dafür ist die Latenzzeit wesentlich.

Üblicherweise werden beim verteilten Rechnen Workstations eines lokalen Netzes mittels der parallelen Programmierumgebungen PVM oder MPI betrieben. Um die hohen Kommunikationslatenzen einer lokalen Netzverbindung (wie z.B. durch Ethernet) zu umgehen, wurden in den letzten Jahren so genannte **Workstation-Farmen** oder **Cluster-Computer** entwickelt, bei denen eine Anzahl gleichartiger Workstations oder PCs durch eine zusätzliche Hochgeschwindigkeitsverbindung gekoppelt und meist dezidiert als ein virtueller Parallelrechner betrieben werden. Die Kopplungshardware besteht aus Einschubplatinen, die eine Kopplung der Workstations auf SCI-Basis (*Scalable Coherent Interface*) oder auf der Basis einer eigens entwickelten Kopplungskarte durchführen. Vorteile sind die gegenüber der Anschaffung eines Parallelrechners geringen Kosten für die Kopplungskarten und die Möglichkeit, die Workstations auch in einem normalen Benutzerbetrieb zu nutzen. Meist wird jedoch nicht die gleiche, hohe Kommunikationsgeschwindigkeit erreicht wie auf einem nachrichtengekoppelten Multiprozessor mit dem gleichen technologischen Stand. Der SCI-Standard unterstützt den Aufbau eines cachekohärenten DSM-Systems.

Jedoch werden die meisten SCI-basierten Kopplungskarten nur für eine Nachrichtenkopplung entworfen und genutzt.

Die Infrastruktur der Datenautobahnen eröffnet darüber hinaus neue Perspektiven der kooperativen Nutzung räumlich weit entfernter Parallelrechner oder die Kopplung räumlich weit entfernter Workstation-Cluster für parallele Programme. In den Campus-Netzen vieler Universitäten und sogar zwischen manchen Universitäten und Forschungseinrichtungen stehen bereits heute schnelle Datenleitungen zur Verfügung.

Meta- und Hypercomputer

Durch eine einheitliche Programmierumgebung (meist PVM oder MPI), die sowohl auf nachrichtengekoppelten Multiprozessoren verschiedener Hersteller als auch für die Kopplung von Workstations innerhalb eines Clusters einsetzbar ist, wird auch die Kopplung entfernter Parallelrechner durch das so genannte **Metacomputing** möglich. Als **Metacomputer** bezeichnet man die logische Integration eigenständiger und über eine Hochgeschwindigkeitsverbindung gekoppelter Rechner zu einem virtuellen System. Die Kopplung von Workstations aus verschiedenen lokalen Netzen auch über weite Entfernungen hinweg wird als **Remote Message Passing** (*RMP*), als **Hypercomputing** und eher selten ebenfalls als *Metacomputing* bezeichnet.

Im Regelfall kann man einen Metacomputer oder Hypercomputer nicht als eine zeitbeständige technisch-organisatorische Einheit betrachten. Vielmehr wird ein Meta- oder Hypercomputer gemäß der Anforderungen eines Anwenders für die Zeitspanne, die für die Berechnungen nötig sind, aus bestehenden Ressourcen organisiert, welche nach Ende der Rechnungen wieder freigegeben werden.

The Grid

In letzter Zeit wird für den Zusammenschluss von räumlich getrennten Rechnern (aller Arten) über ein Weitverkehrsnetz gerne das Schlagwort „**The Grid**“ benutzt.

4.4.2 Cluster Computer

Cluster-Computer

Neben den Multiprozessoren werden auch die **Cluster-Computer** zunehmend Bedeutung erhalten. Cluster Computer zeichnen sich durch die Verwendung von Standardkomponenten nach einem Baukastensystem aus. Typischerweise werden Standard-PC- oder Workstation-Platinen verwendet und mit untereinander vernetzten Switch-Einschub-Platinen kombiniert.

Dabei werden auf der Basis von Standardkomponenten so genannte *Cluster of Workstations* (COW), *Pile of PCs* (PoPC) oder *Local-area Multicomputer* (LAM) gebildet, die sowohl als (nachrichtengekoppelter) Multiprozessor als auch einzeln als Arbeitsplatzrechner betrieben werden können. Dabei können entweder schnelle Netztechnologien wie Fast Ethernet oder ATM für eine schnelle Kommunikation zum Einsatz kommen, oder es kann eine spezielle Hochgeschwindigkeitsverbindung vorgesehen sein. Diese Hochgeschwindigkeitsverbindung besteht hardwaremäßig zusätzlich neben der lokalen Netzverbindung und ist ausschließlich für den Parallelrechnerbetrieb der Workstations vorgesehen. Gestützt wird die Entwicklung zu separaten Hochgeschwindigkeitsverbindungen durch das *Scalable Coherent Interface* (SCI) als Standard für die Hochgeschwindigkeitskommunikation.

Beispiel für einen Cluster-Computer

Das Linux-Cluster-System **hpcLine** der Fujitsu Siemens Computers GmbH erlaubt als Knoten den Einsatz folgender Prozessorkombinationen:

- zwei Intel Pentium III-Prozessoren
- ein Intel Pentium 4-Prozessor
- zwei Intel Pentium Xeon-Prozessoren
- zwei AMD Athlon MP-Prozessoren

Als Netzwerk-Karten werden SCI-Karten verwendet werden. Das SCI-Netzwerk basiert auf einem Ringnetz, das zu einem zwei- oder dreidimensionalen Torus-Netzwerk ausgebaut werden kann. Die Latenz für eine Nachrichtenübertragung wird mit 5 μs und die Kommunikationsbandbreite wird pro SCI-Ring mit 5,3 GB/s angegeben.

4.5 Lösungen zu den Selbsttestaufgaben

Selbsttestaufgabe 4.1 von Seite 200

Gegeben sei ein Multiprozessorsystem mit 16 Prozessoren. Die Leistungssteigerung bezüglich eines Einprozessorsystems sei $S(16) = 4$. Die Ausführungszeit auf dem Einprozessorsystem sei $T(1) = 32$ und die Anzahl der auszuführenden (Einheits-) Operationen auf dem Multiprozessorsystem sei $P(16) = 40$.

1. Geben Sie die Effizienz E(16) an.
2. Berechnen Sie die Ausführungszeit des Multiprozessorsystems T(16).
3. Geben Sie den Parallelindex I(16) an.
4. Welcher Bruchteil der Programme ist nach Amdahls Gesetz nur sequentiell ausführbar?
5. Durch welchen Wert wird die Leistungssteigerung begrenzt, wenn man die Anzahl der Prozessoren beliebig erhöht?

Lösung:

1. Die Effizienz gibt die relative Verbesserung der Verarbeitungsgeschwindigkeit an. Wenn man mit der 16-fachen Anzahl Prozessoren nur die 4-fache Geschwindigkeit erzielt, ist die Effizienz = 1/4.

$$E(n) = S(n)/n = 4/16 = 1/4$$

2. Nach Formel gilt:

$$S(n) = T(1)/T(n) \Rightarrow T(16) = T(1)/S(16) = 32/4 = 8$$

3. Der Parallelindex gibt die Anzahl der parallelen Operationen (16) an. Die Maschine muss 40 Operationen ausführen und benötigt dafür 8 Einheiten.

$$I(n) = P(n)/T(n) = P(16)/T(16) = 40/8 = 5$$

4. Amdahls Gesetz besagt, dass ein bestimmter Prozentsatz a des Programms nur sequentiell bearbeitet werden kann.

$$T(n) = T(1)a + (T(1)(1 - a))/n$$

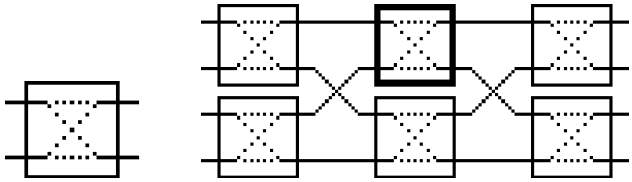
Also folgt

$$a = nT(n) - T(1)/(n - 1)T(1) = 1/5$$

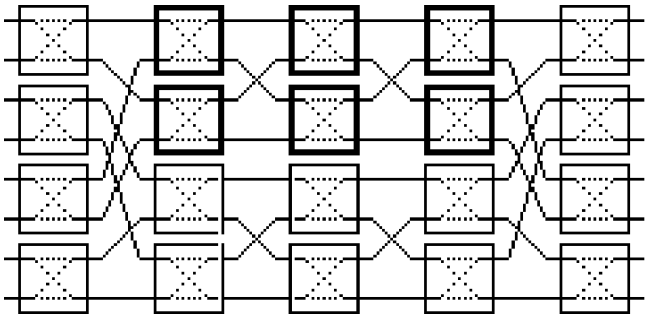
5. Erhöht man die Anzahl der Prozessoren beliebig, so kann die Ausführungszeit T nicht kleiner werden als die Zeit, die für den sequentiell abzuarbeitenden Teil a benötigt wird. Die maximale Beschleunigung ist demzufolge 5.

Selbsttestaufgabe 4.2 von Seite 212

Um mehrere Prozessoren eines Multiprozessorsystems miteinander zu vernetzen kann man Verbindungsnetzwerke aus Zweierschaltern verwenden. Das Benesnetzwerk ist ein solches Verbindungsnetzwerk mit 2^d Eingängen und 2^d Ausgängen. Es hat folgende Struktur:



$d = 1$ $d = 2$



$d = 3$

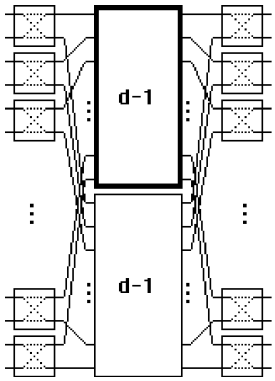


Abbildung 4.13: Rekursiver Aufbau

a) Wieviele Zeilen und Spalten hat ein Benesnetzwerk der Ordnung d?

b) Zeigen Sie, dass das Benesnetzwerk universell ist, d.h. jede Permutation erzeugen kann. Es empfiehlt sich ein konstruktiver Induktionsbeweis über d.

c) Konstruieren Sie für $d = 3$ eine Schalterstellung, welche die Permutation $(3, 0, 1, 2, 4, 6, 5, 7)$ erzeugt.

Lösung:

a) #Zeilen $= 2^{d-1}$, #Spalten $= 2d - 1$

b) Induktionsbeweis.

$d = 1$: stimmt (trivial)

$d - 1 \Rightarrow d$:

Annahme: Ein Netzwerk der Ordnung $d - 1$ kann alle Permutationen über 2^{d-1} erzeugen.

Möchte man eine vorgegebene Permutation zu einem Netzwerk der Größe d erzeugen, geht man folgendermaßen vor:

O.B.d.A. stelle man den Kreuzschalter (Tauscher) links oben auf parallel". Damit geht der Pfad des Eingangs 0 durch das obere Subnetzwerk. (Jeder Tauscher der äußeren Spalten hat eine Verbindung zum oberen und eine zum unteren Subnetzwerk) Man lege damit die Stellung des Tauscher in der rechten Spalte fest, der zum entsprechenden Ausgang $P(0)$ gehört. Als nächstes gehe man vom zweiten Ausgang dieses rechten Tauschers aus, und suche den entsprechenden Eingang auf der linken Seite. Dieser Pfad führt durch das untere Subnetzwerk. Damit lege man einen zweiten Tauscher in der linken Spalte fest. Nun nehme ich den anderen Eingang dieses Tauschers u.s.w.

Mit diesem Zick-Zack-Verfahren werden solange Tauscher in den äußeren Spalten festgelegt, bis man wieder beim linken oberen Tauscher ankommt.

Sind dann noch nicht alle Tauscher in der linken und rechten Spalte festgelegt, so wähle ich den noch freien Eingang mit dem kleinsten Index aus, stelle den zugehörigen Tauscher wieder auf parallel und setze das Verfahren fort.

Sind alle Tauscher in der linken und rechten Reihe festgelegt, so wird das Verfahren rekursiv auf die Subnetzwerke angewandt.

Auf dem Weg von links nach rechts wird jeweils das obere Netzwerk durchlaufen, auf dem Rückweg das untere.

Warum kann es nicht zu Verklemmungen kommen?

Auf der linken Seite ist immer nur ein Eingang an einem schon festgelegten Tauscher frei (im ersten Schritt ist dies der Eingang 1). Wird dieser irgendwann auf einem Rückweg festgelegt so steht der zugehörige linke Tauscher auf jeden Fall richtig, denn der Pfad dieses Eingangs führt durch das untere Subnetzwerk.

Auf der rechten Seite wird immer, nachdem ein Tauscher festgelegt wurde der andere Pfad festgelegt, der durch diesen Tauscher führt. Es gehören daher immer alle öffenenÄusgänge auch zu noch nicht festgelegten Tauschern.

Anmerkung: Das Verfahren wird am Beispiel aus Teilaufgabe c deutlich.

c) Zuerst veranschaulicht man sich die Permutation graphisch:

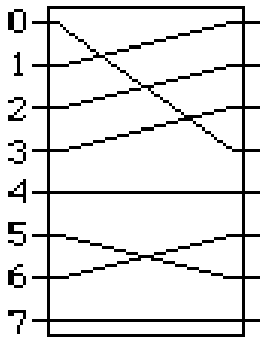


Abbildung 4.14:

Die äußeren Spalten werden im beschriebenen Verfahren konstruiert. Nummerierung der Wege in der Reihenfolge, wie sie im Algorithmus erzeugt werden. Die Wege 0 → 3 und 2 → 1 sind von links nach rechts, also vom Eingang zum Ausgang erzeugt, die Wege 1→ 0 und 3→ 2 umgekehrt.

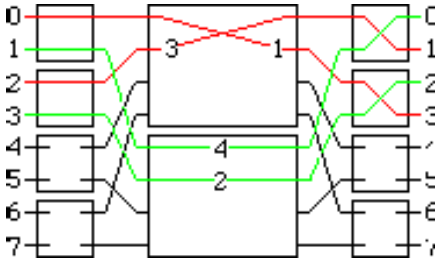


Abbildung 4.15:

Wendet man das Verfahren auf die Unternetzwerke rekursiv an, so kommt man zu folgendem Ergebnis:

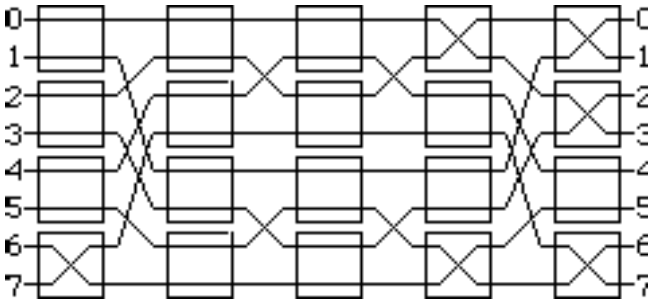


Abbildung 4.16:

Literaturverzeichnis

[1] H. Bähring. *Mikrorechner-Technik*. Springer-Verlag, Berlin, 2002.

[2] T. Beierlein and O. Hagenbruch. *Taschenbuch Mikroprozessortechnik*. Carl Hanser Verlag, München, Wien, 2001.

[3] U. Brinkschulte and T. Ungerer. *Mikrocontroller und Mikroprozessoren*. Springer-Verlag, 2002.

[4] A.W. Burks, H.H. Goldstine, and J. von Neumann. Preliminary discussion of the logical design of an electronic computing instrument. *Report to the U.S. Army Ordnance Department*, 1946. Nachgedruckt in: Aspray W, Burks A (Hrsg.) (1987) Papers of John von Neumann. The MIT Press, Cambridge, Mass., 97 – 146.

[5] K. Diefendorff and M. Allen. Organization of the motorola 88110 superscalar risc microprocessor. *IEEE Micro*, April:40 – 63, 1992.

[6] Th. Flik. *Mikroprozessortechnik*. Springer-Verlag, Berlin, 2001.

[7] K. et al. Gharachorloo. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *17th Annual International Symposium on Computer Architecture*, 1990.

[8] WK Giloi. *Rechnerarchitektur*. Springer-Verlag, 1993.

[9] J.L. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, Inc., San Mateo, California, 2003.

[10] K. Hwang. *Advanced Computer Architecture*. McGraw-Hill-Verlag, 1993.

[11] M. Johnson. *Superscalar Design*. Englewood Cliffs, New Jersey, Prentice Hall, 1991.

[12] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, Band C-28, Heft 9:690–691, 1979.

[13] S-T. Pan, K. So, and J.T Rahmeh. Improving the accuracy of dynamic branch prediction using branch correlation. *ASPLOS-V, Boston*, April:76 – 84, 1992.

[14] D. A. Patterson and J. L. Hennessy. *Rechnerorganisation und -entwurf*. Elsevier, 2005.

[15] A.S Tanenbaum. *Moderne Betriebssysteme*. Hanser-Verlag, 1994.

[16] J. Šilc, B. Robiè, and T. Ungerer. *Processor Architecture – From Dataflow to Superscalar and Beyond*. Springer-Verlag, Berlin, Heidelberg, New York, 1999.

[17] T-Y. Yeh and Y.N Patt. Alternative implementation of two-level adaptive branch prediction. In *ISCA-19*, pages 124 – 134, Gold Coast, Australien, Mai 1992.

Index